Coconut

Release develop

Evan Hubinger

May 13, 2024

CONTENTS

CHAPTER

ONE

COCONUT FAQ

1.1 Frequently Asked Questions

1.1.1 Can I use Python modules from Coconut and Coconut modules from Python?

Yes and yes! Coconut compiles to Python, so Coconut modules are accessible from Python and Python modules are accessible from Coconut, including the entire Python standard library.

1.1.2 What versions of Python does Coconut support?

Coconut supports any Python version `>= 2.6` on the `2.x` branch or `>= 3.2` on the `3.x` branch. In fact, Coconut code is compiled to run the same on every one of those supported versions! See *compatible Python versions* for more information.

1.1.3 Can Coconut be used to convert Python from one version to another?

Yes! But only in the backporting direction: Coconut can convert Python 3 to Python 2, but not the other way around. Coconut really can, though, turn Python 3 code into version-independent Python. Coconut will compile Python 3 syntax, built-ins, and even imports to code that will work on any supported Python version (`2.6`, `2.7`, `>=3.2`).

There a couple of caveats to this, however: Coconut can't magically make all your other third-party packages version-independent, and some constructs will require a particular `--target` to make them work (for a full list, see *compatible Python versions*).

1.1.4 How do I release a Coconut package on PyPI?

Since Coconut just compiles to Python, releasing a Coconut package on PyPI is exactly the same as releasing a Python package, with an extra compilation step. Just write your package in Coconut, run `coconut` on the source code, and upload the compiled code to PyPI. You can even mix Python and Coconut code, since the compiler will only touch `.coco` files. If you want to see an example of a PyPI package written in Coconut, including a Makefile with the exact compiler commands being used, check out bbopt.

1.1.5 I saw that Coconut was recently updated. Where is the change log?

Information on every Coconut release is chronicled on the GitHub releases page. There you can find all of the new features and breaking changes introduced in each release.

1.1.6 Does Coconut support static type checking?

Yes! Coconut compiles the newest, fanciest type annotation syntax into version-independent type comments which can then by checked using Coconut's built-in *MyPy Integration*.

1.1.7 Help! I tried to write a recursive generator and my Python segfaulted!

No problem—just use Coconut's *`recursive_generator`* decorator and you should be fine. This is a known Python issue but `recursive_generator` will fix it for you.

1.1.8 How do I split an expression across multiple lines in Coconut?

Since Coconut syntax is a superset of Python 3 syntax, Coconut supports the same line continuation syntax as Python. That means both backslash line continuation and implied line continuation inside of parentheses, brackets, or braces will all work. Parenthetical continuation is the recommended method, and Coconut even supports an *enhanced version of it*.

1.1.9 I want to use Coconut in a production environment; how do I achieve maximum performance?

First, you're going to want a fast compiler, so you should make sure you're using `cPyparsing`. Second, there are two simple things you can do to make Coconut produce faster Python: compile with `--no-tco` and compile with a `--target` specification for the exact version of Python you want to run your code on. Passing `--target` helps Coconut optimize the compiled code for the Python version you want, and, though *Tail Call Optimization* is useful, it will usually significantly slow down functions that use it, so disabling it will often provide a major performance boost.

1.1.10 How do I use a runtime type checker like `beartype` when Coconut seems to compile all my type annotations to strings/comments?

First, to make sure you get actual type annotations rather than type comments, you'll need to `--target` a Python version that supports the sorts of type annotations you'll be using (specifically `--target 3.6` should usually do the trick). Second, if you're using runtime type checking, you'll need to pass the `--no-wrap` argument, which will tell Coconut not to wrap type annotations in strings. When using type annotations for static type checking, wrapping them in strings is preferred, but when using them for runtime type checking, you'll want to disable it.

1.1.11 When I try to use Coconut on the command line, I get weird unprintable characters and numbers; how do I get rid of them?

You're probably seeing color codes while using a terminal that doesn't support them (e.g. Windows `cmd`). Try setting the `COCONUT_USE_COLOR` environment variable to `FALSE` to get rid of them.

1.1.12 How will I be able to debug my Python if I'm not the one writing it?

Ease of debugging has long been a problem for all compiled languages, including languages like `C` and `C++` that these days we think of as very low-level languages. The solution to this problem has always been the same: line number maps. If you know what line in the compiled code corresponds to what line in the source code, you can easily debug just from the source code, without ever needing to deal with the compiled code at all. In Coconut, this can easily be accomplished by passing the `--line-numbers` or `-l` flag, which will add a comment to every line in the compiled code with the number of the corresponding line in the source code. Alternatively, `--keep-lines` or `-k` will put in the verbatim source line instead of or in addition to the source line number. Then, if Python raises an error, you'll be able to see from the snippet of the compiled code that it shows you a comment telling you what line in your source code you need to look at to debug the error.

1.1.13 If I'm already perfectly happy with Python, why should I learn Coconut?

You're exactly the person Coconut was built for! Coconut lets you keep doing the thing you do well—write Python—without having to worry about annoyances like version compatibility, while also allowing you to do new cool things you might never have thought were possible before like pattern-matching and lazy evaluation. If you've ever used a functional programming language before, you'll know that functional code is often much simpler, cleaner, and more readable (but not always, which is why Coconut isn't purely functional). Python is a wonderful imperative language, but when it comes to modern functional programming—which, in Python's defense, it wasn't designed for—Python falls short, and Coconut corrects that shortfall.

1.1.14 I don't like functional programming, should I still learn Coconut?

Definitely! While Coconut is great for functional programming, it also has a bunch of other awesome features as well, including the ability to compile Python 3 code into universal Python code that will run the same on *any version*. And that's not even mentioning all of the features like pattern-matching and destructuring assignment with utility extending far beyond just functional programming. That being said, I'd highly recommend you give functional programming a shot, and since Coconut isn't purely functional, it's a great introduction to the functional style.

1.1.15 I don't know functional programming, should I still learn Coconut?

Yes, absolutely! Coconut's *tutorial* assumes absolutely no prior knowledge of functional programming, only Python. Because Coconut is not a purely functional programming language, and all valid Python is valid Coconut, Coconut is a great introduction to functional programming. If you learn Coconut, you'll be able to try out a new functional style of programming without having to abandon all the Python you already know and love.

1.1.16 I don't know Python very well, should I still learn Coconut?

Maybe. If you know the very basics of Python, and are also very familiar with functional programming, then definitely—Coconut will let you continue to use all your favorite tools of functional programming while you make your way through learning Python. If you're not very familiar either with Python, or with functional programming, then you may be better making your way through a Python tutorial before you try learning Coconut. That being said, using Coconut to compile your pure Python code might still be very helpful for you, since it will alleviate having to worry about version incompatibility.

1.1.17 Why isn't Coconut purely functional?

The short answer is that Python isn't purely functional, and all valid Python is valid Coconut. The long answer is that Coconut isn't purely functional for the same reason Python was never purely imperative—different problems demand different approaches. Coconut is built to be *useful*, and that means not imposing constraints about what style the programmer is allowed to use. That being said, Coconut is built specifically to work nicely when programming in a functional style, which means if you want to write all your code purely functionally, Coconut will make it a smooth experience, and allow you to have good-looking code to show for it.

1.1.18 Won't a transpiled language like Coconut be bad for the Python community?

I certainly hope not! Unlike most transpiled languages, all valid Python is valid Coconut. Coconut's goal isn't to replace Python, but to *extend* it. If a newbie learns Coconut, it won't mean they have a harder time learning Python, it'll mean they *already know* Python. And not just any Python, the newest and greatest—Python 3. And of course, Coconut is perfectly interoperable with Python, and uses all the same libraries—thus, Coconut can't split the Python community, because the Coconut community *is* the Python community.

1.1.19 I want to contribute to Coconut, how do I get started?

That's great! Coconut is completely open-source, and new contributors are always welcome. Check out Coconut's *contributing guidelines* for more information.

1.1.20 Why the name Coconut?

If you don't get the reference, the image above is from Monty Python and the Holy Grail, in which the Knights of the Round Table bang Coconuts together to mimic the sound of riding a horse. The name was chosen to reference the fact that Python is named after Monty Python as well.

1.1.21 Who developed Coconut?

Evan Hubinger is an AI safety research scientist at Anthropic. He can be reached by asking a question on Coconut's Gitter chat room, through email at evanjhub@gmail.com, or on LinkedIn.

CHAPTER

TWO

COCONUT TUTORIAL

2.1 Introduction

Welcome to the tutorial for the Coconut Programming Language! Coconut is a variant of Python built for **simple, elegant, Pythonic functional programming**. But those are just words; what they mean in practice is that *all valid Python 3 is valid Coconut* but Coconut builds on top of Python a suite of *simple, elegant utilities for functional programming*.

Why use Coconut? Coconut is built to be useful. Coconut enhances the repertoire of Python programmers to include the tools of modern functional programming, in such a way that those tools are *easy* to use and immensely *powerful;* that is, Coconut does to functional programming what Python did to imperative programming. And Coconut code runs the same on *any Python version*, making the Python 2/3 split a thing of the past.

Specifically, Coconut adds to Python *built-in, syntactical support* for:

- pattern-matching
- algebraic data types
- destructuring assignment
- partial application
- lazy lists
- function composition
- prettier lambdas
- infix notation
- pipeline-style programming
- operator functions
- tail call optimization
- where statements

and much more!

2.1.1 Interactive Tutorial

This tutorial is non-interactive. To get an interactive tutorial instead, check out Coconut's interactive tutorial.

Note, however, that the interactive tutorial is less up-to-date than this one and thus may contain old, deprecated syntax (though Coconut will let you know if you encounter such a situation) as well as outdated idioms (meaning that the example code in the interactive tutorial is likely to be much less elegant than the example code here).

2.1.2 Installation

At its very core, Coconut is a compiler that turns Coconut code into Python code. That means that anywhere where you can use a Python script, you can also use a compiled Coconut script. To access that core compiler, Coconut comes with a command-line utility, which can

- compile single Coconut files or entire Coconut projects,
- interpret Coconut code on-the-fly, and
- hook into existing Python applications like IPython/Jupyter and MyPy.

Installing Coconut, including all the features above, is drop-dead simple. Just

1. install Python,
2. open a command-line prompt,
3. and enter:

```
pip install coconut
```

Note: If you are having trouble installing Coconut, try following the debugging steps in the installation section of Coconut's documentation.

To check that your installation is functioning properly, try entering into the command line

```
coconut -h
```

which should display Coconut's command-line help.

Note: If you're having trouble, or if anything mentioned in this tutorial doesn't seem to work for you, feel free to ask for help on Gitter and somebody will try to answer your question as soon as possible.

2.2 Starting Out

2.2.1 Using the Interpreter

Now that you've got Coconut installed, the obvious first thing to do is to play around with it. To launch the Coconut interpreter, just go to the command line and type

```
coconut
```

and you should see something like

```
Coconut Interpreter vX.X.X:
(enter 'exit()' or press Ctrl-D to end)
>>>
```

which is Coconut's way of telling you you're ready to start entering code for it to evaluate. So let's do that!

In case you missed it earlier, *all valid Python 3 is valid Coconut*. That doesn't mean compiled Coconut will only run on Python 3—in fact, compiled Coconut will run the same on any Python version—but it does mean that only Python 3 code is guaranteed to compile as Coconut code.

That means that if you're familiar with Python, you're already familiar with a good deal of Coconut's core syntax and Coconut's entire standard library. To show that, let's try entering some basic Python into the Coconut interpreter. For example:

```
>>> "hello, world!"
'hello, world!'
>>> 1 + 1
2
```

2.2.2 Writing Coconut Files

Of course, while being able to interpret Coconut code on-the-fly is a great thing, it wouldn't be very useful without the ability to write and compile larger programs. To that end, it's time to write our first Coconut program: "hello, world!" Coconut-style.

First, we're going to need to create a file to put our code into. The file extension for Coconut source files is `.coco`, so let's create the new file `hello_world.coco`. After you do that, you should take the time now to set up your text editor to properly highlight Coconut code. For instructions on how to do that, see the documentation on *Coconut syntax highlighting*.

Now let's put some code in our `hello_world.coco` file. Unlike in Python, where headers like

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
from __future__ import print_function, absolute_import, unicode_literals, division
```

are common and often very necessary, the Coconut compiler will automatically take care of all of that for you, so all you need to worry about is your own code. To that end, let's add the code for our "hello, world!" program.

In pure Python 3, "hello, world!" is

```
print("hello, world!")
```

and while that will work in Coconut, equally as valid is to use a pipeline-style approach, which is what we'll do, and write

```
"hello, world!" |> print
```

which should let you see very clearly how Coconut's `|>` operator enables pipeline-style programming: it allows an object to be passed along from function to function, with a different operation performed at each step. In this case, we are piping the object `"hello, world!"` into the operation `print`. Now let's save our simple "hello, world!" program, and try to run it.

2.2.3 Using the Compiler

Compiling Coconut files and projects with the Coconut command-line utility is incredibly simple. Just `cd` into the directory of your `hello_world.coco` file and type

```
coconut hello_world.coco
```

which should give the output

```
Coconut: Compiling       hello_world.coco ...
Coconut: Compiled to     hello_world.py .
```

and deposit a new `hello_world.py` file in the same directory as the `hello_world.coco` file. You should then be able to run that file with

```
python hello_world.py
```

which should produce `hello, world!` as the output.

Note: You can compile and run your code all in one step if you use Coconut's `--run` *option (*`-r` *for short).*

Compiling single files is not the only way to use the Coconut command-line utility, however. We can also compile all the Coconut files in a given directory simply by passing that directory as the first argument, which will get rid of the need to run the same Coconut header code in each file by storing it in a `__coconut__.py` file in the same directory.

The Coconut compiler supports a large variety of different compilation options, the help for which can always be accessed by entering `coconut -h` into the command line. One of the most useful of these is `--line-numbers` (or `-l` for short). Using `--line-numbers` will add the line numbers of your source code as comments in the compiled code, allowing you to see what line in your source code corresponds to a line in the compiled code where an error occurred, for ease of debugging.

Note: If you don't need the full control of the Coconut compiler, you can also access your Coconut code just by importing it*, either from the Coconut interpreter, or in any Python file where you import* `coconut.api`.

2.2.4 Using IPython/Jupyter

Although all different types of programming can benefit from using more functional techniques, scientific computing, perhaps more than any other field, lends itself very well to functional programming, an observation the case studies in this tutorial are very good examples of. That's why Coconut aims to provide extensive support for the established tools of scientific computing in Python.

To that end, Coconut provides *built-in IPython/Jupyter support*. To launch a Jupyter notebook with Coconut, just enter the command

```
coconut --jupyter notebook
```

Alternatively, to launch the Jupyter interpreter with Coconut as the kernel, run `coconut --jupyter console` *instead. Additionally, you can launch an interactive Coconut Jupyter console initialized from the current namespace by inserting* `from coconut import embed; embed()` *into your code, which can be a very useful debugging tool.*

2.2.5 Case Studies

Because Coconut is built to be useful, the best way to demo it is to show it in action. To that end, the majority of this tutorial will be showing how to apply Coconut to solve particular problems, which we'll call case studies.

These case studies are not intended to provide a complete picture of all of Coconut's features. For that, see Coconut's *documentation*. Instead, they are intended to show how Coconut can actually be used to solve practical programming problems.

2.3 Case Study 1: `factorial`

In the first case study we will be defining a `factorial` function, that is, a function that computes `n!` where `n` is an integer `>= 0`. This is somewhat of a toy example, since Python can fairly easily do this, but it will serve as a good showcase of some of the basic features of Coconut and how they can be used to great effect.

To start off with, we're going to have to decide what sort of an implementation of `factorial` we want. There are many different ways to tackle this problem, but for the sake of concision we'll split them into four major categories: imperative, recursive, iterative, and `addpattern`.

2.3.1 Imperative Method

The imperative approach is the way you'd write `factorial` in a language like C. Imperative approaches involve lots of state change, where variables are regularly modified and loops are liberally used. In Coconut, the imperative approach to the `factorial` problem looks like this:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    if n `isinstance` int and n >= 0:
        acc = 1
        for x in range(1, n+1):
            acc *= x
        return acc
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Before we delve into what exactly is happening here, let's give it a run and make sure the test cases check out. If we were really writing a Coconut program, we'd want to save and compile an actual file, but since we're just playing around, let's try copy-pasting into the interpreter. Here, you should get two `TypeErrors`, then `1`, then `6`.

Now that we've verified it works, let's take a look at what's going on. Since the imperative approach is a fundamentally non-functional method, Coconut can't help us improve this example very much. Even here, though, the use of Coconut's infix notation (where the function is put in-between its arguments, surrounded in backticks) in ``n `isinstance` int`` makes the code slightly cleaner and easier to read.

2.3.2 Recursive Method

The recursive approach is the first of the fundamentally functional approaches, in that it doesn't involve the state change and loops of the imperative approach. Recursive approaches avoid the need to change variables by making that variable change implicit in the recursive function call. Here's the recursive approach to the `factorial` problem in Coconut:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    match n:
        case 0:
            return 1
        case x `isinstance` int if x > 0:
            return x * factorial(x-1)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Go ahead and copy and paste the code and tests into the interpreter. You should get the same test results as you got for the imperative version—but you can probably tell there's quite a lot more going on here than there. That's intentional: Coconut is intended for functional programming, not imperative programming, and so its new features are built to be most useful when programming in a functional style.

Let's take a look at the specifics of the syntax in this example. The first thing we see is `match n`. This statement starts a `case` block, in which only `case` statements can occur. Each `case` statement will attempt to match its given pattern against the value in the `case` block. Only the first successful match inside of any given `case` block will be executed. When a match is successful, any variable bindings in that match will also be performed. Additionally, as is true in this case, `case` statements can also have `if` guards that will check the given condition before the match is considered final. Finally, after the `case` block, an `else` statement is allowed, which will only be executed if no `case` statement is.

Specifically, in this example, the first `case` statement checks whether `n` matches to `0`. If it does, it executes `return 1`. Then the second `case` statement checks whether `n` matches to ``x `isinstance` int``, which checks that `n` is an `int` (using `isinstance`) and assigns `x = n` if so, then checks whether `x > 0`, and if so, executes `return x * factorial(x-1)`. If neither of those two statements are executed, the `else` statement triggers and executes `raise TypeError("the argument to factorial must be an integer >= 0")`.

Although this example is very basic, pattern-matching is both one of Coconut's most powerful and most complicated features. As a general intuitive guide, it is helpful to think *assignment* whenever you see the keyword `match`. A good way to showcase this is that all `match` statements can be converted into equivalent destructuring assignment statements, which are also valid Coconut. In this case, the destructuring assignment equivalent to the `factorial` function above would be:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    try:
        # The only value that can be assigned to 0 is 0, since 0 is an
        # immutable constant; thus, this assignment fails if n is not 0.
        0 = n
    except MatchError:
        pass
    else:
```

(continues on next page)

(continued from previous page)

```
        return 1
    try:
        # This attempts to assign n to x, which has been declared to be
        # an int; since only an int can be assigned to an int, this
        # fails if n is not an int.
        x `isinstance` int = n
    except MatchError:
        pass
    else: if x > 0:  # in Coconut, statements can be nested on the same line
        return x * factorial(x-1)
    raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

First, copy and paste! While this destructuring assignment equivalent should work, it is much more cumbersome than `match` statements when you expect that they'll fail, which is why `match` statement syntax exists. But the destructuring assignment equivalent illuminates what exactly the pattern-matching is doing, by making it clear that `match` statements are really just fancy destructuring assignment statements. In fact, to be explicit about using destructuring assignment instead of normal assignment, the `match` keyword can be put before a destructuring assignment statement to signify it as such.

It will be helpful to, as we continue to use Coconut's pattern-matching and destructuring assignment statements in further examples, think *assignment* whenever you see the keyword `match`.

Next, we can make a couple of simple improvements to our `factorial` function. First, we don't actually need to assign `x` as a new variable, since it has the same value as `n`, so if we use `_` instead of `x`, Coconut won't ever actually assign the variable. Thus, we can rewrite our `factorial` function as:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    match n:
        case 0:
            return 1
        case _ `isinstance` int if n > 0:
            return n * factorial(n-1)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Copy, paste! This new `factorial` function should behave exactly the same as before.

Second, we can replace the ``_ `isinstance` int`` pattern with the class pattern `int()`, which, when used with no arguments like that, is equivalent. Thus, we can again rewrite our `factorial` function to:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    match n:
        case 0:
            return 1
        case int() if n > 0:
            return n * factorial(n-1)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Up until now, for the recursive method, we have only dealt with pattern-matching, but there's actually another way that Coconut allows us to improve our `factorial` function. Coconut performs automatic tail call optimization, which means that whenever a function directly returns a call to another function, Coconut will optimize away the additional call. Thus, we can improve our `factorial` function by rewriting it to use a tail call:

```
def factorial(n, acc=1):
    """Compute n! where n is an integer >= 0."""
    match n:
        case 0:
            return acc
        case int() if n > 0:
            return factorial(n-1, acc*n)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Copy, paste! This new `factorial` function is equivalent to the original version, with the exception that it will never raise a `RuntimeError` due to reaching Python's maximum recursion depth, since Coconut will optimize away the tail call.

2.3.3 Iterative Method

The other main functional approach is the iterative one. Iterative approaches avoid the need for state change and loops by using higher-order functions, those that take other functions as their arguments, like `map` and `reduce`, to abstract out the basic operations being performed. In Coconut, the iterative approach to the `factorial` problem is:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    match n:
        case 0:
            return 1
```

(continues on next page)

(continued from previous page)

```
        case int() if n > 0:
            return range(1, n+1) |> reduce$(*)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Copy, paste! This definition differs from the recursive definition only by one line. That's intentional: because both the iterative and recursive approaches are functional approaches, Coconut can provide a great assist in making the code cleaner and more readable. The one line that differs is this one:

```
return range(1, n+1) |> reduce$(*)
```

Let's break down what's happening on this line. First, the `range` function constructs an iterator of all the numbers that need to be multiplied together. Then, it is piped into the function `reduce$(*)`, which does that multiplication. But how? What is `reduce$(*)`?

We'll start with the base, the `reduce` function. `reduce` used to exist as a built-in in Python 2, and Coconut brings it back. `reduce` is a higher-order function that takes a function of two arguments as its first argument, and an iterator as its second argument, and applies that function to the given iterator by starting with the first element, and calling the function on the accumulated call so far and the next element, until the iterator is exhausted. Here's a visual representation:

```
reduce(f, (a, b, c, d))

acc                 iter
                    (a, b, c, d)
a                   (b, c, d)
f(a, b)             (c, d)
f(f(a, b), c)       (d)
f(f(f(a, b), c), d)

return acc
```

Now let's take a look at what we do to `reduce` to make it multiply all the numbers we feed into it together. The Coconut code that we saw for that was `reduce$(*)`. There are two different Coconut constructs being used here: the operator function for multiplication in the form of `(*)`, and partial application in the form of `$`.

First, the operator function. In Coconut, a function form of any operator can be retrieved by surrounding that operator in parentheses. In this case, `(*)` is roughly equivalent to `lambda x, y:  x*y`, but much cleaner and neater. In Coconut's lambda syntax, `(*)` is also equivalent to `(x, y) => x*y`, which we will use from now on for all lambdas, even though both are legal Coconut, because Python's `lambda` statement is too ugly and bulky to use regularly.

Note: If Coconut's `--strict` mode is enabled, which will force your code to obey certain cleanliness standards, it will raise an error whenever Python `lambda` statements are used.

Second, the partial application. Think of partial application as *lazy function calling*, and `$` as the *lazy-ify* operator, where lazy just means "don't evaluate this until you need to." In Coconut, if a function call is prefixed by a `$`, like in this example, instead of actually performing the function call, a new function is returned with the given arguments already provided to it, so that when it is then called, it will be called with both the partially-applied arguments and the new arguments, in that order. In this case, `reduce$(*)` is roughly equivalent to `(*args, **kwargs) => reduce((*), *args, **kwargs)`.

You can partially apply arguments in any order using ? in place of missing arguments, as in `to_binary = int$(?, 2)`.

Putting it all together, we can see how the single line of code

```
range(1, n+1) |> reduce$(*)
```

is able to compute the proper factorial, without using any state or loops, only higher-order functions, in true functional style. By supplying the tools we use here like partial application (`$`), pipeline-style programming (`|>`), higher-order functions (`reduce`), and operator functions (`(*)`), Coconut enables this sort of functional programming to be done cleanly, neatly, and easily.

2.3.4 `addpattern` Method

While the iterative approach is very clean, there are still some bulky pieces—looking at the iterative version below, you can see that it takes three entire indentation levels to get from the function definition to the actual objects being returned:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    match n:
        case 0:
            return 1
        case int() if n > 0:
            return range(1, n+1) |> reduce$(*)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")
```

By making use of the *Coconut `addpattern` syntax*, we can take that from three indentation levels down to one. Take a look:

```
def factorial(0) = 1

addpattern def factorial(int() as n if n > 0) =
    """Compute n! where n is an integer >= 0."""
    range(1, n+1) |> reduce$(*)

# Test cases:
-1 |> factorial |> print  # MatchError
0.5 |> factorial |> print  # MatchError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Copy, paste! This should work exactly like before, except now it raises `MatchError` as a fall through instead of `TypeError`. There are three major new concepts to talk about here: `addpattern`, of course, assignment function notation, and pattern-matching function definition—how both of the functions above are defined.

First, assignment function notation. This one's pretty straightforward. If a function is defined with an `=` instead of a `:`, the last line is required to be an expression, and is automatically returned.

Second, pattern-matching function definition. Pattern-matching function definition does exactly that—pattern-matches against all the arguments that are passed to the function. Unlike normal function definition, however, if the pattern doesn't match (if for example the wrong number of arguments are passed), your function will raise a `MatchError`. Finally, like destructuring assignment, if you want to be more explicit about using pattern-matching function definition, you can add a match before the def. In this case, we're also using one new pattern-matching construct, the as match, which matches against the pattern on the left and assigns the result to the name on the right.

Third, addpattern. addpattern creates a new pattern-matching function by adding the new pattern as an additional case to the old pattern-matching function it is replacing. Thus, addpattern can be thought of as doing exactly what it says—it adds a new pattern to an existing pattern-matching function.

Finally, not only can we rewrite the iterative approach using addpattern, as we did above, we can also rewrite the recursive approach using addpattern, like so:

```
def factorial(0) = 1

addpattern def factorial(int() as n if n > 0) =
    """Compute n! where n is an integer >= 0."""
    n * factorial(n - 1)

# Test cases:
-1 |> factorial |> print  # MatchError
0.5 |> factorial |> print  # MatchError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Copy, paste! It should work exactly like before, except, as above, with TypeError replaced by MatchError.

2.4 Case Study 2: quick_sort

In the second case study, we will be implementing the quick sort algorithm. We will implement two versions: first, a quick_sort function that takes in a list and outputs a list, and second, a quick_sort function that takes in an iterator and outputs an iterator.

2.4.1 Sorting a Sequence

First up is quick_sort for lists. We're going to use a recursive addpattern-based approach to tackle this problem—a similar approach to the very last factorial function we wrote, using addpattern to reduce the amount of indentation we're going to need. Without further ado, here's our implementation of quick_sort for lists:

```
def quick_sort([]) = []

addpattern def quick_sort([head] + tail) =
    """Sort the input sequence using the quick sort algorithm."""
    quick_sort(left) + [head] + quick_sort(right) where:
        left = [x for x in tail if x < head]
        right = [x for x in tail if x >= head]

# Test cases:
[] |> quick_sort |> print  # []
[3] |> quick_sort |> print  # [3]
[0,1,2,3,4] |> quick_sort |> print  # [0,1,2,3,4]
[4,3,2,1,0] |> quick_sort |> print  # [0,1,2,3,4]
[3,0,4,2,1] |> quick_sort |> print  # [0,1,2,3,4]
```

Copy, paste! Two new feature here: head-tail pattern-matching and where statements.

First, `where` statements are extremely straightforward. In fact, I bet you've already figured out what they do from the code above. A `where` statement is just a way to compute something in the context of some set of assignment statements.

Second, head-tail pattern-matching, which you can see here as `[head] + tail`, simply follows the form of a list or tuple added to a variable. When this appears in any pattern-matching context, the value being matched against will be treated as a sequence, the list or tuple matched against the beginning of that sequence, and the rest of it bound to the variable. In this case, we use the head-tail pattern to remove the head so we can use it as the pivot for splitting the rest of the list.

2.4.2 Sorting an Iterator

Now it's time to try `quick_sort` for iterators. Our method for tackling this problem is going to be a combination of the recursive and iterative approaches we used for the `factorial` problem, in that we're going to be lazily building up an iterator, and we're going to be doing it recursively. Here's the code:

```
def quick_sort(l):
    """Sort the input iterator using the quick sort algorithm."""
    match [head] :: tail in l:
        tail = reiterable(tail)
        yield from quick_sort(left) :: [head] :: quick_sort(right) where:
            left = (x for x in tail if x < head)
            right = (x for x in tail if x >= head)
    # By yielding nothing if the match falls through, we implicitly return an empty␣
↪iterator.

# Test cases:
[] |> quick_sort |> list |> print  # []
[3] |> quick_sort |> list |> print  # [3]
[0,1,2,3,4] |> quick_sort |> list |> print  # [0,1,2,3,4]
[4,3,2,1,0] |> quick_sort |> list |> print  # [0,1,2,3,4]
[3,0,4,2,1] |> quick_sort |> list |> print  # [0,1,2,3,4]
```

Copy, paste! This `quick_sort` algorithm works uses a bunch of new constructs, so let's go over them.

First, the `::` operator, which appears here both in pattern-matching and by itself. In essence, the `::` operator is lazy + for iterators. On its own, it takes two iterators and concatenates, or chains, them together, and it does this lazily, not evaluating anything until its needed, so it can be used for making infinite iterators. In pattern-matching, it inverts that operation, destructuring the beginning of an iterator into a pattern, and binding the rest of that iterator to a variable.

Which brings us to the second new thing, `match ... in ...` notation. The notation

```
match pattern in item:
    <body>
else:
    <else>
```

is shorthand for

```
match item:
    case pattern:
        <body>
else:
    <else>
```

that avoids the need for an additional level of indentation when only one `match` is being performed.

The third new construct is the *Coconut built-in `reiterable`*. There is a problem in doing immutable functional programming with Python iterators: whenever an element of an iterator is accessed, it's lost. `reiterable` solves this problem by allowing the iterable it's called on to be iterated over multiple times while still yielding the same result each time

Finally, although it's not a new construct, since it exists in Python 3, the use of `yield from` here deserves a mention. In Python, `yield` is the statement used to construct iterators, functioning much like `return`, with the exception that multiple `yield`s can be encountered, and each one will produce another element. `yield from` is very similar, except instead of adding a single element to the produced iterator, it adds another whole iterator.

Putting it all together, here's our `quick_sort` function again:

```
def quick_sort(l):
    """Sort the input iterator using the quick sort algorithm."""
    match [head] :: tail in l:
        tail = reiterable(tail)
        yield from quick_sort(left) :: [head] :: quick_sort(right) where:
            left = (x for x in tail if x < head)
            right = (x for x in tail if x >= head)
    # By yielding nothing if the match falls through, we implicitly return an empty␣
↪iterator.
```

The function first attempts to split `l` into an initial element and a remaining iterator. If `l` is the empty iterator, that match will fail, and it will fall through, yielding the empty iterator (that's how the function handles the base case). Otherwise, we make a copy of the rest of the iterator, and yield the join of (the quick sort of all the remaining elements less than the initial element), (the initial element), and (the quick sort of all the remaining elements greater than the initial element).

The advantages of the basic approach used here, heavy use of iterators and recursion, as opposed to the classical imperative approach, are numerous. First, our approach is more clear and more readable, since it is describing *what* `quick_sort` is instead of *how* `quick_sort` could be implemented. Second, our approach is *lazy* in that our `quick_sort` won't evaluate any data until it needs it. Finally, and although this isn't relevant for `quick_sort` it is relevant in many other cases, an example of which we'll see later in this tutorial, our approach allows for working with *infinite* series just like they were finite.

And Coconut makes programming in such an advantageous functional approach significantly easier. In this example, Coconut's pattern-matching lets us easily split the given iterator, and Coconut's `::` iterator joining operator lets us easily put it back together again in sorted order.

2.5 Case Study 3: `vector` Part I

In the next case study, we'll be doing something slightly different—instead of defining a function, we'll be creating an object. Specifically, we're going to try to implement an immutable n-vector that supports all the basic vector operations.

In functional programming, it is often very desirable to define *immutable* objects, those that can't be changed once created—like Python's strings or tuples. Like strings and tuples, immutable objects are useful for a wide variety of reasons:

- they're easier to reason about, since you can be guaranteed they won't change,
- they're hashable and pickleable, so they can be used as keys and serialized,
- they're significantly more efficient since they require much less overhead,
- and when combined with pattern-matching, they can be used as what are called *algebraic data types* to build up and then match against large, complicated data structures very easily.

2.5.1 2-Vector

Coconut's `data` statement brings the power and utility of *immutable, algebraic data types* to Python, and it is this that we will be using to construct our `vector` type. The demonstrate the syntax of `data` statements, we'll start by defining a simple 2-vector. Our vector will have one special method `__abs__` which will compute the vector's magnitude, defined as the square root of the sum of the squares of the elements. Here's our 2-vector:

```
data vector2(x, y):
    """Immutable 2-vector."""
    def __abs__(self) =
        """Return the magnitude of the 2-vector."""
        (self.x**2 + self.y**2)**0.5

# Test cases:
vector2(1, 2) |> print  # vector2(x=1, y=2)
vector2(3, 4) |> abs |> print  # 5
vector2(1, 2) |> fmap$(x => x*2) |> print  # vector2(x=2, y=4)
v = vector2(2, 3)
v.x = 7  # AttributeError
```

Copy, paste! This example shows the basic syntax of `data` statements:

```
data <name>(<attributes>):
    <body>
```

where `<name>` and `<body>` are the same as the equivalent `class` definition, but `<attributes>` are the different attributes of the data type, in order that the constructor should take them as arguments. In this case, `vector2` is a data type of two attributes, `x` and `y`, with one defined method, `__abs__`, that computes the magnitude. As the test cases show, we can then create, print, but *not modify* instances of `vector2`.

One other thing to call attention to here is the use of the *Coconut built-in fmap*. `fmap` allows you to map functions over algebraic data types. Coconut's `data` types do support iteration, so the standard `map` works on them, but it doesn't return another object of the same data type. In this case, `fmap` is simply `map` plus a call to the object's constructor.

2.5.2 n-Vector Constructor

Now that we've got the 2-vector under our belt, let's move to back to our original, more complicated problem: n-vectors, that is, vectors of arbitrary length. We're going to try to make our n-vector support all the basic vector operations, but we'll start out with just the `data` definition and the constructor:

```
data vector(*pts):
    """Immutable n-vector."""
    def __new__(cls, *pts):
        """Create a new vector from the given pts."""
        match [v `isinstance` vector] in pts:
            return v  # vector(v) where v is a vector should return v
        else:
            return pts |*> makedata$(cls)  # accesses base constructor

# Test cases:
vector(1, 2, 3) |> print  # vector(*pts=(1, 2, 3))
vector(4, 5) |> vector |> print  # vector(*pts=(4, 5))
```

Copy, paste! The big new thing here is how to write `data` constructors. Since `data` types are immutable, `__init__` construction won't work. Instead, a different special method `__new__` is used, which must return the newly constructed instance, and unlike most methods, takes the class not the object as the first argument. Since `__new__` needs to return a fully constructed instance, in almost all cases it will be necessary to access the underlying `data` constructor. To achieve this, Coconut provides the *built-in* `makedata` *function*, which takes a data type and calls its underlying `data` constructor with the rest of the arguments.

In this case, the constructor checks whether nothing but another `vector` was passed, in which case it returns that, otherwise it returns the result of passing the arguments to the underlying constructor, the form of which is `vector(*pts)`, since that is how we declared the data type. We use sequence pattern-matching to determine whether we were passed a single vector, which is just a list or tuple of patterns to match against the contents of the sequence.

One important pitfall that's worth pointing out here: in this case, you must use ``v `isinstance` vector`` rather than `vector() as v`, since, as we'll see later, patterns like `vector()` behave differently for `data` types than normal classes. In this case, `vector()` would only match a *zero-length* vector, not just any vector.

The other new construct used here is the `|*>`, or star-pipe, operator, which functions exactly like the normal pipe, except that instead of calling the function with one argument, it calls it with as many arguments as there are elements in the sequence passed into it. The difference between `|>` and `|*>` is exactly analogous to the difference between `f(args)` and `f(*args)`.

2.5.3 n-Vector Methods

Now that we have a constructor for our n-vector, it's time to write its methods. First up is `__abs__`, which should compute the vector's magnitude. This will be slightly more complicated than with the 2-vector, since we have to make it work over an arbitrary number of `pts`. Fortunately, we can use Coconut's pipeline-style programming and partial application to make it simple:

```
    def __abs__(self) =
        """Return the magnitude of the vector."""
        self.pts |> map$(.**2) |> sum |> (.**0.5)
```

The basic algorithm here is map square over each element, sum them all, then square root the result. The one new construct here is the `(.**2)` and `(.**0.5)` syntax, which are effectively equivalent to `(x => x**2)` and `(x => x**0.5)`, respectively (though the `(.**2)` syntax produces a pickleable object). This syntax works for all *operator functions*, so you can do things like `(1-.)` or `(cond() or .)`.

Next up is vector addition. The goal here is to add two vectors of equal length by adding their components. To do this, we're going to make use of Coconut's ability to perform pattern-matching, or in this case destructuring assignment, to data types, like so:

```
    def __add__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Add two vectors together."""
        map((+), self.pts, other_pts) |*> vector
```

There are a couple of new constructs here, but the main notable one is the pattern-matching `vector(*other_pts)` which showcases the syntax for pattern-matching against data types: it mimics exactly the original `data` declaration of that data type. In this case, `vector(*other_pts)` will only match a vector, raising a `MatchError` otherwise, and if it does match a vector, will assign the vector's `pts` attribute to the variable `other_pts`.

Next is vector subtraction, which is just like vector addition, but with `(-)` instead of `(+)`:

```
    def __sub__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
```

(continues on next page)

(continued from previous page)

```
        """Subtract one vector from another."""
        map((-), self.pts, other_pts) |*> vector
```

One thing to note here is that unlike the other operator functions, `(-)` can either mean negation or subtraction, the meaning of which will be inferred based on how many arguments are passed, 1 for negation, 2 for subtraction. To show this, we'll use the same `(-)` function to implement vector negation, which should simply negate each element:

```
    def __neg__(self) =
        """Retrieve the negative of the vector."""
        self.pts |> map$(-) |*> vector
```

The last method we'll implement is multiplication. This one is a little bit tricky, since mathematically, there are a whole bunch of different ways to multiply vectors. For our purposes, we're just going to look at two: between two vectors of equal length, we want to compute the dot product, defined as the sum of the corresponding elements multiplied together, and between a vector and a scalar, we want to compute the scalar multiple, which is just each element multiplied by that scalar. Here's our implementation:

```
    def __mul__(self, other):
        """Scalar multiplication and dot product."""
        match vector(*other_pts) in other:
            assert len(other_pts) == len(self.pts)
            return map((*), self.pts, other_pts) |> sum  # dot product
        else:
            return self.pts |> map$(.*other) |*> vector  # scalar multiple
    def __rmul__(self, other) =
        """Necessary to make scalar multiplication commutative."""
        self * other
```

The first thing to note here is that unlike with addition and subtraction, where we wanted to raise an error if the vector match failed, here, we want to do scalar multiplication if the match fails, so instead of using destructuring assignment, we use a `match` statement. The second thing to note here is the combination of pipeline-style programming, partial application, operator functions, and higher-order functions we're using to compute the dot product and scalar multiple. For the dot product, we map multiplication over the two vectors, then sum the result. For the scalar multiple, we take the original points, map multiplication by the scalar over them, then use them to make a new vector.

Finally, putting everything together:

```
data vector(*pts):
    """Immutable n-vector."""
    def __new__(cls, *pts):
        """Create a new vector from the given pts."""
        match [v `isinstance` vector] in pts:
            return v  # vector(v) where v is a vector should return v
        else:
            return pts |*> makedata$(cls)  # accesses base constructor
    def __abs__(self) =
        """Return the magnitude of the vector."""
        self.pts |> map$(.**2) |> sum |> (.**0.5)
    def __add__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Add two vectors together."""
        map((+), self.pts, other_pts) |*> vector
    def __sub__(self, vector(*other_pts)
```

(continues on next page)

(continued from previous page)

```
                if len(other_pts) == len(self.pts)) =
        """Subtract one vector from another."""
        map((-), self.pts, other_pts) |*> vector
    def __neg__(self) =
        """Retrieve the negative of the vector."""
        self.pts |> map$(-) |*> vector
    def __mul__(self, other):
        """Scalar multiplication and dot product."""
        match vector(*other_pts) in other:
            assert len(other_pts) == len(self.pts)
            return map((*), self.pts, other_pts) |> sum  # dot product
        else:
            return self.pts |> map$(.*other) |*> vector  # scalar multiplication
    def __rmul__(self, other) =
        """Necessary to make scalar multiplication commutative."""
        self * other

# Test cases:
vector(1, 2, 3) |> print  # vector(*pts=(1, 2, 3))
vector(4, 5) |> vector |> print  # vector(*pts=(4, 5))
vector(3, 4) |> abs |> print  # 5
vector(1, 2) + vector(2, 3) |> print  # vector(*pts=(3, 5))
vector(2, 2) - vector(0, 1) |> print  # vector(*pts=(2, 1))
-vector(1, 3) |> print  # vector(*pts=(-1, -3))
(vector(1, 2) == "string") |> print  # False
(vector(1, 2) == vector(3, 4)) |> print  # False
(vector(2, 4) == vector(2, 4)) |> print  # True
2*vector(1, 2) |> print  # vector(*pts=(2, 4))
vector(1, 2) * vector(1, 3) |> print  # 7
```

Copy, paste! Now that was a lot of code. But looking it over, it looks clean, readable, and concise, and it does precisely what we intended it to do: create an algebraic data type for an immutable n-vector that supports the basic vector operations. And we did the whole thing without needing any imperative constructs like state or loops—pure functional programming.

2.6 Case Study 4: `vector_field`

For the final case study, instead of me writing the code, and you looking at it, you'll be writing the code—of course, I won't be looking at it, but I will show you how I would have done it after you give it a shot by yourself.

The bonus challenge for this section is to write each of the functions we'll be defining in just one line. Try using assignment functions to help with that!

First, let's introduce the general goal of this case study. We want to write a program that will allow us to produce infinite vector fields that we can iterate over and apply operations to. And in our case, we'll say we only care about vectors with positive components.

Our first step, therefore, is going to be creating a field of all the points with positive `x` and `y` values—that is, the first quadrant of the `x-y` plane, which looks something like this:

```
...
```

(continues on next page)

(continued from previous page)

```
(0,2)   ...

(0,1)   (1,1)   ...

(0,0)   (1,0)   (2,0)   ...
```

But since we want to be able to iterate over that plane, we're going to need to linearize it somehow, and the easiest way to do that is to split it up into diagonals, and traverse the first diagonal, then the second diagonal, and so on, like this:

```
(0, 0), (1, 0), (0, 1), (2, 0), (1, 1), (0, 2), ...
```

2.6.1 diagonal_line

Thus, our first function `diagonal_line(n)` should construct an iterator of all the points, represented as coordinate tuples, in the `n`th diagonal, starting with `(0, 0)` as the `0`th diagonal. Like we said at the start of this case study, this is where we I let go and you take over. Using all the tools of functional programming that Coconut provides, give `diagonal_line` a shot. When you're ready to move on, scroll down.

Here are some tests that you can use:

```
diagonal_line(0) `isinstance` (list, tuple) |> print  # False (should be an iterator)
diagonal_line(0) |> list |> print  # [(0, 0)]
diagonal_line(1) |> list |> print  # [(0, 1), (1, 0)]
```

Hint: the `n`*th diagonal should contain* `n+1` *elements, so try starting with* `range(n+1)` *and then transforming it in some way.*

That wasn't so bad, now was it? Now, let's take a look at my solution:

```
def diagonal_line(n) = range(n+1) |> map$(i => (i, n-i))
```

Pretty simple, huh? We take `range(n+1)`, and use `map` to transform it into the right sequence of tuples.

2.6.2 linearized_plane

Now that we've created our diagonal lines, we need to join them together to make the full linearized plane, and to do that we're going to write the function `linearized_plane()`. `linearized_plane` should produce an iterator that goes through all the points in the plane, in order of all the points in the first `diagonal(0)`, then the second `diagonal(1)`, and so on. `linearized_plane` is going to be, by necessity, an infinite iterator, since it needs to loop through all the points in the plane, which have no end. To help you accomplish this, remember that the `::` operator is lazy, and won't evaluate its operands until they're needed, which means it can be used to construct infinite iterators. When you're ready to move on, scroll down.

Tests:

```
# Note: these tests use $[] notation, which we haven't introduced yet
#  but will introduce later in this case study; for now, just run the
#  tests, and make sure you get the same result as is in the comment
linearized_plane()$[0] |> print  # (0, 0)
linearized_plane()$[:3] |> list |> print  # [(0, 0), (0, 1), (1, 0)]
```

Hint: instead of defining the function as `linearized_plane()`*, try defining it as* `linearized_plane(n=0)`*, where* `n` *is the diagonal to start at, and use recursion to build up from there.*

That was a little bit rougher than the first one, but hopefully still not too bad. Let's compare to my solution:

```
def linearized_plane(n=0) = diagonal_line(n) :: linearized_plane(n+1)
```

As you can see, it's a very fundamentally simple solution: just use `::` and recursion to join all the diagonals together in order.

2.6.3 `vector_field`

Now that we have a function that builds up all the points we need, it's time to turn them into vectors, and to do that we'll define the new function `vector_field()`, which should turn all the tuples in `linearized_plane` into vectors, using the n-vector class we defined earlier.

Tests:

```
# You'll need to bring in the vector class from earlier to make these work
vector_field()$[0] |> print  # vector(*pts=(0, 0))
vector_field()$[2:3] |> list |> print  # [vector(*pts=(1, 0))]
```

Hint: Remember, the way we defined vector it takes the components as separate arguments, not a single tuple. You may find the Coconut built-in `starmap` useful in dealing with that.

We're making good progress! Before we move on, check your solution against mine:

```
def vector_field() = linearized_plane() |> starmap$(vector)
```

All we're doing is taking our `linearized_plane` and mapping `vector` over it, but using `starmap` instead of `map` so that `vector` gets called with each element of the tuple as a separate argument.

2.6.4 Applications

Now that we've built all the functions we need for our vector field, it's time to put it all together and test it. Feel free to substitute in your versions of the functions below:

```
data vector(*pts):
    """Immutable n-vector."""
    def __new__(cls, *pts):
        """Create a new vector from the given pts."""
        match [v `isinstance` vector] in pts:
            return v  # vector(v) where v is a vector should return v
        else:
            return pts |*> makedata$(cls)  # accesses base constructor
    def __abs__(self) =
        """Return the magnitude of the vector."""
        self.pts |> map$(.**2) |> sum |> (.**0.5)
    def __add__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Add two vectors together."""
        map((+), self.pts, other_pts) |*> vector
    def __sub__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Subtract one vector from another."""
        map((-), self.pts, other_pts) |*> vector
```

(continues on next page)

(continued from previous page)

```
    def __neg__(self) =
        """Retrieve the negative of the vector."""
        self.pts |> map$(-) |*> vector
    def __mul__(self, other):
        """Scalar multiplication and dot product."""
        match vector(*other_pts) in other:
            assert len(other_pts) == len(self.pts)
            return map((*), self.pts, other_pts) |> sum  # dot product
        else:
            return self.pts |> map$(.*other) |*> vector  # scalar multiplication
    def __rmul__(self, other) =
        """Necessary to make scalar multiplication commutative."""
        self * other

def diagonal_line(n) = range(n+1) |> map$(i => (i, n-i))
def linearized_plane(n=0) = diagonal_line(n) :: linearized_plane(n+1)
def vector_field() = linearized_plane() |> starmap$(vector)

# Test cases:
diagonal_line(0) `isinstance` (list, tuple) |> print  # False (should be an iterator)
diagonal_line(0) |> list |> print  # [(0, 0)]
diagonal_line(1) |> list |> print  # [(0, 1), (1, 0)]
linearized_plane()$[0] |> print  # (0, 0)
linearized_plane()$[:3] |> list |> print  # [(0, 0), (0, 1), (1, 0)]
vector_field()$[0] |> print  # vector(*pts=(0, 0))
vector_field()$[2:3] |> list |> print  # [vector(*pts=(1, 0))]
```

Copy, paste! Once you've made sure everything is working correctly if you substituted in your own functions, take a look at the last 4 tests. You'll notice that they use a new notation, similar to the notation for partial application we saw earlier, but with brackets instead of parentheses. This is the notation for iterator slicing. Similar to how partial application was lazy function calling, iterator slicing is *lazy sequence slicing*. Like with partial application, it is helpful to think of `$` as the *lazy-ify* operator, in this case turning normal Python slicing, which is evaluated immediately, into lazy iterator slicing, which is evaluated only when the elements in the slice are needed.

With that in mind, now that we've built our vector field, it's time to use iterator slicing to play around with it. Try doing something cool to our vector fields like

- create a `magnitude_field` where each point is that vector's magnitude
- combine entire vector fields together with `map` and the vector addition and multiplication methods we wrote earlier

then use iterator slicing to take out portions and examine them.

2.7 Case Study 5: `vector` Part II

For the some of the applications you might want to use your `vector_field` for, it might be desirable to add some useful methods to our `vector`. In this case study, we're going to be focusing on one in particular: `.angle`.

`.angle` will take one argument, another vector, and compute the angle between the two vectors. Mathematically, the formula for the angle between two vectors is the dot product of the vectors' respective unit vectors. Thus, before we can implement `.angle`, we're going to need `.unit`. Mathematically, the formula for the unit vector of a given vector is that vector divided by its magnitude. Thus, before we can implement `.unit`, and by extension `.angle`, we'll need to start by implementing division.

2.7.1 `__truediv__`

Vector division is just scalar division, so we're going to write a `__truediv__` method that takes `self` as the first argument and `other` as the second argument, and returns a new vector the same size as `self` with every element divided by `other`. For an extra challenge, try writing this one in one line using assignment function notation.

Tests:

```
vector(3, 4) / 1 |> print  # vector(*pts=(3.0, 4.0))
vector(2, 4) / 2 |> print  # vector(*pts=(1.0, 2.0))
```

Hint: Look back at how we implemented scalar multiplication.

Here's my solution for you to check against:

```
    def __truediv__(self, other) = self.pts |> map$(x => x/other) |*> vector
```

2.7.2 `.unit`

Next up, `.unit`. We're going to write a `unit` method that takes just `self` as its argument and returns a new vector the same size as `self` with each element divided by the magnitude of `self`, which we can retrieve with `abs`. This should be a very simple one-line function.

Tests:

```
vector(0, 1).unit() |> print  # vector(*pts=(0.0, 1.0))
vector(5, 0).unit() |> print  # vector(*pts=(1.0, 0.0))
```

Here's my solution:

```
    def unit(self) = self / abs(self)
```

2.7.3 `.angle`

This one is going to be a little bit more complicated. For starters, the mathematical formula for the angle between two vectors is the `math.acos` of the dot product of those vectors' respective unit vectors, and recall that we already implemented the dot product of two vectors when we wrote `__mul__`. So, `.angle` should take `self` as the first argument and `other` as the second argument, and if `other` is a vector, use that formula to compute the angle between `self` and `other`, or if `other` is not a vector, `.angle` should raise a `MatchError`. To accomplish this, we're going to want to use destructuring assignment to check that `other` is indeed a `vector`.

Tests:

```
import math
vector(2, 0).angle(vector(3, 0)) |> print  # 0.0
print(vector(1, 0).angle(vector(0, 2)), math.pi/2)  # should be the same
vector(1, 2).angle(5)  # MatchError
```

Hint: Look back at how we checked whether the argument to `factorial` *was an integer using pattern-matching.*

Here's my solution—take a look:

```
    def angle(self, other `isinstance` vector) = math.acos(self.unit() * other.unit())
```

And now it's time to put it all together. Feel free to substitute in your own versions of the methods we just defined.

```
import math  # necessary for math.acos in .angle

data vector(*pts):
    """Immutable n-vector."""
    def __new__(cls, *pts):
        """Create a new vector from the given pts."""
        match [v `isinstance` vector] in pts:
            return v  # vector(v) where v is a vector should return v
        else:
            return pts |*> makedata$(cls)  # accesses base constructor
    def __abs__(self) =
        """Return the magnitude of the vector."""
        self.pts |> map$(.**2) |> sum |> (.**0.5)
    def __add__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Add two vectors together."""
        map((+), self.pts, other_pts) |*> vector
    def __sub__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Subtract one vector from another."""
        map((-), self.pts, other_pts) |*> vector
    def __neg__(self) =
        """Retrieve the negative of the vector."""
        self.pts |> map$(-) |*> vector
    def __mul__(self, other):
        """Scalar multiplication and dot product."""
        match vector(*other_pts) in other:
            assert len(other_pts) == len(self.pts)
            return map((*), self.pts, other_pts) |> sum  # dot product
        else:
            return self.pts |> map$(.*other) |*> vector  # scalar multiplication
    def __rmul__(self, other) =
        """Necessary to make scalar multiplication commutative."""
        self * other
    # New one-line functions necessary for finding the angle between vectors:
    def __truediv__(self, other) = self.pts |> map$(x => x/other) |*> vector
    def unit(self) = self / abs(self)
    def angle(self, other `isinstance` vector) = math.acos(self.unit() * other.unit())

# Test cases:
vector(3, 4) / 1 |> print  # vector(*pts=(3.0, 4.0))
```

(continues on next page)

(continued from previous page)

```
vector(2, 4) / 2 |> print  # vector(*pts=(1.0, 2.0))
vector(0, 1).unit() |> print  # vector(*pts=(0.0, 1.0))
vector(5, 0).unit() |> print  # vector(*pts=(1.0, 0.0))
vector(2, 0).angle(vector(3, 0)) |> print  # 0.0
print(vector(1, 0).angle(vector(0, 2)), math.pi/2)  # should be the same
vector(1, 2).angle(5)  # MatchError
```

One note of warning here: be careful not to leave a blank line when substituting in your methods, or the interpreter will cut off the code for the `vector` *there. This isn't a problem in normal Coconut code, only here because we're copy-and-pasting into the command line.*

Copy, paste! If everything is working, you can try going back to playing around with `vector_field` *applications* using our new methods.

2.8 Filling in the Gaps

And with that, this tutorial is out of case studies—but that doesn't mean Coconut is out of features! In this last section, we'll touch on some of the other useful features of Coconut that we managed to miss in the case studies.

2.8.1 Lazy Lists

First up is lazy lists. Lazy lists are lazily-evaluated lists, similar in their laziness to Coconut's `::` operator, in that any expressions put inside a lazy list won't be evaluated until that element of the lazy list is needed. The syntax for lazy lists is exactly the same as the syntax for normal lists, but with "banana brackets" (`(|` and `|)`) instead of normal brackets, like so:

```
abc = (| a, b, c |)
```

Unlike Python iterators, lazy lists can be iterated over multiple times and still return the same result.

Unlike Python lists, however, using a lazy list, it is possible to define the values used in the following expressions as needed without raising a `NameError`:

```
abcd = (| d(a), d(b), d(c) |)  # a, b, c, and d are not defined yet
def d(n) = n + 1

a = 1
abcd$[0]
b = 2
abcd$[1]
c = 3
abcd$[2]
```

2.8.2 Function Composition

Next is function composition. In Coconut, this is primarily accomplished through the `f1 ..> f2` operator, which takes two functions and composes them, creating a new function equivalent to `(*args, **kwargs) => f2(f1(*args, **kwargs))`. This can be useful in combination with partial application for piecing together multiple higher-order functions, like so:

```
zipsum = zip ..> map$(sum)
```

While `..>` is generally preferred, if you'd rather use the more traditional mathematical function composition ordering, you can get that with the `<..` operator.

If the composed functions are wrapped in parentheses, arguments can be passed into them:

```
def plus1(x) = x + 1
def square(x) = x * x

(square ..> plus1)(3) == 10  # True
```

Functions of different arities can be composed together, as long as they are in the correct order. If they are in the incorrect order, a `TypeError` will be raised. In this example we will compose a unary function with a binary function:

```
def add(n, m) = n + m  # binary function
def square(n) = n * n  # unary function

(square ..> add)(3, 1)    # Raises TypeError: square() takes exactly 1 argument (2 given)
(add ..> square)(3, 1)    # 16
```

Another useful trick with function composition involves composing a function with a higher-order function:

```
def inc_or_dec(t):
    # Our higher-order function, which returns another function
    if t:
        return x => x+1
    else:
        return x => x-1

def square(n) = n * n

square_inc = inc_or_dec(True) ..> square
square_dec = inc_or_dec(False) ..> square
square_inc(4)  # 25
square_dec(4)  # 9
```

Note: Coconut also supports the function composition operators `..`, `..*>`, `<*..`, `..**>`, and `<**..`.

2.8.3 Implicit Partials

Another useful Coconut feature is implicit partials. Coconut supports a number of different "incomplete" expressions that will evaluate to a function that takes in the part necessary to complete them, that is, an implicit partial application function. The different allowable expressions are:

```
.attr
.method(args)
func$
seq[]
iter$[]
.[slice]
.$[slice]
```

For a full explanation of what each implicit partial does, see Coconut's documentation on *implicit partials*.

2.8.4 Type Annotations

For many people, one of the big downsides of Python is the fact that it is dynamically-typed. In Python, this problem is addressed by MyPy, a static type analyzer for Python, which can check Python-3-style type annotations such as

```
def plus1(x: int) -> int:
    return x + 1
a: int = plus1(10)
```

Unfortunately, in Python, such type annotation syntax only exists in Python 3. Not to worry in Coconut, however, which compiles Python-3-style type annotations to universally compatible type comments. Not only that, but Coconut has built-in *MyPy integration* for automatically type-checking your code, and its own *enhanced type annotation syntax* for more easily expressing complex types, like so:

```
def int_map(
    f: int -> int,
    xs: int[],
) -> int[] =
    xs |> map$(f) |> list
```

2.8.5 Further Reading

And that's it for this tutorial! But that's hardly it for Coconut. All of the features examined in this tutorial, as well as a bunch of others, are detailed in Coconut's *documentation*.

Also, if you have any other questions not covered in this tutorial, feel free to ask around at Coconut's Gitter, a GitHub-integrated chat room for Coconut developers.

Finally, Coconut is a new, growing language, and if you'd like to get involved in the development of Coconut, all the code is available completely open-source on Coconut's GitHub. Contributing is a simple as forking the code, making your changes, and proposing a pull request! See Coconuts *contributing guidelines* for more information.

CHAPTER

THREE

COCONUT DOCUMENTATION

3.1 Overview

This documentation covers all the features of the Coconut Programming Language, and is intended as a reference/specification, not a tutorialized introduction. For an introduction to and tutorial of Coconut, see *the tutorial*.

Coconut is a variant of Python built for **simple, elegant, Pythonic functional programming**. Coconut syntax is a strict superset of the latest Python 3 syntax. Thus, users familiar with Python will already be familiar with most of Coconut.

The Coconut compiler turns Coconut code into Python code. The primary method of accessing the Coconut compiler is through the Coconut command-line utility, which also features an interpreter for real-time compilation. In addition to the command-line utility, Coconut also supports the use of IPython/Jupyter notebooks.

Thought Coconut syntax is primarily based on that of Python, other languages that inspired Coconut include Haskell, CoffeeScript, F#, and Julia.

3.1.1 Try It Out

If you want to try Coconut in your browser, check out the online interpreter. Note, however, that it may be running an outdated version of Coconut.

3.2 Installation

3.2.1 Using Pip

Since Coconut is hosted on the Python Package Index, it can be installed easily using `pip`. Simply install Python, open up a command-line prompt, and enter

```
pip install coconut
```

which will install Coconut and its required dependencies.

Note: If you have an old version of Coconut installed and you want to upgrade, run `pip install --upgrade coconut` *instead.*

If you are encountering errors running `pip install coconut`, try adding `--user` or running

```
pip install --no-deps --upgrade coconut "pyparsing<3"
```

which will force Coconut to use the pure-Python `pyparsing` module instead of the faster `cPyparsing` module. If you are still getting errors, you may want to try *using conda* instead.

If `pip install coconut` works, but you cannot access the `coconut` command, be sure that Coconut's installation location is in your `PATH` environment variable. On UNIX, that is `/usr/local/bin` (without `--user`) or `${HOME}/.local/bin/` (with `--user`).

3.2.2 Using Conda

If you prefer to use conda instead of `pip` to manage your Python packages, you can also install Coconut using `conda`. Just install conda, open up a command-line prompt, and enter

```
conda config --add channels conda-forge
conda install coconut
```

which will properly create and build a `conda` recipe out of Coconut's `conda-forge` feedstock.

Note: Coconut's `conda` recipe uses `pyparsing` rather than `cPyparsing`, which may lead to degraded performance relative to installing Coconut via `pip`.

3.2.3 Using Homebrew

If you prefer to use Homebrew, you can also install Coconut using `brew`:

```
brew install coconut
```

Note: Coconut's Homebrew formula may not always be up-to-date with the latest version of Coconut.

3.2.4 Optional Dependencies

Coconut also has optional dependencies, which can be installed by entering

```
pip install coconut[name_of_optional_dependency]
```

or, to install multiple optional dependencies,

```
pip install coconut[opt_dep_1,opt_dep_2]
```

The full list of optional dependencies is:

- `all`: alias for everything below (this is the recommended way to install a feature-complete version of Coconut).
- `jupyter/ipython`: enables use of the `--jupyter` / `--ipython` flag.
- `kernel`: lightweight subset of `jupyter` that only includes the dependencies that are strictly necessary for Coconut's *Jupyter kernel*.
- `watch`: enables use of the `--watch` flag.
- `mypy`: enables use of the `--mypy` flag.
- `xonsh`: enables use of Coconut's *`xonsh` support*.
- `numpy`: installs everything necessary for making use of Coconut's *`numpy` integration*.
- `jupyterlab`: installs everything necessary to use JupyterLab with Coconut.
- `jupytext`: installs everything necessary to use Jupytext with Coconut.

3.2.5 Develop Version

Alternatively, if you want to test out Coconut's latest and greatest, enter

```
pip install coconut-develop
```

which will install the most recent working version from Coconut's `develop` branch. Optional dependency installation is supported in the same manner as above. For more information on the current development build, check out the development version of this documentation. Be warned: `coconut-develop` is likely to be unstable—if you find a bug, please report it by creating a new issue.

Note: if you have an existing release version of `coconut` *installed, you'll need to* `pip uninstall coconut` *before installing* `coconut-develop`.

3.3 Compilation

3.3.1 Usage

```
coconut [-h] [--and source [dest ...]] [-v] [-t version] [-i] [-p] [-a] [-l]
        [--no-line-numbers] [-k] [-w] [-r] [-n] [-d] [-q] [-s] [--no-tco]
        [--no-wrap-types] [-c code] [--incremental] [-j processes] [-f] [--minify]
        [--jupyter ...] [--mypy ...] [--argv ...] [--tutorial] [--docs] [--style name]
        [--vi-mode] [--recursion-limit limit] [--stack-size kbs] [--site-install]
        [--site-uninstall] [--verbose] [--trace] [--profile]
        [source] [dest]
```

Positional Arguments

```
source                path to the Coconut file/folder to compile
dest                  destination directory for compiled files (defaults to
                      the source directory)
```

Optional Arguments

```
-h, --help            show this help message and exit
--and source [dest ...]
                      add an additional source/dest pair to compile (dest is optional)
-v, -V, --version     print Coconut and Python version information
-t version, --target version
                      specify target Python version (defaults to universal)
-i, --interact        force the interpreter to start (otherwise starts if no other␣
↪command is
                      given) (implies --run)
-p, --package         compile source as part of a package (defaults to only if source is␣
↪a
                      directory)
-a, --standalone, --stand-alone
                      compile source as standalone files (defaults to only if source is␣
↪a single
                      file)
-l, --line-numbers, --linenumbers
                      force enable line number comments (--line-numbers are enabled by␣
↪default
                      unless --minify is passed)
--no-line-numbers, --nolinenumbers
                      disable line number comments (opposite of --line-numbers)
-k, --keep-lines, --keeplines
                      include source code in comments for ease of debugging
-w, --watch           watch a directory and recompile on changes
-r, --run             execute compiled Python
-n, --no-write, --nowrite
                      disable writing compiled Python
-d, --display         print compiled Python
-q, --quiet           suppress all informational output (combine with --display to write
                      runnable code to stdout)
-s, --strict          enforce code cleanliness standards
--no-tco, --notco     disable tail call optimization
--no-wrap-types, --nowraptypes
                      disable wrapping type annotations in strings and turn off 'from __
↪future__
                      import annotations' behavior
-c code, --code code  run Coconut passed in as a string (can also be piped into stdin)
-j processes, --jobs processes
                      number of additional processes to use (defaults to 'sys') (0 is no
                      additional processes; 'sys' uses machine default)
-f, --force           force re-compilation even when source code and compilation␣
↪parameters
                      haven't changed
--minify              reduce size of compiled Python
--jupyter ..., --ipython ...
                      run Jupyter/IPython with Coconut as the kernel (remaining args␣
↪passed to
                      Jupyter)
--mypy ...            run MyPy on compiled Python (remaining args passed to MyPy)␣
↪(implies
```

(continues on next page)

(continued from previous page)

```
                      --package --line-numbers)
--argv ..., --args ...
                      set sys.argv to source plus remaining args for use in the Coconut␣
↪script
                      being run
--tutorial            open Coconut's tutorial in the default web browser
--docs, --documentation
                      open Coconut's documentation in the default web browser
--style name          set Pygments syntax highlighting style (or 'list' to list styles)
                      (defaults to COCONUT_STYLE environment variable if it exists,␣
↪otherwise
                      'default')
--vi-mode, --vimode   enable vi mode in the interpreter (currently set to False) (can be
                      modified by setting COCONUT_VI_MODE environment variable)
--recursion-limit limit, --recursionlimit limit
                      set maximum recursion depth in compiler (defaults to 1920) (when
                      increasing --recursion-limit, you may also need to increase --
↪stack-size;
                      setting them to approximately equal values is recommended)
--stack-size kbs, --stacksize kbs
                      run the compiler in a separate thread with the given stack size in
                      kilobytes
--fail-fast           causes the compiler to fail immediately upon encountering a␣
↪compilation
                      error rather than attempting to continue compiling other files
--no-cache            disables use of Coconut's incremental parsing cache (caches␣
↪previous
                      parses to improve recompilation performance for slightly modified␣
↪files)
--site-install, --siteinstall
                      set up coconut.api to be imported on Python start
--site-uninstall, --siteuninstall
                      revert the effects of --site-install
--verbose             print verbose debug output
--trace               print verbose parsing data (only available in coconut-develop)
--profile             collect and print timing info (only available in coconut-develop)
```

3.3.2 Coconut Scripts

To run a Coconut file as a script, Coconut provides the command

```
coconut-run <source> <args>
```

as an alias for

```
coconut --quiet --target sys --keep-lines --run <source> --argv <args>
```

which will quietly compile and run `<source>`, passing any additional arguments to the script, mimicking how the `python` command works. To instead pass additional compilation arguments to Coconut itself (e.g. `--no-tco`), put them before the `<source>` file.

`coconut-run` can be used to compile and run directories rather than files, again mimicking how the `python` command works. Specifically, Coconut will compile the directory and then run the `__main__.coco` in that directory, which must exist.

`coconut-run` can be used in a Unix shebang line to create a Coconut script by adding the following line to the start of your script:

```
#!/usr/bin/env coconut-run
```

`coconut-run` will always enable *automatic compilation*, such that Coconut source files can be directly imported from any Coconut files run via `coconut-run`. Additionally, compilation parameters (e.g. `--no-tco`) used in `coconut-run` will be passed along and used for any auto compilation.

On Python 3.4+, `coconut-run` will use a `__coconut_cache__` directory to cache the compiled Python. Note that `__coconut_cache__` will always be removed from `__file__`.

3.3.3 Naming Source Files

Coconut source files should, so the compiler can recognize them, use the extension `.coco`.

When Coconut compiles a `.coco` file, it will compile to another file with the same name, except with `.py` instead of `.coco`, which will hold the compiled code.

If an extension other than `.py` is desired for the compiled files, then that extension can be put before `.coco` in the source file name, and it will be used instead of `.py` for the compiled files. For example, `name.coco` will compile to `name.py`, whereas `name.abc.coco` will compile to `name.abc`.

3.3.4 Compilation Modes

Files compiled by the `coconut` command-line utility will vary based on compilation parameters. If an entire directory of files is compiled (which the compiler will search recursively for any folders containing `.coco` files), a `__coconut__.py` file will be created to house necessary functions (package mode), whereas if only a single file is compiled, that information will be stored within a header inside the file (standalone mode). Standalone mode is better for single files because it gets rid of the overhead involved in importing `__coconut__.py`, but package mode is better for large packages because it gets rid of the need to run the same Coconut header code again in every file, since it can just be imported from `__coconut__.py`.

By default, if the `source` argument to the command-line utility is a file, it will perform standalone compilation on it, whereas if it is a directory, it will recursively search for all `.coco` files and perform package compilation on them. Thus, in most cases, the mode chosen by Coconut automatically will be the right one. But if it is very important that no additional files like `__coconut__.py` be created, for example, then the command-line utility can also be forced to use a specific mode with the `--package` (`-p`) and `--standalone` (`-a`) flags.

3.3.5 Compatible Python Versions

While Coconut syntax is based off of the latest Python 3, Coconut code compiled in universal mode (the default `--target`)—and the Coconut compiler itself—should run on any Python version `>= 2.6` on the `2.x` branch or `>= 3.2` on the `3.x` branch (and on either CPython or PyPy).

To make Coconut built-ins universal across Python versions, Coconut makes available on any Python version built-ins that only exist in later versions, including **automatically overwriting Python 2 built-ins with their Python 3 counterparts.** Additionally, Coconut also *overwrites some Python 3 built-ins for optimization and enhancement purposes*. If access to the original Python versions of any overwritten built-ins is desired, the old built-ins can be retrieved by prefixing them with `py_`. Specifically, the overwritten built-ins are:

- `py_bytes`

- `py_chr`
- `py_dict`
- `py_hex`
- `py_input`
- `py_int`
- `py_map`
- `py_object`
- `py_oct`
- `py_open`
- `py_print`
- `py_range`
- `py_str`
- `py_super`
- `py_zip`
- `py_filter`
- `py_reversed`
- `py_enumerate`
- `py_raw_input`
- `py_xrange`
- `py_repr`
- `py_breakpoint`
- `py_min`
- `py_max`

Note: Coconut's `repr` can be somewhat tricky, as it will attempt to remove the `u` before reprs of unicode strings on Python 2, but will not always be able to do so if the unicode string is nested.

For standard library compatibility, **Coconut automatically maps imports under Python 3 names to imports under Python 2 names**. Thus, Coconut will automatically take care of any standard library modules that were renamed from Python 2 to Python 3 if just the Python 3 name is used. For modules or packages that only exist in Python 3, however, Coconut has no way of maintaining compatibility.

Finally, while Coconut will try to compile Python-3-specific syntax to its universal equivalent, the following constructs have no equivalent in Python 2, and require the specification of a target of at least `3` to be used:

- the `nonlocal` keyword,
- keyword-only function parameters (use *pattern-matching function definition* for universal code),
- `async` and `await` statements (requires a specific target; Coconut will attempt different *backports* based on the targeted version),
- `:=` assignment expressions (requires `--target 3.8`),
- positional-only function parameters (use *pattern-matching function definition* for universal code) (requires `--target 3.8`),

- `a[x, *y]` variadic generic syntax (use *type parameter syntax* for universal code) (requires `--target 3.11`), and
- `except*` multi-except statements (requires `--target 3.11`).

Note: Coconut also universalizes many magic methods, including making `__bool__` and `__set_name__` work on any Python version.

3.3.6 Allowable Targets

If the version of Python that the compiled code will be running on is known ahead of time, a target should be specified with `--target`. The given target will only affect the compiled code and whether or not the Python-3-specific syntax detailed above is allowed. Where Python syntax differs across versions, Coconut syntax will always follow the latest Python 3 across all targets. The supported targets are:

- `universal`, `univ` (the default): will work on *any* of the below
- `2`, `2.6`: will work on any Python `>= 2.6` but `< 3`
- `2.7`: will work on any Python `>= 2.7` but `< 3`
- `3`, `3.2`: will work on any Python `>= 3.2`
- `3.3`: will work on any Python `>= 3.3`
- `3.4`: will work on any Python `>= 3.4`
- `3.5`: will work on any Python `>= 3.5`
- `3.6`: will work on any Python `>= 3.6`
- `3.7`: will work on any Python `>= 3.7`
- `3.8`: will work on any Python `>= 3.8`
- `3.9`: will work on any Python `>= 3.9`
- `3.10`: will work on any Python `>= 3.10`
- `3.11`: will work on any Python `>= 3.11`
- `3.12`: will work on any Python `>= 3.12`
- `3.13`: will work on any Python `>= 3.13`
- `sys`: chooses the target corresponding to the current Python version
- `psf`: will work on any Python not considered end-of-life by the PSF (Python Software Foundation)

Note: Periods are optional in target specifications, such that the target `27` is equivalent to the target `2.7`.

3.3.7 `strict` Mode

If the `--strict` (`-s` for short) flag is enabled, Coconut will perform additional checks on the code being compiled. It is recommended that you use the `--strict` flag if you are starting a new Coconut project, as it will help you write cleaner code. Specifically, the extra checks done by `--strict` are:

- disabling deprecated features (making them entirely unavailable to code compiled with `--strict`),
- errors instead of warnings on unused imports (unless they have a `# NOQA` or `# noqa` comment),
- errors instead of warnings when overwriting built-ins (unless a backslash is used to escape the built-in name),
- warning on missing `__init__.coco` files when compiling in `--package` mode,

- throwing errors on various style problems (see list below).

The style issues which will cause `--strict` to throw an error are:

- mixing of tabs and spaces
- use of `"hello" "world"` implicit string concatenation (use explicit + instead)
- use of `from __future__` imports (Coconut does these automatically)
- inheriting from `object` in classes (Coconut does this automatically)
- semicolons at end of lines
- use of `u` to denote Unicode strings (all Coconut strings are Unicode strings)
- `f`-strings with no format expressions in them
- commas after *statement lambdas* (not recommended as it can be unclear whether the comma is inside or outside the lambda)
- missing new line at end of file
- trailing whitespace at end of lines
- use of the Python-style `lambda` statement (use *Coconut's lambda syntax* instead)
- use of backslash continuation (use *parenthetical continuation* instead)
- Python-3.10/PEP-634-style dotted names in pattern-matching (Coconut style is to preface these with ==)
- use of `:` instead of `<:` to specify upper bounds in *Coconut's type parameter syntax*

Note that many of the above style issues will still show a warning if `--strict` is not present.

3.3.8 Backports

In addition to the newer Python features that Coconut can backport automatically itself to older Python versions, Coconut will also automatically compile code to make use of a variety of external backports as well. These backports are automatically installed with Coconut if needed and Coconut will automatically use them instead of the standard library if the standard library is not available. These backports are:

- `typing` for backporting `typing`.
- `typing_extensions` for backporting individual `typing` objects.
- `backports.functools-lru-cache` for backporting `functools.lru_cache`.
- `exceptiongroup` for backporting `ExceptionGroup` and `BaseExceptionGroup`.
- `dataclasses` for backporting `dataclasses`.
- `aenum` for backporting `enum`.
- `async_generator` for backporting `async` generators and `asynccontextmanager`.
- `trollius` for backporting `async`/`await` and `asyncio`.

Note that, when distributing compiled Coconut code, if you use any of these backports, you'll need to make sure that the requisite backport module is included as a dependency.

3.4 Integrations

3.4.1 Syntax Highlighting

Text editors with support for Coconut syntax highlighting are:

- **VSCode**: Install Coconut (Official) (for **VSCodium**, install from Open VSX here instead).
- **SublimeText**: See SublimeText section below.
- **Spyder** (or any other editor that supports **Pygments**): See Pygments section below.
- **Vim**: See `coconut.vim`.
- **Emacs**: See `emacs-coconut`/`emacs-ob-coconut`.
- **Atom**: See `language-coconut`.

Alternatively, if none of the above work for you, you can just treat Coconut as Python. Simply set up your editor so it interprets all `.coco` files as Python and that should highlight most of your code well enough (e.g. for IntelliJ IDEA see registering file types).

SublimeText

Coconut syntax highlighting for SublimeText requires that Package Control, the standard package manager for SublimeText, be installed. Once that is done, simply:

1. open the SublimeText command palette by pressing `Ctrl+Shift+P` (or `Cmd+Shift+P` on Mac),
2. type and enter `Package Control: Install Package`, and
3. finally type and enter `Coconut`.

To make sure everything is working properly, open a `.coco` file, and make sure `Coconut` appears in the bottom right-hand corner. If something else appears, like `Plain Text`, click on it, select `Open all with current extension as...` at the top of the resulting menu, and then select `Coconut`.

Note: Coconut syntax highlighting for SublimeText is provided by the sublime-coconut package.

Pygments

The same `pip install coconut` command that installs the Coconut command-line utility will also install the `coconut` Pygments lexer. How to use this lexer depends on the Pygments-enabled application being used, but in general simply use the `.coco` file extension (should be all you need to do for Spyder) and/or enter `coconut` as the language being highlighted and Pygments should be able to figure it out.

For example, this documentation is generated with Sphinx, with the syntax highlighting you see created by adding the line

```
highlight_language = "coconut"
```

to Coconut's `conf.py`.

3.4.2 IPython/Jupyter Support

If you use IPython (the Python kernel for the Jupyter framework) notebooks or console, Coconut can be used as a Jupyter kernel or IPython extension.

Kernel

If Coconut is used as a kernel, all code in the console or notebook will be sent directly to Coconut instead of Python to be evaluated. Otherwise, the Coconut kernel behaves exactly like the IPython kernel, including support for `%magic` commands.

Simply installing Coconut should add a `Coconut` kernel to your Jupyter/IPython notebooks. If you are having issues accessing the Coconut kernel, however, the special command `coconut --jupyter install` will re-install the `Coconut` kernel to ensure it is using the current Python as well as add the additional kernels `Coconut (Default Python)`, `Coconut (Default Python 2)`, and `Coconut (Default Python 3)` which will use, respectively, the Python accessible as `python`, `python2`, and `python3` (these kernels are accessible in the console as `coconut_py`, `coconut_py2`, and `coconut_py3`). Coconut also supports `coconut --jupyter install --user` for user installation. Furthermore, the Coconut kernel fully supports `nb_conda_kernels` to enable accessing the Coconut kernel in one Conda environment from another Conda environment.

The Coconut kernel will always compile using the parameters: `--target sys --line-numbers --keep-lines --no-wrap-types`.

Coconut also provides the following commands:

- `coconut --jupyter notebook` will ensure that the Coconut kernel is available and launch a Jupyter/IPython notebook.
- `coconut --jupyter console` will launch a Jupyter/IPython console using the Coconut kernel.
- `coconut --jupyter lab` will ensure that the Coconut kernel is available and launch JupyterLab.

Additionally, Jupytext contains special support for the Coconut kernel and Coconut contains special support for Papermill.

Extension

If Coconut is used as an extension, a special magic command will send snippets of code to be evaluated using Coconut instead of IPython, but IPython will still be used as the default.

The line magic `%load_ext coconut` will load Coconut as an extension, providing the `%coconut` and `%%coconut` magics and adding Coconut built-ins. The `%coconut` line magic will run a line of Coconut with default parameters, and the `%%coconut` block magic will take command-line arguments on the first line, and run any Coconut code provided in the rest of the cell with those parameters.

Note: Unlike the normal Coconut command-line, `%%coconut` *defaults to the* `sys` *target rather than the* `universal` *target.*

3.4.3 Type Checking

MyPy Integration

Coconut has the ability to integrate with MyPy to provide optional static type_checking, including for all Coconut built-ins. Simply pass `--mypy` to `coconut` to enable MyPy integration, though be careful to pass it only as the last argument, since all arguments after `--mypy` are passed to `mypy`, not Coconut.

You can also run `mypy`—or any other static type checker—directly on the compiled Coconut. If the static type checker is unable to find the necessary stub files, however, then you may need to:

1. run `coconut --mypy install` and
2. tell your static type checker of choice to look in `~/.coconut_stubs` for stub files (for `mypy`, this is done by adding it to your `MYPYPATH`).

To distribute your code with checkable type annotations, you'll need to include `coconut` as a dependency (though a `--no-deps` install should be fine), as installing it is necessary to make the requisite stub files available. You'll also probably want to include a `py.typed` file.

Syntax

To explicitly annotate your code with types to be checked, Coconut supports (on all Python versions):

- Python 3 function type annotations,
- Python 3.6 variable type annotations,
- *Python 3.12 type parameter syntax* for easily adding type parameters to classes, functions, `data` *types*, and type aliases,
- Coconut's own *enhanced type annotation syntax*, and
- Coconut's *protocol intersection operator*.

By default, all type annotations are compiled to Python-2-compatible type comments, which means they should all work on any Python version.

Sometimes, MyPy will not know how to handle certain Coconut constructs, such as `addpattern`. For the `addpattern` case, it is recommended to pass `--allow-redefinition` to MyPy (i.e. run `coconut <args> --mypy --allow-redefinition`), though in some cases `--allow-redefinition` may not be sufficient. In that case, either hide the offending code using *`TYPE_CHECKING`* or put a `# type: ignore` comment on the Coconut line which is generating the line MyPy is complaining about and the comment will be added to every generated line.

Interpreter

Coconut even supports --mypy in the interpreter, which will intelligently scan each new line of code, in the context of previous lines, for newly-introduced MyPy errors. For example:

```
>>> a: str = count()[0]
<string>:14: error: Incompatible types in assignment (expression has type "int",␣
↪variable has type "str")
>>> reveal_type(a)
0
<string>:19: note: Revealed type is 'builtins.unicode'
```

For more information on `reveal_type`*, see* `reveal_type` and `reveal_locals`.

3.4.4 numpy Integration

To allow for better use of numpy objects in Coconut, all compiled Coconut code will do a number of special things to better integrate with numpy (if numpy is available to import when the code is run). Specifically:

- Coconut's *multidimensional array literal and array concatenation syntax* supports numpy objects, including using fast numpy concatenation methods if given numpy arrays rather than Coconut's default much slower implementation built for Python lists of lists.
- Many of Coconut's built-ins include special numpy support, specifically:
  - *fmap* will use numpy.vectorize to map over numpy arrays.
  - *multi_enumerate* allows for easily looping over all the multidimensional indices in a numpy array.
  - *cartesian_product* can compute the Cartesian product of given numpy arrays as a numpy array.
  - *all_equal* allows for easily checking if all the elements in a numpy array are the same.
- numpy.ndarray is registered as a collections.abc.Sequence, enabling it to be used in *sequence patterns*.
- numpy objects are allowed seamlessly in Coconut's *implicit coefficient syntax*, allowing the use of e.g. `A B**2` shorthand for `A * B**2` when A and B are numpy arrays (note: **not** `A @ B**2`).
- Coconut supports @ for matrix multiplication of numpy arrays on all Python versions, as well as supplying the (@) *operator function*.

Additionally, Coconut provides the exact same support for pandas, xarray, pytorch, and jax.numpy objects.

3.4.5 xonsh Support

Coconut integrates with xonsh to allow the use of Coconut code directly from your command line. To use Coconut in xonsh, simply `pip install coconut` should be all you need to enable the use of Coconut syntax in the xonsh shell. In some circumstances, in particular depending on the installed xonsh version, adding `xontrib load coconut` to your xonshrc file might be necessary.

For an example of using Coconut from xonsh:

```
user@computer ~ $ xontrib load coconut
user@computer ~ $ cd ./files
user@computer ~ $ $(ls -la) |> .splitlines() |> len
30
```

Compilation always uses the same parameters as in the *Coconut Jupyter kernel*.

Note that the way that Coconut integrates with `xonsh`, `@(<code>)` syntax and the `execx` command will only work with Python code, not Coconut code. Additionally, Coconut will only compile individual commands—Coconut will not touch the `.xonshrc` or any other `.xsh` files.

3.5 Operators

3.5.1 Precedence

In order of precedence, highest first, the operators supported in Coconut are:

```
===================== =========================
Symbol(s)             Associativity
===================== =========================
await x               n/a
**                    right (allows unary)
f x                   n/a
+, -, ~               unary
*, /, //, %, @        left
+, -                  left
<<, >>                left
&                     left
&:                    yes
^                     left
|                     left
::                    yes (lazy)
..                    yes
a `b` c,              left (captures lambda)
  all custom operators
??                    left (short-circuits)
```

(continues on next page)

(continued from previous page)

```
..>, <.., ..*>, <*..,  n/a (captures lambda)
  ..**>, <**.., etc.
|>, <|, |*>, <*|,      left (captures lambda)
  |**>, <**|, etc.
==, !=, <, >,
  <=, >=,
  in, not in,
  is, is not           n/a
not                    unary
and                    left (short-circuits)
or                     left (short-circuits)
x if c else y,         ternary (short-circuits)
  if c then x else y
=>                     right
====================== ==========================
```

For example, since addition has a higher precedence than piping, expressions of the form `x |> y + z` are equivalent to `x |> (y + z)`.

3.5.2 Lambdas

Coconut provides the simple, clean => operator as an alternative to Python's `lambda` statements. The syntax for the => operator is `(parameters) => expression` (or `parameter => expression` for one-argument lambdas). The operator has the same precedence as the old statement, which means it will often be necessary to surround the lambda in parentheses, and is right-associative.

Additionally, Coconut also supports an implicit usage of the => operator of the form `(=> expression)`, which is equivalent to `((_=None) => expression)`, which allows an implicit lambda to be used both when no arguments are required, and when one argument (assigned to `_`) is required.

Note: If normal lambda syntax is insufficient, Coconut also supports an extended lambda syntax in the form of statement lambdas. *Statement lambdas support full statements rather than just expressions and allow for the use of* pattern-matching function definition.

Deprecated: -> can be used as an alternative to =>, though ->-based lambdas are disabled inside type annotations to avoid conflicting with Coconut's enhanced type annotation syntax.

Rationale

In Python, lambdas are ugly and bulky, requiring the entire word `lambda` to be written out every time one is constructed. This is fine if in-line functions are very rarely needed, but in functional programming in-line functions are an essential tool.

Python Docs

Lambda forms (lambda expressions) have the same syntactic position as expressions. They are a shorthand to create anonymous functions; the expression `(arguments) => expression` yields a function object. The unnamed object behaves like a function object defined with:

```
def <lambda>(arguments):
    return expression
```

Note that functions created with lambda forms cannot contain statements or annotations.

Example

Coconut:

```
dubsums = map((x, y) => 2*(x+y), range(0, 10), range(10, 20))
dubsums |> list |> print
```

Python:

```
dubsums = map(lambda x, y: 2*(x+y), range(0, 10), range(10, 20))
print(list(dubsums))
```

Implicit Lambdas

Coconut also supports implicit lambdas, which allow a lambda to take either no arguments or a single argument. Implicit lambdas are formed with the usual Coconut lambda operator `=>`, in the form `(=> expression)`. This is equivalent to `((_=None) => expression)`. When an argument is passed to an implicit lambda, it will be assigned to `_`, replacing the default value `None`.

Below are two examples of implicit lambdas. The first uses the implicit argument `_`, while the second does not.

Single Argument Example:

```
square = (=> _**2)
```

No-Argument Example:

```
import random

get_random_number = (=> random.random())
```

Note: Nesting implicit lambdas can lead to problems with the scope of the `_` parameter to each lambda. It is recommended that nesting implicit lambdas be avoided.

3.5.3 Partial Application

Coconut uses a `$` sign right after a function's name but before the open parenthesis used to call the function to denote partial application.

Coconut's partial application also supports the use of a `?` to skip partially applying an argument, deferring filling in that argument until the partially-applied function is called. This is useful if you want to partially apply arguments that aren't first in the argument order.

Additionally, `?` can even be used as the value of keyword arguments to convert them into positional arguments. For example, `f$(x=?)` is effectively equivalent to

```
def new_f(x, *args, **kwargs):
    kwargs["x"] = x
    return f(*args, **kwargs)
```

Unlike `functools.partial`, Coconut's partial application will preserve the `__name__` of the wrapped function.

Rationale

Partial application, or currying, is a mainstay of functional programming, and for good reason: it allows the dynamic customization of functions to fit the needs of where they are being used. Partial application allows a new function to be created out of an old function with some of its arguments pre-specified.

Python Docs

Return a new `partial` object which when called will behave like *func* called with the positional arguments *args* and keyword arguments *keywords*. If more arguments are supplied to the call, they are appended to *args*. If additional keyword arguments are supplied, they extend and override *keywords*. Roughly equivalent to:

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*(args + fargs), **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

The `partial` object is used for partial function application which "freezes" some portion of a function's arguments and/or keywords resulting in a new object with a simplified signature.

Example

Coconut:

```
expnums = range(5) |> map$(pow$(?, 2))
expnums |> list |> print
```

Python:

```
# unlike this simple lambda, $ produces a pickleable object
expnums = map(lambda x: pow(x, 2), range(5))
print(list(expnums))
```

3.5.4 Pipes

Coconut uses pipe operators for pipeline-style function application. All the operators have a precedence in-between function composition pipes and comparisons, and are left-associative. All operators also support in-place versions. The different operators are:

```
(|>)     => pipe forward
(|*>)    => multiple-argument pipe forward
(|**>)   => keyword argument pipe forward
(<|)     => pipe backward
(<*|)    => multiple-argument pipe backward
(<**|)   => keyword argument pipe backward
(|?>)    => None-aware pipe forward
(|?*>)   => None-aware multi-arg pipe forward
(|?**>)  => None-aware keyword arg pipe forward
(<?|)    => None-aware pipe backward
(<*?|)   => None-aware multi-arg pipe backward
(<**?|)  => None-aware keyword arg pipe backward
```

The None-aware pipe operators here are equivalent to a monadic bind treating the object as a `Maybe` monad composed of either `None` or the given object. Thus, `x |?> f` is equivalent to `None if x is None else f(x)`. Note that only the object being piped, not the function being piped into, may be `None` for `None`-aware pipes.

Additionally, some special syntax constructs are only available in pipes to enable doing as many operations as possible via pipes if so desired:

- For working with `async` functions in pipes, all non-starred pipes support piping into `await` to await the awaitable piped into them, such that `x |> await` is equivalent to `await x`.
- All non-starred pipes support piping into `(<name> := .)` (mirroring the syntax for *operator implicit partials*) to assign the piped in item to `<name>`.
- All pipe operators support a lambda as the last argument, despite lambdas having a lower precedence. Thus, `a |> x => b |> c` is equivalent to `a |> (x => b |> c)`, not `a |> (x => b) |> c`.

Note: To visually spread operations across several lines, just use parenthetical continuation.

Optimizations

It is common in Coconut to write code that uses pipes to pass an object through a series of *partials* and/or *implicit partials*, as in

```
obj |> .attribute |> .method(args) |> func$(args) |> .[index]
```

which is often much more readable, as it allows the operations to be written in the order in which they are performed, instead of as in

```
func(args, obj.attribute.method(args))[index]
```

where func has to go at the beginning.

If Coconut compiled each of the partials in the pipe syntax as an actual partial application object, it would make the Coconut-style syntax significantly slower than the Python-style syntax. Thus, Coconut does not do that. If any of the above styles of partials or implicit partials are used in pipes, they will whenever possible be compiled to the Python-style syntax, producing no intermediate partial application objects.

This applies even to in-place pipes such as |>=.

Examples

Coconut:

```
def sq(x) = x**2
(1, 2) |*> (+) |> sq |> print
```

```
async def do_stuff(some_data) = (
    some_data
    |> async_func
    |> await
    |> post_proc
)
```

Python:

```
import operator
def sq(x): return x**2
print(sq(operator.add(1, 2)))
```

```
async def do_stuff(some_data):
    return post_proc(await async_func(some_data))
```

3.5.5 Function Composition

Coconut has three basic function composition operators: .., ..>, and <... Both .. and <.. use math-style "backwards" function composition, where the first function is called last, while ..> uses "forwards" function composition, where the first function is called first. Forwards and backwards function composition pipes cannot be used together in the same expression (unlike normal pipes) and have precedence in-between None-coalescing and normal pipes.

The ..> and <.. function composition pipe operators also have multi-arg, keyword, and None variants as with *normal pipes*. The full list of function composition pipe operators is:

```
..>     => forwards function composition pipe
<..     => backwards function composition pipe
..*>    => forwards multi-arg function composition pipe
<*..    => backwards multi-arg function composition pipe
..**>   => forwards keyword arg function composition pipe
<**..   => backwards keyword arg function composition pipe
..?>    => forwards None-aware function composition pipe
<?..    => backwards None-aware function composition pipe
..?*>   => forwards None-aware multi-arg function composition pipe
<*?..   => backwards None-aware multi-arg function composition pipe
```

(continues on next page)

(continued from previous page)

```
..?**> => forwards None-aware keyword arg function composition pipe
<**?.. => backwards None-aware keyword arg function composition pipe
```

Note that None-aware function composition pipes don't allow either function to be None—rather, they allow the return of the first evaluated function to be None, in which case None is returned immediately rather than calling the next function.

The `..` operator has lower precedence than `::` but higher precedence than infix functions while the `..>` pipe operators have a precedence directly higher than normal pipes.

All function composition operators also have in-place versions (e.g. `..=`).

Since all forms of function composition always call the first function in the composition (`f` in `f ..> g` and `g` in `f <.. g`) with exactly the arguments passed into the composition, all forms of function composition will preserve all metadata attached to the first function in the composition, including the function's signature and any of that function's attributes.

Note: for composing `async` functions, see `and_then` and `and_then_await`.

Example

Coconut:

```
fog = f..g
f_into_g = f ..> g
```

Python:

```
# unlike these simple lambdas, Coconut produces pickleable objects
fog = lambda *args, **kwargs: f(g(*args, **kwargs))
f_into_g = lambda *args, **kwargs: g(f(*args, **kwargs))
```

3.5.6 Iterator Slicing

Coconut uses a `$` sign right after an iterator before a slice to perform iterator slicing, as in `it$[:5]`. Coconut's iterator slicing works much the same as Python's sequence slicing, and looks much the same as Coconut's partial application, but with brackets instead of parentheses.

Iterator slicing works just like sequence slicing, including support for negative indices and slices, and support for `slice` objects in the same way as can be done with normal slicing. Iterator slicing makes no guarantee, however, that the original iterator passed to it be preserved (to preserve the iterator, use Coconut's *`reiterable`* built-in).

Coconut's iterator slicing is very similar to Python's `itertools.islice`, but unlike `itertools.islice`, Coconut's iterator slicing supports negative indices, and will preferentially call an object's `__iter_getitem__` (always used if available) or `__getitem__` (only used if the object is a `collections.abc.Sequence`). Coconut's iterator slicing is also optimized to work well with all of Coconut's built-in objects, only computing the elements of each that are actually necessary to extract the desired slice.

Example

Coconut:

```
map(x => x*2, range(10**100))$[-1] |> print
```

Python: *Can't be done without a complicated iterator slicing function and inspection of custom objects. The necessary definitions in Python can be found in the Coconut header.*

3.5.7 Iterator Chaining

Coconut uses the `::` operator for iterator chaining. Coconut's iterator chaining is done lazily, in that the arguments are not evaluated until they are needed. It has a precedence in-between bitwise or and infix calls. Chains are reiterable (can be iterated over multiple times and get the same result) only when the iterators passed in are reiterable. The in-place operator is `::=`.

Note that lazy lists and flatten are used under the hood to implement chaining such that `a :: b` is equivalent to `flatten((|a, b|))`.

Rationale

A useful tool to make working with iterators as easy as working with sequences is the ability to lazily combine multiple iterators together. This operation is called chain, and is equivalent to addition with sequences, except that nothing gets evaluated until it is needed.

Python Docs

Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a single sequence. Chained inputs are evaluated lazily. Roughly equivalent to:

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

Example

Coconut:

```
def N(n=0) = (n,) :: N(n+1)  # no infinite loop because :: is lazy

(range(-10, 0) :: N())$[5:15] |> list |> print
```

Python: *Can't be done without a complicated iterator comprehension in place of the lazy chaining. See the compiled code for the Python syntax.*

3.5.8 Infix Functions

Coconut allows for infix function calling, where an expression that evaluates to a function is surrounded by backticks and then can have arguments placed in front of or behind it. Infix calling has a precedence in-between chaining and `None`-coalescing, and is left-associative.

The allowable notations for infix calls are:

```
x `f` y  =>  f(x, y)
`f` x    =>  f(x)
x `f`    =>  f(x)
`f`      =>  f()
```

Additionally, infix notation supports a lambda as the last argument, despite lambdas having a lower precedence. Thus, ``a `func` b => c`` is equivalent to `func(a, b => c)`.

Coconut also supports infix function definition to make defining functions that are intended for infix usage simpler. The syntax for infix function definition is

```
def <arg> `<name>` <arg>:
    <body>
```

where `<name>` is the name of the function, the `<arg>`s are the function arguments, and `<body>` is the body of the function. If an `<arg>` includes a default, the `<arg>` must be surrounded in parentheses.

Note: Infix function definition can be combined with assignment and/or pattern-matching function definition.

Rationale

A common idiom in functional programming is to write functions that are intended to behave somewhat like operators, and to call and define them by placing them between their arguments. Coconut's infix syntax makes this possible.

Example

Coconut:

```
def a `mod` b = a % b
(x `mod` 2) `print`
```

Python:

```
def mod(a, b): return a % b
print(mod(x, 2))
```

3.5.9 Custom Operators

Coconut allows you to declare your own custom operators with the syntax

```
operator <op>
```

where <op> is whatever sequence of Unicode characters you want to use as a custom operator. The `operator` statement must appear at the top level and only affects code that comes after it.

Once declared, you can use your custom operator anywhere where you would be able to use an *infix function* as well as refer to the actual operator itself with the same `(<op>)` syntax as in other *operator functions*. Since custom operators work like infix functions, they always have the same precedence as infix functions and are always left-associative. Custom operators can be used as binary, unary, or none-ary operators, and both prefix and postfix notation for unary operators is supported.

Some example syntaxes for defining custom operators once declared:

```
def x <op> y: ...
def <op> x = ...
match def (x) <op> (y): ...
(<op>) = ...
from module import name as (<op>)
```

And some example syntaxes for using custom operators:

```
x <op> y
x <op> y <op> z
<op> x
x <op>
x = (<op>)
f(<op>)
(x <op> .)
(. <op> y)
match x <op> in ...: ...
match x <op> y in ...: ...
```

Additionally, to import custom operators from other modules, Coconut supports the special syntax:

```
from <module> import operator <op>
```

Custom operators will often need to be surrounded by whitespace (or parentheses when used as an operator function) to be parsed correctly.

If a custom operator that is also a valid name is desired, you can use a backslash before the name to get back the name instead using Coconut's *keyword/variable disambiguation syntax*.

Note: redefining existing Coconut operators using custom operator definition syntax is forbidden, including Coconut's built-in Unicode operator alternatives.

Examples

Coconut:

```
operator %%
(%%) = math.remainder
10 %% 3 |> print

operator !!
(!!) = bool
!! 0 |> print

operator log10
from math import \log10 as (log10)
100 log10 |> print
```

Python:

```
print(math.remainder(10, 3))

print(bool(0))

print(math.log10(100))
```

3.5.10 None Coalescing

Coconut provides `??` as a `None`-coalescing operator, similar to the `??` null-coalescing operator in C# and Swift. Additionally, Coconut implements all of the `None`-aware operators proposed in PEP 505.

Coconut's `??` operator evaluates to its left operand if that operand is not `None`, otherwise its right operand. The expression `foo ?? bar` evaluates to `foo` as long as it isn't `None`, and to `bar` if it is. The `None`-coalescing operator is short-circuiting, such that if the left operand is not `None`, the right operand won't be evaluated. This allows the right operand to be a potentially expensive operation without incurring any unnecessary cost.

The `None`-coalescing operator has a precedence in-between infix function calls and composition pipes, and is left-associative.

Example

Coconut:

```
could_be_none() ?? calculate_default_value()
```

Python:

```
(lambda result: result if result is not None else calculate_default_value())(could_be_
↪none())
```

Coalescing Assignment Operator

The in-place assignment operator is `??=`, which allows conditionally setting a variable if it is currently `None`.

```
foo = 1
bar = None
foo ??= 10  # foo is still 1
bar ??= 10  # bar is now 10
```

As described with the standard `??` operator, the `None`-coalescing assignment operator will not evaluate the right hand side unless the left hand side is `None`.

```
baz = 0
baz ??= expensive_task()  # right hand side isn't evaluated
```

Other None-Aware Operators

Coconut also allows a single `?` before attribute access, function calling, partial application, and (iterator) indexing to short-circuit the rest of the evaluation if everything so far evaluates to `None`. This is sometimes known as a "safe navigation" operator.

When using a `None`-aware operator for member access, either for a method or an attribute, the syntax is `obj?.method()` or `obj?.attr` respectively. `obj?.attr` is equivalent to `obj.attr if obj is not None else obj`. This does not prevent an `AttributeError` if `attr` is not an attribute or method of `obj`.

The `None`-aware indexing operator is used identically to normal indexing, using `?[]` instead of `[]`. `seq?[index]` is equivalent to the expression `seq[index] is seq is not None else seq`. Using this operator will not prevent an `IndexError` if `index` is outside the bounds of `seq`.

Coconut also supports None-aware *pipe operators* and *function composition pipes*.

Example

Coconut:

```
could_be_none?.attr     # attribute access
could_be_none?(arg)     # function calling
could_be_none?.method() # method calling
could_be_none?$(arg)    # partial application
could_be_none()?[0]     # indexing
could_be_none()?.attr[index].method()
```

Python:

```
import functools
(lambda result: None if result is None else result.attr)(could_be_none())
(lambda result: None if result is None else result(arg))(could_be_none())
(lambda result: None if result is None else result.method())(could_be_none())
(lambda result: None if result is None else functools.partial(result, arg))(could_be_
↪none())
(lambda result: None if result is None else result[0])(could_be_none())
(lambda result: None if result is None else result.attr[index].method())(could_be_none())
```

3.5.11 Protocol Intersection

Coconut uses the `&:` operator to indicate protocol intersection. That is, for two `typing.Protocols` `Protocol1` and `Protocol1`, `Protocol1 &: Protocol2` is equivalent to a `Protocol` that combines the requirements of both `Protocol1` and `Protocol2`.

The recommended way to use Coconut's protocol intersection operator is in combination with Coconut's *operator `Protocols`*. Note, however, that while `&:` will work anywhere, operator `Protocols` will only work inside type annotations (which means, for example, you'll need to do `type HasAdd = (+)` instead of just `HasAdd = (+)`).

See Coconut's *enhanced type annotation* for more information on how Coconut handles type annotations more generally.

Example

Coconut:

```
from typing import Protocol

class X(Protocol):
    x: str

class Y(Protocol):
    y: str

def foo(xy: X &: Y) -> None:
    print(xy.x, xy.y)

type CanAddAndSub = (+) &: (-)
```

Python:

```
from typing import Protocol, TypeVar, Generic

class X(Protocol):
    x: str

class Y(Protocol):
    y: str

class XY(X, Y, Protocol):
    pass

def foo(xy: XY) -> None:
    print(xy.x, xy.y)

T = TypeVar("T", infer_variance=True)
U = TypeVar("U", infer_variance=True)
V = TypeVar("V", infer_variance=True)

class CanAddAndSub(Protocol, Generic[T, U, V]):
    def __add__(self: T, other: U) -> V:
        raise NotImplementedError
    def __sub__(self: T, other: U) -> V:
        raise NotImplementedError
    def __neg__(self: T) -> V:
        raise NotImplementedError
```

3.5.12 Unicode Alternatives

Coconut supports Unicode alternatives to many different operator symbols. The Unicode alternatives are relatively straightforward, and chosen to reflect the look and/or meaning of the original symbol.

Note: these are only the default, built-in unicode operators. Coconut supports custom operator definition *to define your own.*

Full List

```
 (\u21d2)                    => "=>"
→ (\u2192)                     => "->"
× (\xd7)                      => "*" (only multiplication)
↑ (\u2191)                    => "**" (only exponentiation)
÷ (\xf7)                      => "/" (only division)
÷/ (\xf7/)                    => "//"
 (\u207b)                     => "-" (only negation)
 (\u2260) or ¬= (\xac=)       => "!="
 (\u2264) or  (\u2286)       => "<="
 (\u2265) or  (\u2287)       => ">="
 (\u228a)                     => "<"
 (\u228b)                     => ">"
 (\u2229)                     => "&"
 (\u222a)                     => "|"
« (\xab)                      => "<<"
» (\xbb)                      => ">>"
... (\u2026)                    => "..."
 (\u03bb)                     => "lambda"
 (\u21a6)                    => "|>"
 (\u21a4)                    => "<|"
* (*\u21a6)                  => "|*>"
* (\u21a4*)                  => "<*|"
** (**\u21a6)                => "|**>"
** (\u21a4**)                => "<**|"
? (?\u21a6)                  => "|?>"
? (?\u21a4)                  => "<?|"
?* (?*\u21a6)                => "|?*>"
*? (\u21a4*?)                => "<*?|"
?** (?**\u21a6)              => "|?**>"
**? (\u21a4**?)              => "<**?|"
 (\u2218)                     => ".."
> (\u2218>)                   => "..>"
< (<\u2218)                   => "<.."
*> (\u2218*>)                 => "..*>"
<* (<*\u2218)                 => "<*.."
**> (\u2218**>)               => "..**>"
<** (<**\u2218)               => "<**.."
?> (\u2218?>)                 => "..?>"
<? (<?\u2218)                 => "<?.."
?*> (\u2218?*>)               => "..?*>"
<*? (<*?\u2218)               => "<*?.."
?**> (\u2218?**>)             => "..?**>"
```

(continues on next page)

(continued from previous page)

```
<**? (<**?\u2218)          => "<**?.."
 (\u23e8)                  => "e" (in scientific notation)
```

3.6 Keywords

3.6.1 match

Coconut provides fully-featured, functional pattern-matching through its `match` statements. Coconut `match` syntax is a strict superset of Python's `match` syntax.

Note: In describing Coconut's pattern-matching syntax, this section focuses on `match` statements, but Coconut's pattern-matching can also be used in many other places, such as pattern-matching function definition, `case` statements, destructuring assignment, `match data`, and `match for`.

Overview

Match statements follow the basic syntax `match <pattern> in <value>`. The match statement will attempt to match the value against the pattern, and if successful, bind any variables in the pattern to whatever is in the same position in the value, and execute the code below the match statement.

Match statements also support, in their basic syntax, an `if <cond>` that will check the condition after executing the match before executing the code below, and an `else` statement afterwards that will only be executed if the `match` statement is not.

All pattern-matching in Coconut is atomic, such that no assignments will be executed unless the whole match succeeds.

Syntax Specification

Coconut match statement syntax is

```
match <pattern> [not] in <value> [if <cond>]:
    <body>
[else:
    <body>]
```

where `<value>` is the item to match against, `<cond>` is an optional additional check, and `<body>` is simply code that is executed if the header above it succeeds. `<pattern>` follows its own, special syntax, defined roughly as below. In the syntax specification below, brackets denote optional syntax and parentheses followed by a * denote that the syntax may appear zero or more times.

```
pattern ::= and_pattern ("or" and_pattern)*  # match any

and_pattern ::= as_pattern ("and" as_pattern)*  # match all

as_pattern ::= infix_pattern ("as" name)*  # explicit binding

infix_pattern ::= bar_or_pattern ("`" EXPR "`" [EXPR])*  # infix check

bar_or_pattern ::= pattern ("|" pattern)*  # match any

base_pattern ::= (
    "(" pattern ")"                  # parentheses
    | "None" | "True" | "False"      # constants
    | NUMBER                         # numbers
    | STRING                         # strings
    | ["as"] NAME                    # variable binding
    | "==" EXPR                      # equality check
    | "is" EXPR                      # identity check
    | DOTTED_NAME                    # implicit equality check (disabled in␣
↪destructuring assignment)
    | NAME "(" patterns ")"          # classes or data types
    | "data" NAME "(" patterns ")"   # data types
    | "class" NAME "(" patterns ")"  # classes
    | "{" pattern_pairs              # dictionaries
        ["," "**" (NAME | "{}")] "}" #  (keys must be constants or equality checks)
    | ["s" | "f" | "m"] "{"
        pattern_consts
        ["," ("*_" | "*()")]
      "}"                            # sets
    | (EXPR) -> pattern              # view patterns
    | "(" patterns ")"               # sequences can be in tuple form
    | "[" patterns "]"               #  or in list form
    | "(|" patterns "|)"             # lazy lists
    | ("(" | "[")                    # star splits
        [patterns ","]
        "*" pattern
        ["," patterns]
        ["*" pattern
        ["," patterns]]
      (")" | "]")
    | [(                             # sequence splits
        "(" patterns ")"
        | "[" patterns "]"
      ) "+"] NAME ["+" (
        "(" patterns ")"                 # this match must be the same
        | "[" patterns "]"               #  construct as the first match
      )] ["+" NAME ["+" (
        "(" patterns ")"                 # and same here
        | "[" patterns "]"
      )]]
```

(continues on next page)

(continued from previous page)

```
    | [(                              # iterable splits
        "(" patterns ")"
        | "[" patterns "]"
        | "(|" patterns "|)"
      ) "::"] NAME ["::" (
        "(" patterns ")"
        | "[" patterns "]"
        | "(|" patterns "|)"
      )] [ "::" NAME [
        "(" patterns ")"
        | "[" patterns "]"
        | "(|" patterns "|)"
      ]]
    | [STRING "+"] NAME               # complex string matching
        ["+" STRING]
        ["+" NAME ["+" STRING]]
)
```

Semantics Specification

`match` statements will take their pattern and attempt to "match" against it, performing the checks and deconstructions on the arguments as specified by the pattern. The different constructs that can be specified in a pattern, and their function, are:

- Variable Bindings:
  - Implicit Bindings (<`var`>): will match to anything, and will be bound to whatever they match to, with some exceptions:
    - If the same variable is used multiple times, a check will be performed that each use matches to the same value.
    - If the variable name `_` is used, nothing will be bound and everything will always match to it (`_` is Coconut's "wildcard").
  - Explicit Bindings (<`pattern`> `as` <`var`>): will bind <`var`> to <`pattern`>.
- Basic Checks:
  - Constants, Numbers, and Strings: will only match to the same constant, number, or string in the same position in the arguments.
  - Equality Checks (`==`<`expr`>): will check that whatever is in that position is `==` to the expression <`expr`>.
  - Identity Checks (`is` <`expr`>): will check that whatever is in that position `is` the expression <`expr`>.
- Arbitrary Function Patterns:
  - Infix Checks (<`pattern`> `` ` ``<`op`>`` ` `` <`expr`>): will check that the operator <`op`>`$(?, `<`expr`>`)` returns a truthy value when called on whatever is in that position, then matches <`pattern`>. For example, ``x `isinstance` int`` will check that whatever is in that position `isinstance$(?, int)` and bind it to `x`. If <`expr`> is not given, will simply check <`op`> directly rather than <`op`>`$(`<`expr`>`)`. Additionally, `` ` ``<`op`>`` ` `` can instead be a *custom operator* (in that case, no backticks should be used).
  - View Patterns (`(`<`expression`>`) ->` <`pattern`>): calls <`expression`> on the item being matched and matches the result to <`pattern`>. The match fails if a *`MatchError`* is raised. <`expression`> may be unparenthesized only when it is a single atom.

- Class and Data Type Matching:
  - Classes or Data Types (`<name>(<args>)`): will match as a data type if given *a Coconut* `data` *type* (or a tuple of Coconut data types) and a class otherwise.
  - Data Types (`data <name>(<args>)`): will check that whatever is in that position is of data type `<name>` and will match the attributes to `<args>`. Generally, `data <name>(<args>)` will match any data type that could have been constructed with `makedata(<name>, <args>)`. Includes support for positional arguments, named arguments, default arguments, and starred arguments. Also supports strict attributes by prepending a dot to the attribute name that raises `AttributError` if the attribute is not present rather than failing the match (e.g. `data MyData(.my_attr=<some_pattern>)`).
  - Classes (`class <name>(<args>)`): does PEP-634-style class matching. Also supports strict attribute matching as above.
- Mapping Destructuring:
  - Dicts (`{<key>: <value>, ...}`): will match any mapping (`collections.abc.Mapping`) with the given keys and values that match the value patterns. Keys must be constants or equality checks.
  - Dicts With Rest (`{<pairs>, **<rest>}`): will match a mapping (`collections.abc.Mapping`) containing all the `<pairs>`, and will put a `dict` of everything else into `<rest>`. If `<rest>` is `{}`, will enforce that the mapping is exactly the same length as `<pairs>`.
- Set Destructuring:
  - Sets (`s{<constants>, *_}`): will match a set (`collections.abc.Set`) that contains the given `<constants>`, though it may also contain other items. The `s` prefix and the `*_` are optional.
  - Fixed-length Sets (`s{<constants>, *()}`): will match a `set` (`collections.abc.Set`) that contains the given `<constants>`, and nothing else.
  - Frozensets (`f{<constants>}`): will match a `frozenset` (`frozenset`) that contains the given `<constants>`. May use either normal or fixed-length syntax.
  - Multisets (`m{<constants>}`): will match a *`multiset`* (`collections.Counter`) that contains at least the given `<constants>`. May use either normal or fixed-length syntax.
- Sequence Destructuring:
  - Lists (`[<patterns>]`), Tuples (`(<patterns>)`): will only match a sequence (`collections.abc.Sequence`) of the same length, and will check the contents against `<patterns>` (Coconut automatically registers `numpy` arrays and `collections.deque` objects as sequences).
  - Lazy lists (`(|<patterns>|)`): same as list or tuple matching, but checks for an Iterable (`collections.abc.Iterable`) instead of a Sequence.
  - Head-Tail Splits (`<list/tuple> + <var>` or `(<patterns>, *<var>)`): will match the beginning of the sequence against the `<list/tuple>`/`<patterns>`, then bind the rest to `<var>`, and make it the type of the construct used.
  - Init-Last Splits (`<var> + <list/tuple>` or `(*<var>, <patterns>)`): exactly the same as head-tail splits, but on the end instead of the beginning of the sequence.
  - Head-Last Splits (`<list/tuple> + <var> + <list/tuple>` or `(<patterns>, *<var>, <patterns>)`): the combination of a head-tail and an init-last split.
  - Search Splits (`<var1> + <list/tuple> + <var2>` or `(*<var1>, <patterns>, *<var2>)`): searches for the first occurrence of the `<list/tuple>`/`<patterns>` in the sequence, then puts everything before into `<var1>` and everything after into `<var2>`.

- Head-Last Search Splits (`<list/tuple> + <var> + <list/tuple> + <var> + <list/tuple>` or `(<patterns>, *<var>, <patterns>, *<var>, <patterns>)`): the combination of a head-tail split and a search split.
- Iterable Splits (`<list/tuple/lazy list> :: <var> :: <list/tuple/lazy list> :: <var> :: <list/tuple/lazy list>`): same as other sequence destructuring, but works on any iterable (`collections.abc.Iterable`), including infinite iterators (note that if an iterator is matched against it will be modified unless it is *`reiterable`*).
- Complex String Matching (`<string> + <var> + <string> + <var> + <string>`): string matching supports the same destructuring options as above.

Note: Like iterator slicing, iterator and lazy list matching make no guarantee that the original iterator matched against be preserved (to preserve the iterator, use Coconut's `reiterable` built-in).

When checking whether or not an object can be matched against in a particular fashion, Coconut makes use of Python's abstract base classes. Therefore, to ensure proper matching for a custom object, it's recommended to register it with the proper abstract base classes.

Examples

Coconut:

```
def factorial(value):
    match 0 in value:
        return 1
    else: match n `isinstance` int in value if n > 0:  # Coconut allows nesting of␣
↪statements on the same line
        return n * factorial(n-1)
    else:
        raise TypeError("invalid argument to factorial of: "+repr(value))

3 |> factorial |> print
```

Showcases `else` statements, which work much like `else` statements in Python: the code under an `else` statement is only executed if the corresponding match fails.

```
data point(x, y):
    def transform(self, other):
        match point(x, y) in other:
            return point(self.x + x, self.y + y)
        else:
            raise TypeError("arg to transform must be a point")

point(1,2) |> point(3,4).transform |> print
point(1,2) |> (==)$(point(1,2)) |> print
```

Showcases matching to data types and the default equality operator. Values defined by the user with the `data` statement can be matched against and their contents accessed by specifically referencing arguments to the data type's constructor.

```
class Tree
data Empty() from Tree
data Leaf(n) from Tree
data Node(l, r) from Tree
```

(continues on next page)

(continued from previous page)

```
case def depth:
    case(Tree()) = 0
    case(Tree(n)) = 1
    case(Tree(l, r)) = 1 + max(depth(l), depth(r))

Empty() |> depth |> print
Leaf(5) |> depth |> print
Node(Leaf(2), Node(Empty(), Leaf(3))) |> depth |> print
```

Showcases how the combination of data types and match statements can be used to powerful effect to replicate the usage of algebraic data types in other functional programming languages.

```
def duplicate_first([x] + xs as l) =
    [x] + l

[1,2,3] |> duplicate_first |> print
```

Showcases head-tail splitting, one of the most common uses of pattern-matching, where a `+ <var>` (or `:: <var>` for any iterable) at the end of a list or tuple literal can be used to match the rest of the sequence.

```
case def sieve:
    case([head] :: tail) =
        [head] :: sieve(n for n in tail if n % head)
    case((||)) = []
```

Showcases how to match against iterators, namely that the empty iterator case (`(||)`) must come last, otherwise that case will exhaust the whole iterator before any other pattern has a chance to match against it.

```
def odd_primes(p=3) =
    (p,) :: filter(=> _ % p != 0, odd_primes(p + 2))

def primes() =
    (2,) :: odd_primes()

case def twin_primes:
    case(_ :: [p, (.-2) -> p] :: ps) =
        [(p, p+2)] :: twin_primes([p + 2] :: ps)
    case() =
        twin_primes(primes())

twin_primes()$[:5] |> list |> print
```

Showcases the use of an iterable search pattern and a view pattern to construct an iterator of all twin primes.

Python: *Can't be done without a long series of checks for each `match` statement. See the compiled code for the Python syntax.*

3.6.2 case

Coconut's case blocks serve as an extension of Coconut's match statement for performing multiple match statements against the same value, where only one of them should succeed. Unlike lone match statements, only one match statement inside of a case block will ever succeed, and thus more general matches should be put below more specific ones.

Coconut's case blocks are an extension of Python 3.10's case blocks to support additional pattern-matching constructs added by Coconut (and Coconut will ensure that they work on all Python versions, not just 3.10+).

Each pattern in a case block is checked until a match is found, and then the corresponding body is executed, and the case block terminated. The syntax for case blocks is

```
match <value>:
    case <pattern> [if <cond>]:
        <body>
    case <pattern> [if <cond>]:
        <body>
    ...
[else:
    <body>]
```

where <pattern> is any match pattern, <value> is the item to match against, <cond> is an optional additional check, and <body> is simply code that is executed if the header above it succeeds. Note the absence of an in in the match statements: that's because the <value> in case <value> is taking its place. If no else is present and no match succeeds, then the case statement is simply skipped over as with *match statements* (though unlike *destructuring assignments*).

Deprecated: Additionally, cases or case can be used as the top-level keyword instead of match, and in such a block match is used for each case rather than case.

Examples

Coconut:

```
def classify_sequence(value):
    out = ""        # unlike with normal matches, only one of the patterns
    match value:     #  will match, and out will only get appended to once
        case ():
            out += "empty"
        case (_,):
            out += "singleton"
        case (x,x):
            out += "duplicate pair of "+str(x)
        case (_,_):
            out += "pair"
        case _ is (tuple, list):
            out += "sequence"
    else:
        raise TypeError()
    return out

[] |> classify_sequence |> print
() |> classify_sequence |> print
```

(continues on next page)

(continued from previous page)

```
[1] |> classify_sequence |> print
(1,1) |> classify_sequence |> print
(1,2) |> classify_sequence |> print
(1,1,1) |> classify_sequence |> print
```

Example of using Coconut's `case` *syntax.*

```
cases {"a": 1, "b": 2}:
    match {"a": a}:
        pass
    match _:
        assert False
assert a == 1
```

Example of the `cases` *keyword instead.*

Python: *Can't be done without a long series of checks for each* `match` *statement. See the compiled code for the Python syntax.*

3.6.3 `match for`

Coconut supports pattern-matching in for loops, where the pattern is matched against each item in the iterable. The syntax is

```
[match] for <pattern> in <iterable>:
    <body>
```

which is equivalent to the *destructuring assignment*

```
for elem in <iterable>:
    match <pattern> = elem
    <body>
```

Pattern-matching can also be used in `async for` loops, with both `async match for` and `match async for` allowed as explicit syntaxes.

Example

Coconut:

```
for {"user": uid, **_} in get_data():
    print(uid)
```

Python:

```
for user_data in get_data():
    uid = user_data["user"]
    print(uid)
```

3.6.4 `data`

Coconut's `data` keyword is used to create immutable, algebraic data types, including built-in support for destructuring *pattern-matching* and *fmap*.

The syntax for `data` blocks is a cross between the syntax for functions and the syntax for classes. The first line looks like a function definition, but the rest of the body looks like a class, usually containing method definitions. This is because while `data` blocks actually end up as classes in Python, Coconut automatically creates a special, immutable constructor based on the given arguments.

Coconut data statement syntax looks like:

```
data <name>(<args>) [from <inherits>]:
    <body>
```

`<name>` is the name of the new data type, `<args>` are the arguments to its constructor as well as the names of its attributes, `<body>` contains the data type's methods, and `<inherits>` optionally contains any desired base classes.

Coconut allows data fields in `<args>` to have defaults and/or *type annotations* attached to them, and supports a starred parameter at the end to collect extra arguments. Additionally, Coconut allows type parameters to be specified in brackets after `<name>` using Coconut's *type parameter syntax*.

Writing constructors for `data` types must be done using the `__new__` method instead of the `__init__` method. For helping to easily write `__new__` methods, Coconut provides the *makedata* built-in.

Subclassing `data` types can be done easily by inheriting from them either in another `data` statement or a normal Python `class`. If a normal `class` statement is used, making the new subclass immutable will require adding the line

```
__slots__ = ()
```

which will need to be put in the subclass body before any method or attribute definitions. If you need to inherit magic methods from a base class in your `data` type, such subclassing is the recommended method, as the `data ... from ...` syntax will overwrite any magic methods in the base class with magic methods built for the new `data` type.

Compared to *`namedtuples`*, from which `data` types are derived, `data` types:

- use typed equality,
- don't support tuple addition or multiplication (unless explicitly defined via special methods in the `data` body),
- support starred, typed, and *pattern-matching* arguments, and
- have special *pattern-matching* behavior.

Like `namedtuples`, `data` types also support a variety of extra methods, such as `._asdict()` and `._replace(**kwargs)`.

Rationale

A mainstay of functional programming that Coconut improves in Python is the use of values, or immutable data types. Immutable data can be very useful because it guarantees that once you have some data it won't change, but in Python creating custom immutable data types is difficult. Coconut makes it very easy by providing `data` blocks.

Examples

Coconut:

```
data vector2(x:int=0, y:int=0):
    def __abs__(self):
        return (self.x**2 + self.y**2)**.5

v = vector2(3, 4)
v |> print  # all data types come with a built-in __repr__
v |> abs |> print
v.x = 2  # this will fail because data objects are immutable
vector2() |> print
```

Showcases the syntax, features, and immutable nature of `data` *types, as well as the use of default arguments and type annotations.*

```
data Empty()
data Leaf(n)
data Node(l, r)

case def size:
    case(Empty()) = 0
    case(Leaf(n)) = 1
    case(Node(l, r)) = size(l) + size(r)

size(Node(Empty(), Leaf(10))) == 1
```

Showcases the use of pattern-matching to deconstruct `data` *types.*

```
data vector(*pts):
    """Immutable arbitrary-length vector."""

    def __abs__(self) =
        self.pts |> map$(pow$(?, 2)) |> sum |> pow$(?, 0.5)

    def __add__(self, other) =
        vector(*other_pts) = other
        assert len(other_pts) == len(self.pts)
        map((+), self.pts, other_pts) |*> vector

    def __neg__(self) =
        self.pts |> map$((-)) |*> vector

    def __sub__(self, other) =
        self + -other
```

Showcases starred `data` *declaration.*

Python: *Can't be done without a series of method definitions for each data type. See the compiled code for the Python syntax.*

`match data`

In addition to normal `data` statements, Coconut also supports pattern-matching data statements that enable the use of Coconut's pattern-matching syntax to define the data type's constructor. Pattern-matching data types look like

```
[match] data <name>(<patterns>) [from <base class>]:
    <body>
```

where `<patterns>` are exactly as in *pattern-matching functions*.

It is important to keep in mind that pattern-matching data types vary from normal data types in a variety of ways. First, like pattern-matching functions, they raise *`MatchError`* instead of `TypeError` when passed the wrong arguments. Second, pattern-matching data types will not do any special handling of starred arguments. Thus,

```
data vec(*xs)
```

when iterated over will iterate over all the elements of `xs`, but

```
match data vec(*xs)
```

when iterated over will only give the single element `xs`.

Example

Coconut:

```
data namedpt(name `isinstance` str, x `isinstance` int, y `isinstance` int):
    def mag(self) = (self.x**2 + self.y**2)**0.5
```

Python: *Can't be done without a series of method definitions for each data type. See the compiled code for the Python syntax.*

3.6.5 `where`

Coconut's `where` statement is fairly straightforward. The syntax for a `where` statement is just

```
<stmt> where:
    <body>
```

which executes `<body>` followed by `<stmt>`, with the exception that any new variables defined in `<body>` are available *only* in `<stmt>` (though they are only mangled, not deleted, such that e.g. lambdas can still capture them).

Example

Coconut:

```
result = a + b where:
    a = 1
    b = 2
```

Python:

```
_a = 1
_b = 2
result = _a + _b
```

3.6.6 `async with for`

In modern Python `async` code, such as when using `contextlib.aclosing`, it is often recommended to use a pattern like

```
async with aclosing(my_generator()) as values:
    async for value in values:
        ...
```

since it is substantially safer than the more syntactically straightforward

```
async for value in my_generator():
    ...
```

This is especially true when using `trio`, which completely disallows iterating over `async` generators with `async for`, instead requiring the above `async with ... async for` pattern using utilities such as `trio_util.trio_async_generator`.

Since this pattern can often be quite syntactically cumbersome, Coconut provides the shortcut syntax

```
async with for value in aclosing(my_generator()):
    ...
```

which compiles to exactly the pattern above.

`async with for` also *supports pattern-matching, just like normal Coconut* `for` *loops*.

Example

Coconut:

```
from trio_util import trio_async_generator

@trio_async_generator
async def my_generator():
    # yield values, possibly from a nursery or cancel scope
    # ...

async with for value in my_generator():
    print(value)
```

Python:

```
from trio_util import trio_async_generator

@trio_async_generator
async def my_generator():
    # yield values, possibly from a nursery or cancel scope
    # ...
```

(continues on next page)

(continued from previous page)

```
async with my_generator() as agen:
    async for value in agen:
        print(value)
```

3.6.7 Handling Keyword/Variable Name Overlap

In Coconut, the following keywords are also valid variable names:

- `data`
- `match`
- `case`
- `cases`
- `addpattern`
- `where`
- `operator`
- `then`
- (a *Unicode alternative* for `lambda`)

While Coconut can usually disambiguate these two use cases, special syntax is available for disambiguating them if necessary. Note that, if what you're writing can be interpreted as valid Python 3, Coconut will always prefer that interpretation by default.

To specify that you want a *variable*, prefix the name with a backslash as in `\data`, and to specify that you want a *keyword*, prefix the name with a colon as in `:match`.

Additionally, backslash syntax for escaping variable names can also be used to distinguish between variable names and *custom operators* as well as explicitly signify that an assignment to a built-in is desirable to dismiss *`--strict` warnings*.

Finally, such disambiguation syntax can also be helpful for letting syntax highlighters know what you're doing.

Examples

Coconut:

```
\data = 5
print(\data)
```

```
# without the colon, Coconut will interpret this as the valid Python match[x, y] = input_
↪list
:match [x, y] = input_list
```

Python:

```
data = 5
print(data)
```

```
x, y = input_list
```

3.7 Expressions

3.7.1 Statement Lambdas

The statement lambda syntax is an extension of the *normal lambda syntax* to support statements, not just expressions.

The syntax for a statement lambda is

```
[async|match|copyclosure] def (arguments) => statement; statement; ...
```

where `arguments` can be standard function arguments or *pattern-matching function definition* arguments and `statement` can be an assignment statement or a keyword statement. Note that the `async`, `match`, and `copyclosure` keywords can be combined and can be in any order.

If the last `statement` (not followed by a semicolon) in a statement lambda is an `expression`, it will automatically be returned.

Statement lambdas also support implicit lambda syntax such that `def => _` is equivalent to `def (_=None) => _` as well as explicitly marking them as pattern-matching such that `match def (x) => x` will be a pattern-matching function.

Additionally, statement lambdas have slightly different scoping rules than normal lambdas. When a statement lambda is inside of an expression with an expression-local variable, such as a normal lambda or comprehension, the statement lambda will capture the value of the variable at the time that the statement lambda is defined (rather than a reference to the overall namespace as with normal lambdas). As a result, while `[=> y for y in range(2)] |> map$(call) |> list` is `[1, 1]`, `[def => y for y in range(2)] |> map$(call) |> list` is `[0, 1]`. Note that this only works for expression-local variables: to copy the entire namespace at the time of function definition, use `copyclosure` (which can be used with statement lambdas).

Note that statement lambdas have a lower precedence than normal lambdas and thus capture things like trailing commas. To avoid confusion, statement lambdas should always be wrapped in their own set of parentheses.

Deprecated: Statement lambdas also support -> instead of =>. Note that when using ->, any lambdas in the body of the statement lambda must also use -> rather than =>.

Example

Coconut:

```
L |> map$(def (x) =>
    y = 1/x;
    y*(1 - y))
```

Python:

```
def _lambda(x):
    y = 1/x
    return y*(1 - y)
map(_lambda, L)
```

Type annotations

Another case where statement lambdas would be used over standard lambdas is when the parameters to the lambda are typed with type annotations. Statement lambdas use the standard Python syntax for adding type annotations to their parameters:

```
f = def (c: str) -> None => print(c)

g = def (a: int, b: int) -> int => a ** b
```

Deprecated: if the deprecated -> is used in place of =>, then return type annotations will not be available.

3.7.2 Operator Functions

Coconut uses a simple operator function short-hand: surround an operator with parentheses to retrieve its function. Similarly to iterator comprehensions, if the operator function is the only argument to a function, the parentheses of the function call can also serve as the parentheses for the operator function.

All operator functions also support *implicit partial application*, e.g. `(. + 1)` is equivalent to `(=> _ + 1)`.

Rationale

A very common thing to do in functional programming is to make use of function versions of built-in operators: currying them, composing them, and piping them. To make this easy, Coconut provides a short-hand syntax to access operator functions.

Full List

```
(::)         => (itertools.chain)  # will not evaluate its arguments lazily
($)          => (functools.partial)
(.)          => (getattr)
(,)          => (*args) => args  # (but pickleable)
(+)          => (operator.add)
(-)          => # 1 arg: operator.neg, 2 args: operator.sub
(*)          => (operator.mul)
(**)         => (operator.pow)
(/)          => (operator.truediv)
(//)         => (operator.floordiv)
(%)          => (operator.mod)
(&)          => (operator.and_)
(^)          => (operator.xor)
(|)          => (operator.or_)
(<<)         => (operator.lshift)
(>>)         => (operator.rshift)
(<)          => (operator.lt)
(>)          => (operator.gt)
(==)         => (operator.eq)
(<=)         => (operator.le)
(>=)         => (operator.ge)
(!=)         => (operator.ne)
(~)          => (operator.inv)
(@)          => (operator.matmul)
(|>)         => # pipe forward
(|*>)        => # multi-arg pipe forward
(|**>)       => # keyword arg pipe forward
(<|)         => # pipe backward
(<*|)        => # multi-arg pipe backward
(<**|)       => # keyword arg pipe backward
(|?>)        => # None-aware pipe forward
(|?*>)       => # None-aware multi-arg pipe forward
(|?**>)      => # None-aware keyword arg pipe forward
(<?|)        => # None-aware pipe backward
(<*?|)       => # None-aware multi-arg pipe backward
(<**?|)      => # None-aware keyword arg pipe backward
(..), (<..) => # backward function composition
(..>)        => # forward function composition
(<*..)       => # multi-arg backward function composition
(..*>)       => # multi-arg forward function composition
(<**..)      => # keyword arg backward function composition
(..**>)      => # keyword arg forward function composition
(not)        => (operator.not_)
(and)        => # boolean and
(or)         => # boolean or
(is)         => (operator.is_)
(is not)     => (operator.is_not)
(in)         => (operator.contains)
(not in)     => # negative containment
(assert)     => def (cond, msg=None) => assert cond, msg  # (but a better msg if msg is␣
↪None)
```

(continues on next page)

(continued from previous page)

```
(raise)     => def (exc=None, from_exc=None) => raise exc from from_exc  # or just raise␣
↪if exc is None
# operator functions for multidimensional array concatenation use brackets:
[;]         => def (x, y) => [x; y]
[;;]        => def (x, y) => [x;; y]
...  # and so on for any number of semicolons
# there are two operator functions that don't require parentheses:
.[]         => (operator.getitem)
.$[]        => # iterator slicing operator
```

For an operator function for function application, see `call`.

Though no operator function is available for `await`, an equivalent syntax is available for *pipes* in the form of `awaitable |> await`.

Example

Coconut:

```
(range(0, 5), range(5, 10)) |*> map$(+) |> list |> print
```

Python:

```
import operator
print(list(map(operator.add, range(0, 5), range(5, 10))))
```

3.7.3 Implicit Partial Application

Coconut supports a number of different syntactical aliases for common partial application use cases. These are:

```
# attribute access and method calling
.attr1.attr2        =>  operator.attrgetter("attr1.attr2")
.method(args)       =>  operator.methodcaller("method", args)
.attr.method(args)  =>  .attr ..> .method(args)

# indexing
.[a:b:c]            =>  operator.itemgetter(slice(a, b, c))
.[x][y]             => .[x] ..> .[y]
.method[x]          => .method ..> .[x]
seq[]               =>  operator.getitem$(seq)

# iterator indexing
.$[a:b:c]           =>  # the equivalent of .[a:b:c] for iterators
.$[x]$[y]           => .$[x] ..> .$[y]
iter$[]             =>  # the equivalent of seq[] for iterators

# currying
func$               =>  ($)$(func)
```

In addition, for every Coconut *operator function*, Coconut supports syntax for implicitly partially applying that operator function as

```
(. <op> <arg>)
(<arg> <op> .)
```

where `<op>` is the operator function and `<arg>` is any expression. Note that, as with operator functions themselves, the parentheses are necessary for this type of implicit partial application. This syntax is slightly different for multidimensional array concatenation operator functions, which use brackets instead of parentheses.

Furthermore, Coconut also supports implicit operator function partials for arbitrary functions as

```
(. `<name>` <arg>)
(<arg> `<name>` .)
```

based on Coconut's *infix notation* where `<name>` is the name of the function. Additionally, `` `<name>` `` can instead be a *custom operator* (in that case, no backticks should be used).

Deprecated: Coconut also supports `obj.` *as an implicit partial for* `getattr$(obj)`*, but its usage is deprecated and will show a warning to switch to* `getattr$(obj)` *instead.*

Example

Coconut:

```
1 |> "123"[]
mod$ <| 5 <| 3
3 |> (.*2) |> (.+1)
```

Python:

```
"123"[1]
mod(5, 3)
(3 * 2) + 1
```

3.7.4 Enhanced Type Annotation

Since Coconut syntax is a superset of the latest Python 3 syntax, it supports Python 3 function type annotation syntax and Python 3.6 variable type annotation syntax. By default, Coconut compiles all type annotations into Python-2-compatible type comments. If you want to keep the type annotations instead, simply pass a `--target` that supports them.

Since not all supported Python versions support the `typing` module, Coconut provides the *`TYPE_CHECKING`* built-in for hiding your `typing` imports and `TypeVar` definitions from being executed at runtime. Coconut will also automatically use `typing_extensions` over `typing` objects at runtime when importing them from `typing`, even when they aren't natively supported on the current Python version (this works even if you just do `import typing` and then `typing.<Object>`).

Furthermore, when compiling type annotations to Python 3 versions without PEP 563 support, Coconut wraps annotation in strings to prevent them from being evaluated at runtime (to avoid this, e.g. if you want to use annotations at runtime, `--no-wrap-types` will disable all wrapping, including via PEP 563 support). Only on `--target 3.13` does `--no-wrap-types` do nothing, since there PEP 649 support is used instead.

Additionally, Coconut adds special syntax for making type annotations easier and simpler to write. When inside of a type annotation, Coconut treats certain syntax constructs differently, compiling them to type annotations instead of what they would normally represent. Specifically, Coconut applies the following transformations:

```
A | B
    => typing.Union[A, B]
(A; B)
    => typing.Tuple[A, B]
A?
    => typing.Optional[A]
A[]
    => typing.Sequence[A]
A$[]
    => typing.Iterable[A]
() -> <ret>
    => typing.Callable[[], <ret>]
<arg> -> <ret>
    => typing.Callable[[<arg>], <ret>]
(<args>) -> <ret>
    => typing.Callable[[<args>], <ret>]
-> <ret>
    => typing.Callable[..., <ret>]
(<args>, **<ParamSpec>) -> <ret>
    => typing.Callable[typing.Concatenate[<args>, <ParamSpec>], <ret>]
async (<args>) -> <ret>
    => typing.Callable[[<args>], typing.Awaitable[<ret>]]
```

where `typing` is the Python 3.5 built-in `typing` module. For more information on the Callable syntax, see PEP 677, which Coconut fully supports.

Additionally, many of Coconut's *operator functions* will compile into equivalent `Protocols` instead when inside a type annotation. See below for the full list and specification.

Note: The transformation to `Union` *is not done on Python 3.10 as Python 3.10 has native PEP 604 support.*

To use these transformations in a type alias, use the syntax

```
type <name> = <type>
```

which will allow `<type>` to include Coconut's special type annotation syntax and type `<name>` as a `typing.TypeAlias`. If you try to instead just do a naked `<name> = <type>` type alias, Coconut won't be able to tell you're attempting a type alias and thus won't apply any of the above transformations.

Such type alias statements—as well as all `class`, `data`, and function definitions in Coconut—also support Coconut's *type parameter syntax*, allowing you to do things like `type OrStr[T] = T | str`.

Supported Protocols

Using Coconut's *operator function* syntax inside of a type annotation will instead produce a `Protocol` corresponding to that operator (or raise a syntax error if no such `Protocol` is available). All available `Protocols` are listed below.

For the operator functions

```
(+)
(*)
(**)
(/)
(//)
(%)
```

(continues on next page)

(continued from previous page)

```
(&)
(^)
(|)
(<<)
(>>)
(@)
```

the resulting `Protocol` is

```
class SupportsOp[T, U, V](Protocol):
    def __op__(self: T, other: U) -> V:
        raise NotImplementedError(...)
```

where `__op__` is the magic method corresponding to that operator.

For the operator function `(-)`, the resulting `Protocol` is:

```
class SupportsMinus[T, U, V](Protocol):
    def __sub__(self: T, other: U) -> V:
        raise NotImplementedError
    def __neg__(self: T) -> V:
        raise NotImplementedError
```

For the operator function `(~)`, the resulting `Protocol` is:

```
class SupportsInv[T, V](Protocol):
    def __invert__(self: T) -> V:
        raise NotImplementedError(...)
```

`List` vs. `Sequence`

Importantly, note that `T[]` does not map onto `typing.List[T]` but onto `typing.Sequence[T]`. This allows the resulting type to be covariant, such that if `U` is a subtype of `T`, then `U[]` is a subtype of `T[]`. Additionally, `Sequence[T]` allows for tuples, and when writing in an idiomatic functional style, assignment should be rare and tuples should be common. Using `Sequence` covers both cases, accommodating tuples and lists and preventing indexed assignment. When an indexed assignment is attempted into a variable typed with `Sequence`, MyPy will generate an error:

```
foo: int[] = [0, 1, 2, 3, 4, 5]
foo[0] = 1   # MyPy error: "Unsupported target for indexed assignment"
```

If you want to use `List` instead (e.g. if you want to support indexed assignment), use the standard Python 3.5 variable type annotation syntax: `foo: List[<type>]`.

Note: To easily view your defined types, see `reveal_type` *and* `reveal_locals`.

Example

Coconut:

```
def int_map(
    f: int -> int,
    xs: int[],
) -> int[] =
    xs |> map$(f) |> list

type CanAddAndSub = (+) &: (-)
```

Python:

```
import typing  # unlike this typing import, Coconut produces universal code

def int_map(
    f,  # type: typing.Callable[[int], int]
    xs,  # type: typing.Sequence[int]
):
    # type: (...) -> typing.Sequence[int]
    return list(map(f, xs))

T = typing.TypeVar("T", infer_variance=True)
U = typing.TypeVar("U", infer_variance=True)
V = typing.TypeVar("V", infer_variance=True)
class CanAddAndSub(typing.Protocol, typing.Generic[T, U, V]):
    def __add__(self: T, other: U) -> V:
        raise NotImplementedError
    def __sub__(self: T, other: U) -> V:
        raise NotImplementedError
    def __neg__(self: T) -> V:
        raise NotImplementedError
```

3.7.5 Multidimensional Array Literal/Concatenation Syntax

Coconut supports multidimensional array literal and array concatenation/stack syntax.

By default, all multidimensional array syntax will simply operate on Python lists of lists (or any non-`str` `Sequence`). However, if *numpy* objects are used, the appropriate `numpy` calls will be made instead. To give custom objects multidimensional array concatenation support, define `type(obj).__matconcat__` (should behave as `np.concat`), `obj.ndim` (should behave as `np.ndarray.ndim`), and `obj.reshape` (should behave as `np.ndarray.reshape`).

As a simple example, 2D matrices can be constructed by separating the rows with `;;` inside of a list literal:

```
>>> [1, 2 ;;
     3, 4]

[[1, 2], [3, 4]]
>>> import numpy as np
>>> np.array([1, 2 ;; 3, 4])
array([[1, 2],
       [3, 4]])
```

As can be seen, `np.array` (or equivalent) can be used to turn the resulting list of lists into an actual array. This syntax works because `;;` inside of a list literal functions as a concatenation/stack along the `-2` axis (with the inner arrays being broadcast to `(1, 2)` arrays before concatenation). Note that this concatenation is done entirely in Python lists of lists here, since the `np.array` call comes only at the end.

In general, the number of semicolons indicates the dimension from the end on which to concatenate. Thus, `;` indicates conatenation along the `-1` axis, `;;` along the `-2` axis, and so on. Before concatenation, arrays are always broadcast to a shape which is large enough to allow the concatenation.

Thus, if `a` is a `numpy` array, `[a; a]` is equivalent to `np.concatenate((a, a), axis=-1)`, while `[a ;; a]` would be equivalent to a version of `np.concatenate((a, a), axis=-2)` that also ensures that `a` is at least two dimensional. For normal lists of lists, the behavior is the same, but is implemented without any `numpy` calls.

If multiple different concatenation operators are used, the operators with the least number of semicolons will bind most tightly. Thus, you can write a 3D array literal as:

```
>>> [1, 2 ;;
     3, 4
     ;;;
     5, 6 ;;
     7, 8]

[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
```

Note: the operator functions for multidimensional array concatenation are spelled `[;]`, `[;;]`, etc. (with any number of parentheses). The implicit partials are similarly spelled `[. ; x]`, `[x ; .]`, etc.

Comparison to Julia

Coconut's multidimensional array syntax is based on that of Julia. The primary difference between Coconut's syntax and Julia's syntax is that multidimensional arrays are row-first in Coconut (following `numpy`), but column-first in Julia. Thus, `;` is vertical concatenation in Julia but **horizontal concatenation** in Coconut and `;;` is horizontal concatenation in Julia but **vertical concatenation** in Coconut.

Examples

Coconut:

```
>>> [[1;;2] ; [3;;4]]
[[1, 3], [2, 4]]
```

Array literals can be written in column-first order if the columns are first created via vertical concatenation (`;;`) and then joined via horizontal concatenation (`;`).

```
>>> [range(3) |> list ;; x+1 for x in range(3)]
[[0, 1, 2], [1, 2, 3]]
```

Arbitrary expressions, including comprehensions, are allowed in multidimensional array literals.

```
>>> import numpy as np
>>> a = np.array([1, 2 ;; 3, 4])
>>> [a ; a]
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
```

(continues on next page)

(continued from previous page)

```
>>> [a ;; a]
array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])
>>> [a ;;; a]
array([[[1, 2],
        [3, 4]],

       [[1, 2],
        [3, 4]]])
```

General showcase of how the different concatenation operators work using `numpy` *arrays.*

Python: *The equivalent Python array literals can be seen in the printed representations in each example.*

3.7.6 Lazy Lists

Coconut supports the creation of lazy lists, where the contents in the list will be treated as an iterator and not evaluated until they are needed. Unlike normal iterators, however, lazy lists can be iterated over multiple times and still return the same result. Lazy lists can be created in Coconut simply by surrounding a comma-separated list of items with `(|` and `|)` (so-called "banana brackets") instead of `[` and `]` for a list or `(` and `)` for a tuple.

Lazy lists use *reiterable* under the hood to enable them to be iterated over multiple times. Lazy lists will even continue to be reiterable when combined with *lazy chaining*.

Rationale

Lazy lists, where sequences are only evaluated when their contents are requested, are a mainstay of functional programming, allowing for dynamic evaluation of the list's contents.

Example

Coconut:

```
(| print("hello,"), print("world!") |) |> consume
```

Python: *Can't be done without a complicated iterator comprehension in place of the lazy list. See the compiled code for the Python syntax.*

3.7.7 Implicit Function Application and Coefficients

Coconut supports implicit function application of the form `f x y`, which is compiled to `f(x, y)` (note: **not** `f(x)(y)` as is common in many languages with automatic currying).

Additionally, if the first argument is not callable, and is instead an `int`, `float`, `complex`, or *numpy* object, then the result is multiplication rather than function application, such that `2 x` is equivalent to `2*x`.

Though the first item may be any atom, following arguments are highly restricted, and must be:

- variables/attributes (e.g. `a.b`),
- literal constants (e.g. `True`),

- number literals (e.g. `1.5`) (and no binary, hex, or octal), or
- one of the above followed by an exponent (e.g. `a**-5`).

For example, `(f .. g) x 1` will work, but `f x [1]`, `f x (1+2)`, and `f "abc"` will not.

Implicit function application and coefficient syntax is only intended for simple use cases. For more complex cases, use the standard multiplication operator `*`, standard function application, or *pipes*.

Implicit function application and coefficient syntax has a lower precedence than `**` but a higher precedence than unary operators. As a result, `2 x**2 + 3 x` is equivalent to `2 * x**2 + 3 * x`.

Due to potential confusion, some syntactic constructs are explicitly disallowed in implicit function application and coefficient syntax. Specifically:

- Strings are always disallowed everywhere in implicit function application / coefficient syntax due to conflicting with Python's implicit string concatenation.
- Multiplying two or more numeric literals with implicit coefficient syntax is prohibited, so `10 20` is not allowed.
- `await` is not allowed in front of implicit function application and coefficient syntax. To use `await`, simply parenthesize the expression, as in `await (f x)`.

Note: implicit function application and coefficient syntax is disabled when using Coconut in `xonsh` *due to conflicting with console commands.*

Examples

Coconut:

```
def f(x, y) = (x, y)
print(f 5 10)
```

```
def p1(x) = x + 1
print <| p1 5
```

```
quad = 5 x**2 + 3 x + 1
```

Python:

```
def f(x, y): return (x, y)
print(f(100, 5+6))
```

```
def p1(x): return x + 1
print(p1(5))
```

```
quad = 5 * x**2 + 3 * x + 1
```

3.7.8 Keyword Argument Name Elision

When passing in long variable names as keyword arguments of the same name, Coconut supports the syntax

```
f(long_variable_name=)
```

as a shorthand for

```
f(long_variable_name=long_variable_name)
```

Such syntax is also supported in *partial application* and *anonymous* `namedtuples`.

Deprecated: Coconut also supports `f(...=long_variable_name)` *as an alternative shorthand syntax.*

Example

Coconut:

```
really_long_variable_name_1 = get_1()
really_long_variable_name_2 = get_2()
main_func(
    really_long_variable_name_1=,
    really_long_variable_name_2=,
)
```

Python:

```
really_long_variable_name_1 = get_1()
really_long_variable_name_2 = get_2()
main_func(
    really_long_variable_name_1=really_long_variable_name_1,
    really_long_variable_name_2=really_long_variable_name_2,
)
```

3.7.9 Anonymous Namedtuples

Coconut supports anonymous `namedtuple` literals, such that `(a=1, b=2)` can be used just as `(1, 2)`, but with added names. Anonymous `namedtuples` are always pickleable.

The syntax for anonymous namedtuple literals is:

```
(<name> [: <type>] = <value>, ...)
```

where, if `<type>` is given for any field, `typing.NamedTuple` is used instead of `collections.namedtuple`.

Anonymous `namedtuples` also support *keyword argument name elision*.

`_namedtuple_of`

On Python versions >=`3.6`, `_namedtuple_of` is provided as a built-in that can mimic the behavior of anonymous namedtuple literals such that `_namedtuple_of(a=1, b=2)` is equivalent to `(a=1, b=2)`. Since `_namedtuple_of` is only available on Python 3.6 and above, however, it is generally recommended to use anonymous namedtuple literals instead, as they work on any Python version.

`_namedtuple_of` is just provided to give namedtuple literals a representation that corresponds to an expression that can be used to recreate them.

Example

Coconut:

```
users = [
    (id=1, name="Alice"),
    (id=2, name="Bob"),
]
```

Python:

```
from collections import namedtuple

users = [
    namedtuple("_", "id, name")(1, "Alice"),
    namedtuple("_", "id, name")(2, "Bob"),
]
```

3.7.10 Set Literals

Coconut allows an optional `s` to be prepended in front of Python set literals. While in most cases this does nothing, in the case of the empty set it lets Coconut know that it is an empty set and not an empty dictionary. Set literals also support unpacking syntax (e.g. `s{*xs}`).

Additionally, Coconut also supports replacing the `s` with an `f` to generate a `frozenset` or an `m` to generate a Coconut *`multiset`*.

Example

Coconut:

```
empty_frozen_set = f{}
```

Python:

```
empty_frozen_set = frozenset()
```

3.7.11 Imaginary Literals

In addition to Python's `<num>j` or `<num>J` notation for imaginary literals, Coconut also supports `<num>i` or `<num>I`, to make imaginary literals more readable if used in a mathematical context.

Python Docs

Imaginary literals are described by the following lexical definitions:

```
imagnumber ::= (floatnumber | intpart) ("j" | "J" | "i" | "I")
```

An imaginary literal yields a complex number with a real part of 0.0. Complex numbers are represented as a pair of floating point numbers and have the same restrictions on their range. To create a complex number with a nonzero real part, add a floating point number to it, e.g., `(3+4i)`. Some examples of imaginary literals:

```
3.14i   10.i    10i     .001i   1e100i  3.14e-10i
```

Example

Coconut:

```
3 + 4i |> abs |> print
```

Python:

```
print(abs(3 + 4j))
```

3.7.12 Alternative Ternary Operator

Python supports the ternary operator syntax

```
result = if_true if condition else if_false
```

which, since Coconut is a superset of Python, Coconut also supports.

However, Coconut also provides an alternative syntax that uses the more conventional argument ordering as

```
result = if condition then if_true else if_false
```

making use of the Coconut-specific `then` keyword (*though Coconut still allows `then` as a variable name*).

Example

Coconut:

```
value = (
    if should_use_a() then a
    else if should_use_b() then b
    else if should_use_c() then c
    else fallback
)
```

Python:

```
value = (
    a if should_use_a() else
    b if should_use_b() else
    c if should_use_c() else
    fallback
)
```

3.8 Function Definition

3.8.1 Tail Call Optimization

Coconut will perform automatic tail call optimization and tail recursion elimination on any function that meets the following criteria:

1. it must directly return (using either `return` or *assignment function notation*) a call to itself (tail recursion elimination, the most powerful optimization) or another function (tail call optimization),
2. it must not be a generator (uses `yield`) or an asynchronous function (uses `async`).

Tail call optimization (though not tail recursion elimination) will work even for 1) mutual recursion and 2) pattern-matching functions split across multiple definitions using `addpattern`.

Example

Coconut:

```
# unlike in Python, this function will never hit a maximum recursion depth error
def factorial(n, acc=1):
    match n:
        case 0:
            return acc
        case int() if n > 0:
            return factorial(n-1, acc*n)
```

Showcases tail recursion elimination.

```
# unlike in Python, neither of these functions will ever hit a maximum recursion depth␣
↪error
def is_even(0) = True
addpattern def is_even(n `isinstance` int if n > 0) = is_odd(n-1)

def is_odd(0) = False
addpattern def is_odd(n `isinstance` int if n > 0) = is_even(n-1)
```

Showcases tail call optimization.

Python: *Can't be done without rewriting the function(s).*

`--no-tco` flag

Tail call optimization will be turned off if you pass the `--no-tco` command-line option, which is useful if you are having trouble reading your tracebacks and/or need maximum performance.

`--no-tco` does not disable tail recursion elimination. This is because tail recursion elimination is usually faster than doing nothing, while other types of tail call optimization are usually slower than doing nothing. Tail recursion elimination results in a big performance win because Python has a fairly large function call overhead. By unwinding a recursive function, far fewer function calls need to be made. When the `--no-tco` flag is disabled, Coconut will attempt to do all types of tail call optimizations, handling non-recursive tail calls, split pattern-matching functions, mutual recursion, and tail recursion. When the `--no-tco` flag is enabled, Coconut will no longer perform any tail call optimizations other than tail recursion elimination.

Tail Recursion Elimination and Python lambdas

Coconut does not perform tail recursion elimination in functions that utilize lambdas or inner functions. This is because of the way that Python handles lambdas.

Each lambda stores a pointer to the namespace enclosing it, rather than a copy of the namespace. Thus, if the Coconut compiler tries to recycle anything in the namespace that produced the lambda, which needs to be done for TRE, the lambda can be changed retroactively.

A simple example demonstrating this behavior in Python:

```
x = 1
foo = lambda: x
print(foo())  # 1
x = 2         # Directly alter the values in the namespace enclosing foo
print(foo())  # 2 (!)
```

Because this could have unintended and potentially damaging consequences, Coconut opts to not perform TRE on any function with a lambda or inner function.

3.8.2 Assignment Functions

Coconut allows for assignment function definition that automatically returns the last line of the function body. An assignment function is constructed by substituting `=` for `:` after the function definition line. Thus, the syntax for assignment function definition is either

```
[async] def <name>(<args>) = <expr>
```

for one-liners or

```
[async] def <name>(<args>) =
    <stmts>
    <expr>
```

for full functions, where `<name>` is the name of the function, `<args>` are the functions arguments, `<stmts>` are any statements that the function should execute, and `<expr>` is the value that the function should return.

Note: Assignment function definition can be combined with infix and/or pattern-matching function definition.

Rationale

Coconut's Assignment function definition is as easy to write as assignment to a lambda, but will appear named in tracebacks, as it compiles to normal Python function definition.

Example

Coconut:

```
def binexp(x) = 2**x
5 |> binexp |> print
```

Python:

```
def binexp(x): return 2**x
print(binexp(5))
```

3.8.3 Pattern-Matching Functions

Coconut pattern-matching functions are just normal functions, except where the arguments are patterns to be matched against instead of variables to be assigned to. The syntax for pattern-matching function definition is

```
[match] def <name>(<arg>, <arg>, ... [if <cond>]) [-> <return_type>]:
    <body>
```

where `<arg>` is defined as

```
[*|**] <pattern> [= <default>]
```

where `<name>` is the name of the function, `<cond>` is an optional additional check, `<body>` is the body of the function, `<pattern>` is defined by Coconut's *match statement*, `<default>` is the optional default if no argument is passed, and `<return_type>` is the optional return type annotation (note that argument type annotations are not supported for pattern-matching functions). The `match` keyword at the beginning is optional, but is sometimes necessary to disambiguate pattern-matching function definition from normal function definition, since Python function definition will always take precedence. Note that the `async` and `match` keywords can be in any order.

If `<pattern>` has a variable name (via any variable binding that binds the entire pattern, e.g. `x` in `int(x)` or `[a, b] as x`), the resulting pattern-matching function will support keyword arguments using that variable name.

In addition to supporting pattern-matching in their arguments, pattern-matching function definitions also have a couple of notable differences compared to Python functions. Specifically:

- If pattern-matching function definition fails, it will raise a *`MatchError`* (just like *destructuring assignment*) instead of a `TypeError`.
- All defaults in pattern-matching function definition are late-bound rather than early-bound. Thus, `match def f(xs=[]) = xs` will instantiate a new list for each call where `xs` is not given, unlike `def f(xs=[]) = xs`, which will use the same list for all calls where `xs` is unspecified. This also allows defaults for later arguments to be specified in terms of matched values from earlier arguments, as in `match def f(x, y=x) = (x, y)`.

Pattern-matching function definition can also be combined with `async` functions, *`copyclosure` functions*, *`yield` functions*, *infix function definition*, and *assignment function syntax*. The various keywords in front of the `def` can be put in any order.

Example

Coconut:

```
def last_two(_ + [a, b]):
    return a, b
def xydict_to_xytuple({"x": x `isinstance` int, "y": y `isinstance` int}):
    return x, y

range(5) |> last_two |> print
{"x":1, "y":2} |> xydict_to_xytuple |> print
```

Python: *Can't be done without a long series of checks at the top of the function. See the compiled code for the Python syntax.*

3.8.4 case Functions

For easily defining a pattern-matching function with many different cases, Coconut provides the `case def` syntax based on Coconut's *`case`* syntax. The basic syntax is

```
case def <name>:
    case(<arg>, <arg>, ... [if <cond>]):
        <body>
    case(<arg>, <arg>, ... [if <cond>]):
        <body>
    ...
```

where the patterns in each `case` are checked in sequence until a match is found and the body under that match is executed, or a *`MatchError`* is raised. Each `case(...)` statement is effectively treated as a separate pattern-matching function signature that is checked independently, as if they had each been defined separately and then combined with *`addpattern`*.

Any individual body can also be defined with *assignment function syntax* such that

```
case def <name>:
    case(<arg>, <arg>, ... [if <cond>]) = <body>
```

is equivalent to

```
case def <name>:
    case(<arg>, <arg>, ... [if <cond>]): return <body>
```

`case` function definition can also be combined with `async` functions, *`copyclosure` functions*, and *`yield` functions*. The various keywords in front of the `def` can be put in any order.

`case def` also allows for easily providing type annotations for pattern-matching functions. To add type annotations, inside the body of the `case def`, instead of just `case(...)` statements, include some `type(...)` statements as well, which will compile into `typing.overload` declarations. The syntax is

```
case def <name>[<type vars>]:
    type(<arg>: <type>, <arg>: <type>, ...) -> <type>
    type(<arg>: <type>, <arg>: <type>, ...) -> <type>
    ...
```

which can be interspersed with the `case(...)` statements.

Example

Coconut:

```
case def my_min[T]:
    type(x: T, y: T) -> T
    case(x, y if x <= y) = x
    case(x, y) = y

    type(xs: T[]) -> T
    case([x]) = x
    case([x] + xs) = my_min(x, my_min(xs))
```

Python: *Can't be done without a long series of checks for each pattern-matching. See the compiled code for the Python syntax.*

3.8.5 `addpattern` Functions

Coconut provides the `addpattern def` syntax as a shortcut for the full

```
@addpattern(func)
match def func(...):
  ...
```

syntax using the *`addpattern`* decorator.

If you want to put a decorator on an `addpattern def` function, make sure to put it on the *last* pattern function.

For complex multi-pattern functions, it is generally recommended to use *`case def`* over `addpattern def` in most situations.

Deprecated: `addpattern def` *will act just like a normal* `match def` *if the function has not previously been defined. This will show a* `CoconutWarning` *and is not recommended.*

Example

Coconut:

```
addpattern def factorial(0) = 1
addpattern def factorial(n) = n * factorial(n - 1)
```

Python: *Can't be done without a complicated decorator definition and a long series of checks for each pattern-matching. See the compiled code for the Python syntax.*

3.8.6 `copyclosure` Functions

Coconut supports the syntax

```
copyclosure def <name>(<args>):
    <body>
```

to define a function that uses as its closure a shallow copy of its enclosing scopes at the time that the function is defined, rather than a reference to those scopes (as with normal Python functions).

For example,`in

```
def outer_func():
    funcs = []
    for x in range(10):
        copyclosure def inner_func():
            return x
        funcs.append(inner_func)
    return funcs
```

the resulting `inner_funcs` will each return a *different* `x` value rather than all the same `x` value, since they look at what `x` was bound to at function definition time rather than during function execution.

`copyclosure` functions can also be combined with `async` functions, *`yield` functions*, *pattern-matching functions*, *infix function definition*, and *assignment function syntax*. The various keywords in front of the `def` can be put in any order.

If `global` or `nonlocal` are used in a `copyclosure` function, they will not be able to modify variables in enclosing scopes. However, they will allow state to be preserved accross multiple calls to the `copyclosure` function.

Example

Coconut:

```
def outer_func():
    funcs = []
    for x in range(10):
        copyclosure def inner_func():
            return x
        funcs.append(inner_func)
    return funcs
```

Python:

```
from functools import partial

def outer_func():
    funcs = []
    for x in range(10):
        def inner_func(_x):
            return _x
        funcs.append(partial(inner_func, x))
    return funcs
```

3.8.7 Explicit Generators

Coconut supports the syntax

```
yield def <name>(<args>):
    <body>
```

to denote that you are explicitly defining a generator function. This is useful to ensure that, even if all the `yield`s in your function are removed, it'll always be a generator function.

Explicit generator functions can also be combined with `async` functions, *`copyclosure` functions*, *pattern-matching functions*, *infix function definition*, and *assignment function syntax* (though note that assignment function syntax here creates a generator return). The various keywords in front of the `def` can be put in any order.

Example

Coconut:

```
yield def empty_it(): pass
```

Python:

```
def empty_it():
    if False:
        yield
```

3.8.8 Dotted Function Definition

Coconut allows for function definition using a dotted name to assign a function as a method of an object as specified in PEP 542. Dotted function definition can be combined with all other types of function definition above.

Example

Coconut:

```
def MyClass.my_method(self):
    ...
```

Python:

```
def my_method(self):
    ...
MyClass.my_method = my_method
```

3.9 Statements

3.9.1 Destructuring Assignment

Coconut supports significantly enhanced destructuring assignment, similar to Python's tuple/list destructuring, but much more powerful. The syntax for Coconut's destructuring assignment is

```
[match] <pattern> = <value>
```

where `<value>` is any expression and `<pattern>` is defined by Coconut's *`match` statement*. The `match` keyword at the beginning is optional, but is sometimes necessary to disambiguate destructuring assignment from normal assignment, which will always take precedence. Coconut's destructuring assignment is equivalent to a match statement that follows the syntax:

```
match <pattern> in <value>:
    pass
else:
    err = MatchError(<error message>)
    err.pattern = "<pattern>"
    err.value = <value>
    raise err
```

If a destructuring assignment statement fails, then instead of continuing on as if a `match` block had failed, a *`MatchError`* object will be raised describing the failure.

Example

Coconut:

```
_ + [a, b] = [0, 1, 2, 3]
print(a, b)
```

Python: *Can't be done without a long series of checks in place of the destructuring assignment statement. See the compiled code for the Python syntax.*

3.9.2 Type Parameter Syntax

Coconut fully supports Python 3.12 PEP 695 type parameter syntax on all Python versions.

That includes type parameters for classes, *`data` types*, and *all types of function definition*. For different types of function definition, the type parameters always come in brackets right after the function name. Coconut's *enhanced type annotation syntax* is supported for all type parameter bounds.

Warning: until `mypy` adds support for `infer_variance=True` in `TypeVar`, `TypeVar`s created this way will always be invariant.

Additionally, Coconut supports the alternative bounds syntax of `type NewType[T <: bound] = ...` rather than `type NewType[T: bound] = ...`, to make it more clear that it is an upper bound rather than a type. In `--strict` mode, `<:` is required over `:` for all type parameter bounds. *Deprecated: `<=` can also be used as an alternative to `<:`.*

Note that the `<:` syntax should only be used for type bounds, not type constraints—for type constraints, Coconut style prefers the vanilla Python `:` syntax, which helps to disambiguate between the two cases, as they are functionally different but otherwise hard to tell apart at a glance. This is enforced in `--strict` mode.

Note that, by default, all type declarations are wrapped in strings to enable forward references and improve runtime performance. If you don't want that—e.g. because you want to use type annotations at runtime—simply pass the `--no-wrap-types` flag.

PEP 695 Docs

Defining a generic class prior to this PEP looks something like this.

```
from typing import Generic, TypeVar

_T_co = TypeVar("_T_co", covariant=True, bound=str)

class ClassA(Generic[_T_co]):
    def method1(self) -> _T_co:
        ...
```

With the new syntax, it looks like this.

```
class ClassA[T: str]:
    def method1(self) -> T:
        ...
```

Here is an example of a generic function today.

```
from typing import TypeVar

_T = TypeVar("_T")

def func(a: _T, b: _T) -> _T:
    ...
```

And the new syntax.

```
def func[T](a: T, b: T) -> T:
    ...
```

Here is an example of a generic type alias today.

```
from typing import TypeAlias

_T = TypeVar("_T")

ListOrSet: TypeAlias = list[_T] | set[_T]
```

And with the new syntax.

```
type ListOrSet[T] = list[T] | set[T]
```

Example

Coconut:

```
data D[T](x: T, y: T)

def my_ident[T](x: T) -> T = x
```

Python: *Can't be done without a complex definition for the data type. See the compiled code for the Python syntax.*

3.9.3 Implicit `pass`

Coconut supports the simple `class name(base)` and `data name(args)` as aliases for `class name(base): pass` and `data name(args): pass`.

Example

Coconut:

```
class Tree
data Empty from Tree
data Leaf(item) from Tree
data Node(left, right) from Tree
```

Python: *Can't be done without a series of method definitions for each data type. See the compiled code for the Python syntax.*

3.9.4 Statement Nesting

Coconut supports the nesting of compound statements on the same line. This allows the mixing of `match` and `if` statements together, as well as compound `try` statements.

Example

Coconut:

```
if invalid(input_list):
    raise Exception()
else: match [head] + tail in input_list:
    print(head, tail)
else:
    print(input_list)
```

Python:

```
from collections.abc import Sequence
if invalid(input_list):
    raise Exception()
elif isinstance(input_list, Sequence) and len(input_list) >= 1:
    head, tail = inputlist[0], inputlist[1:]
    print(head, tail)
else:
    print(input_list)
```

3.9.5 `except` Statements

Python 3 requires that if multiple exceptions are to be caught, they must be placed inside of parentheses, so as to disallow Python 2's use of a comma instead of `as`. Coconut allows commas in except statements to translate to catching multiple exceptions without the need for parentheses, since, as in Python 3, `as` is always required to bind the exception to a name.

Example

Coconut:

```
try:
    unsafe_func(arg)
except SyntaxError, ValueError as err:
    handle(err)
```

Python:

```
try:
    unsafe_func(arg)
except (SyntaxError, ValueError) as err:
    handle(err)
```

3.9.6 In-line `global` And `nonlocal` Assignment

Coconut allows for `global` or `nonlocal` to precede assignment to a list of variables or (augmented) assignment to a variable to make that assignment `global` or `nonlocal`, respectively.

Example

Coconut:

```
global state_a, state_b = 10, 100
global state_c += 1
```

Python:

```
global state_a, state_b; state_a, state_b = 10, 100
global state_c; state_c += 1
```

3.9.7 Code Passthrough

Coconut supports the ability to pass arbitrary code through the compiler without being touched, for compatibility with other variants of Python, such as Cython or Mython. When using Coconut to compile to another variant of Python, make sure you *name your source file properly* to ensure the resulting compiled code has the right file extension for the intended usage.

Anything placed between `\(` and the corresponding close parenthesis will be passed through, as well as any line starting with `\\`, which will have the additional effect of allowing indentation under it.

Example

Coconut:

```
\\cdef f(x):
    return x |> g
```

Python:

```
cdef f(x):
    return g(x)
```

3.9.8 Enhanced Parenthetical Continuation

Since Coconut syntax is a superset of the latest Python 3 syntax, Coconut supports the same line continuation syntax as Python. That means both backslash line continuation and implied line continuation inside of parentheses, brackets, or braces will all work.

In Python, however, there are some cases (such as multiple `with` statements) where only backslash continuation, and not parenthetical continuation, is supported. Coconut adds support for parenthetical continuation in all these cases. This also includes support as per PEP 679 for parenthesized `assert` statements.

Supporting parenthetical continuation everywhere allows the PEP 8 convention, which avoids backslash continuation in favor of implied parenthetical continuation, to always be possible to follow. From PEP 8:

> The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation.

Note: Passing `--strict` will enforce the PEP 8 convention by disallowing backslash continuations.

Example

Coconut:

```
with (open('/path/to/some/file/you/want/to/read') as file_1,
      open('/path/to/some/file/being/written', 'w') as file_2):
    file_2.write(file_1.read())
```

Python:

```
# split into two with statements for Python 2.6 compatibility
with open('/path/to/some/file/you/want/to/read') as file_1:
    with open('/path/to/some/file/being/written', 'w') as file_2:
        file_2.write(file_1.read())
```

3.9.9 Assignment Expression Chaining

Unlike Python, Coconut allows assignment expressions to be chained, as in `a := b := c`. Note, however, that assignment expressions in general are currently only supported on `--target 3.8` or higher.

Example

Coconut:

```
(a := b := 1)
```

Python:

```
(a := (b := 1))
```

3.10 Built-Ins

3.10.1 Built-In Function Decorators

addpattern

addpattern(*base_func*, **add_funcs*, *allow_any_func*=False)

Takes one argument that is a *pattern-matching function*, and returns a decorator that adds the patterns in the existing function to the new function being decorated, where the existing patterns are checked first, then the new. addpattern also supports a shortcut syntax where the new patterns can be passed in directly.

Roughly equivalent to:

```
def _pattern_adder(base_func, add_func):
    def add_pattern_func(*args, **kwargs):
        try:
            return base_func(*args, **kwargs)
        except MatchError:
            return add_func(*args, **kwargs)
    return add_pattern_func
def addpattern(base_func, *add_funcs, allow_any_func=False):
    """Decorator to add a new case to a pattern-matching function (where the new case is␣
↪checked last).

    Pass allow_any_func=True to allow any object as the base_func rather than just␣
↪pattern-matching functions.
    If add_func is passed, addpattern(base_func, add_func) is equivalent to␣
↪addpattern(base_func)(add_func).
    """
    if not add_funcs:
        return addpattern$(base_func)
    for add_func in add_funcs:
        base_func = pattern_adder(base_func, add_func)
    return base_func
```

If you want to give an addpattern function a docstring, make sure to put it on the *last* function.

Note that the function taken by addpattern must be a pattern-matching function. If addpattern receives a non pattern-matching function, the function with not raise MatchError, and addpattern won't be able to detect the failed match. Thus, if a later function was meant to be called, `addpattern` will not know that the first match failed and the correct path will never be reached.

For example, the following code raises a `TypeError`:

```
def print_type():
    print("Received no arguments.")

@addpattern(print_type)
def print_type(int()):
    print("Received an int.")

print_type()  # appears to work
print_type(1) # TypeError: print_type() takes 0 positional arguments but 1 was given
```

This can be fixed by using either the `match` keyword. For example:

```
match def print_type():
    print("Received no arguments.")

addpattern def print_type(int()):
    print("Received an int.")

print_type(1)  # Works as expected
print_type("This is a string.") # Raises MatchError
```

The last case in an `addpattern` function, however, doesn't have to be a pattern-matching function if it is intended to catch all remaining cases.

To catch this mistake, `addpattern` will emit a warning if passed what it believes to be a non-pattern-matching function. However, this warning can sometimes be erroneous if the original pattern-matching function has been wrapped in some way, in which case you can pass `allow_any_func=True` to dismiss the warning.

Example

Coconut:

```
def factorial(0) = 1

@addpattern(factorial)
def factorial(n) = n * factorial(n - 1)
```

Simple example of adding a new pattern to a pattern-matching function.

```
"[A], [B]" |> windowsof$(3) |> map$(addpattern(
    (def (("[","A","]")) => "A"),
    (def (("[","B","]")) => "B"),
    (def ((_,_,_)) => None),
)) |> filter$((.is None) ..> (not)) |> list |> print
```

An example of a case where using the `addpattern` function is necessary over the `addpattern` keyword due to the use of in-line pattern-matching statement lambdas.

Python: *Can't be done without a complicated decorator definition and a long series of checks for each pattern-matching. See the compiled code for the Python syntax.*

`prepattern`

DEPRECATED: Coconut also has a `prepattern` built-in, which adds patterns in the opposite order of `addpattern`; `prepattern` is defined as:

```
def prepattern(base_func):
    """Decorator to add a new case to a pattern-matching function,
    where the new case is checked first."""
    def pattern_prepender(func):
        return addpattern(func)(base_func)
    return pattern_prepender
```

Note: Passing `--strict` *disables deprecated features.*

`memoize`

memoize(*maxsize*=`None`, *typed*=`False`)

memoize(*user_function*)

Coconut provides `functools.lru_cache` as a built-in under the name `memoize` with the modification that the *maxsize* parameter is set to `None` by default. `memoize` makes the use case of optimizing recursive functions easier, as a *maxsize* of `None` is usually what is desired in that case.

Use of `memoize` requires `functools.lru_cache`, which exists in the Python 3 standard library, but under Python 2 will require `pip install backports.functools_lru_cache` to function. Additionally, if on Python 2 and `backports.functools_lru_cache` is present, Coconut will patch `functools` such that `functools.lru_cache = backports.functools_lru_cache.lru_cache`.

Note that, if the function to be memoized is a generator or otherwise returns an iterator, *`recursive_generator`* can also be used to achieve a similar effect, the use of which is required for recursive generators.

Python Docs

@**memoize**(*user_function*)

@**memoize**(*maxsize=None, typed=False*)

Decorator to wrap a function with a memoizing callable that saves up to the *maxsize* most recent calls. It can save time when an expensive or I/O bound function is periodically called with the same arguments.

Since a dictionary is used to cache results, the positional and keyword arguments to the function must be hashable.

Distinct argument patterns may be considered to be distinct calls with separate cache entries. For example, `f(a=1, b=2)` and `f(b=2, a=1)` differ in their keyword argument order and may have two separate cache entries.

If *user_function* is specified, it must be a callable. This allows the *memoize* decorator to be applied directly to a user function, leaving the maxsize at its default value of `None`:

```
@memoize
def count_vowels(sentence):
    return sum(sentence.count(vowel) for vowel in 'AEIOUaeiou')
```

If *maxsize* is set to `None`, the LRU feature is disabled and the cache can grow without bound.

If *typed* is set to true, function arguments of different types will be cached separately. If typed is false, the implementation will usually regard them as equivalent calls and only cache a single result. (Some types such as str and int may be cached separately even when typed is false.)

Note, type specificity applies only to the function's immediate arguments rather than their contents. The scalar arguments, `Decimal(42)` and `Fraction(42)` are be treated as distinct calls with distinct results. In contrast, the tuple arguments `('answer', Decimal(42))` and `('answer', Fraction(42))` are treated as equivalent.

The decorator also provides a `cache_clear()` function for clearing or invalidating the cache.

The original underlying function is accessible through the `__wrapped__` attribute. This is useful for introspection, for bypassing the cache, or for rewrapping the function with a different cache.

The cache keeps references to the arguments and return values until they age out of the cache or until the cache is cleared.

If a method is cached, the `self` instance argument is included in the cache. See How do I cache method calls?

An LRU (least recently used) cache works best when the most recent calls are the best predictors of upcoming calls (for example, the most popular articles on a news server tend to change each day). The cache's size limit assures that the cache does not grow without bound on long-running processes such as web servers.

In general, the LRU cache should only be used when you want to reuse previously computed values. Accordingly, it doesn't make sense to cache functions with side-effects, functions that need to create distinct mutable objects on each call, or impure functions such as time() or random().

Example of efficiently computing Fibonacci numbers using a cache to implement a dynamic programming technique:

```
@memoize
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

Example

Coconut:

```
def fib(n if n < 2) = n

@memoize
@addpattern(fib)
def fib(n) = fib(n-1) + fib(n-2)
```

Python:

```
try:
    from functools import lru_cache
except ImportError:
    from backports.functools_lru_cache import lru_cache
```

(continues on next page)

(continued from previous page)

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

override

override(*func*)

Coconut provides the `@override` decorator to allow declaring a method definition in a subclass as an override of some parent class method. When `@override` is used on a method, if a method of the same name does not exist on some parent class, the class definition will raise a `RuntimeError`. `@override` works with other decorators such as `@classmethod` and `@staticmethod`, but only if `@override` is the outer-most decorator.

Additionally, `override` will present to type checkers as `typing_extensions.override`.

Example

Coconut:

```
class A:
    x = 1
    def f(self, y) = self.x + y

class B:
    @override
    def f(self, y) = self.x + y + 1
```

Python: *Can't be done without a long decorator definition. The full definition of the decorator in Python can be found in the Coconut header.*

recursive_generator

recursive_generator(*func*)

Coconut provides a `recursive_generator` decorator that memoizes and makes *reiterable* any generator or other stateless function that returns an iterator. To use `recursive_generator` on a function, it must meet the following criteria:

1. your function either always `return`s an iterator or generates an iterator using `yield`,
2. when called multiple times with arguments that are equal, your function produces the same iterator (your function is stateless), and
3. your function gets called (usually calls itself) multiple times with the same arguments.

Importantly, `recursive_generator` also allows the resolution of a nasty segmentation fault in Python's iterator logic that has never been fixed. Specifically, instead of writing

```
seq = get_elem() :: seq
```

which will crash due to the aforementioned Python issue, write

```
@recursive_generator
def seq() = get_elem() :: seq()
```

which will work just fine.

One pitfall to keep in mind working with `recursive_generator` is that it shouldn't be used in contexts where the function can potentially be called multiple times with the same iterator object as an input, but with that object not actually corresponding to the same items (e.g. because the first time the object hasn't been iterated over yet and the second time it has been).

Deprecated: `recursive_iterator` *is available as a deprecated alias for* `recursive_generator`*. Note that deprecated features are disabled in* `--strict` *mode.*

Example

Coconut:

```
@recursive_generator
def fib() = (1, 1) :: map((+), fib(), fib()$[1:])
```

Python: *Can't be done without a long decorator definition. The full definition of the decorator in Python can be found in the Coconut header.*

3.10.2 Built-In Types

`multiset`

multiset(*iterable*=None, /, **kwds)

Coconut provides `multiset` as a built-in subclass of `collections.Counter` that additionally implements the full Set and MutableSet interfaces.

For easily constructing multisets, Coconut also provides *multiset literals*.

The new methods provided by `multiset` on top of `collections.Counter` are:

- multiset.**add**(*item*): Add an element to a multiset.
- multiset.**discard**(*item*): Remove an element from a multiset if it is a member.
- multiset.**remove**(*item*): Remove an element from a multiset; it must be a member.
- multiset.**isdisjoint**(*other*): Return True if two multisets have a null intersection.
- multiset.**__xor__**(*other*): Return the symmetric difference of two multisets as a new multiset. Specifically: `a ^ b = (a - b) | (b - a)`
- multiset.**count**(*item*): Return the number of times an element occurs in a multiset. Equivalent to `multiset[item]`, but additionally verifies the count is non-negative.

- multiset.**__fmap__**(*func*): Apply a function to the contents of the multiset, preserving counts; magic method for *fmap*.

Coconut also ensures that `multiset` supports rich comparisons and `Counter.total()` on all Python versions.

Example

Coconut:

```
my_multiset = m{1, 1, 2}
my_multiset.add(3)
my_multiset.remove(2)
print(my_multiset)
```

Python:

```
from collections import Counter
my_counter = Counter((1, 1, 2))
my_counter[3] += 1
my_counter[2] -= 1
if my_counter[2] <= 0:
    del my_counter[2]
print(my_counter)
```

`Expected`

Expected(*result*=None, *error*=None)

Coconut's `Expected` built-in is a Coconut *data type* that represents a value that may or may not be an error, similar to Haskell's `Either`.

`Expected` is effectively equivalent to the following:

```
data Expected[T](result: T? = None, error: BaseException? = None):
    def __bool__(self) -> bool:
        return self.error is None
    def __fmap__[U](self, func: T -> U) -> Expected[U]:
        """Maps func over the result if it exists.

        __fmap__ should be used directly only when fmap is not available (e.g. when␣
↪consuming an Expected in vanilla Python).
        """
        return self.__class__(func(self.result)) if self else self
    def and_then[U](self, func: T -> Expected[U]) -> Expected[U]:
        """Maps a T -> Expected[U] over an Expected[T] to produce an Expected[U].
        Implements a monadic bind. Equivalent to fmap ..> .join()."""
        return self |> fmap$(func) |> .join()
    def join(self: Expected[Expected[T]]) -> Expected[T]:
        """Monadic join. Converts Expected[Expected[T]] to Expected[T]."""
        if not self:
            return self
        if not self.result `isinstance` Expected:
            raise TypeError("Expected.join() requires an Expected[Expected[_]]")
```

(continues on next page)

(continued from previous page)

```
        return self.result
    def map_error(self, func: BaseException -> BaseException) -> Expected[T]:
        """Maps func over the error if it exists."""
        return self if self else self.__class__(error=func(self.error))
    def handle(self, err_type, handler: BaseException -> T) -> Expected[T]:
        """Recover from the given err_type by calling handler on the error to determine␣
↪the result."""
        if not self and isinstance(self.error, err_type):
            return self.__class__(handler(self.error))
        return self
    def expect_error(self, *err_types: BaseException) -> Expected[T]:
        """Raise any errors that do not match the given error types."""
        if not self and not isinstance(self.error, err_types):
            raise self.error
        return self
    def unwrap(self) -> T:
        """Unwrap the result or raise the error."""
        if not self:
            raise self.error
        return self.result
    def or_else[U](self, func: BaseException -> Expected[U]) -> Expected[T | U]:
        """Return self if no error, otherwise return the result of evaluating func on␣
↪the error."""
        return self if self else func(self.error)
    def result_or_else[U](self, func: BaseException -> U) -> T | U:
        """Return the result if it exists, otherwise return the result of evaluating␣
↪func on the error."""
        return self.result if self else func(self.error)
    def result_or[U](self, default: U) -> T | U:
        """Return the result if it exists, otherwise return the default.

        Since .result_or() completely silences errors, it is highly recommended that you
        call .expect_error() first to explicitly declare what errors you are okay␣
↪silencing.
        """
        return self.result if self else default
```

`Expected` is primarily used as the return type for *safe_call*.

Generally, the best way to use `Expected` is with *fmap*, which will apply a function to the result if it exists, or otherwise retain the error. If you want to sequence multiple `Expected`-returning operations, `.and_then` should be used instead of `fmap`. To handle specific errors, the following patterns are equivalent:

```
safe_call(might_raise_IOError).handle(IOError, const 10).unwrap()
safe_call(might_raise_IOError).expect_error(IOError).result_or(10)
```

To match against an `Expected`, just:

```
Expected(res) = Expected("result")
Expected(error=err) = Expected(error=TypeError())
```

Example

Coconut:

```
def try_divide(x: float, y: float) -> Expected[float]:
    try:
        return Expected(x / y)
    except Exception as err:
        return Expected(error=err)

try_divide(1, 2) |> fmap$(.+1) |> print
try_divide(1, 0) |> fmap$(.+1) |> print
```

Python: *Can't be done without a complex `Expected` definition. See the compiled code for the Python syntax.*

`MatchError`

A `MatchError` is raised when a *destructuring assignment* or *pattern-matching function* fails, and thus `MatchError` is provided as a built-in for catching those errors. `MatchError` objects support three attributes: `pattern`, which is a string describing the failed pattern; `value`, which is the object that failed to match that pattern; and `message` which is the full error message. To avoid unnecessary `repr` calls, `MatchError` only computes the `message` once it is actually requested.

Additionally, if you are using *view patterns*, you might need to raise your own `MatchError` (though you can also just use a destructuring assignment or pattern-matching function definition to do so). To raise your own `MatchError`, just `raise MatchError(pattern, value)` (both arguments are optional).

In some cases where there are multiple Coconut packages installed at the same time, there may be multiple `MatchError`s defined in different packages. Coconut can perform some magic under the hood to make sure that all these `MatchError`s will seamlessly interoperate, but only if all such packages are compiled in *`--package` mode rather than `--standalone` mode*.

3.10.3 `CoconutWarning`

`CoconutWarning` is the `Warning` subclass used for all runtime Coconut warnings; see `warnings`.

3.10.4 Generic Built-In Functions

- *`makedata`*
- *`fmap`*
- *`call`*
- *`safe_call`*
- *`ident`*
- *`const`*
- *`flip`*
- *`lift` and `lift_apart`*
- *`and_then` and `and_then_await`*

makedata

makedata(*data_type*, **args*)

Coconut provides the makedata function to construct a container given the desired type and contents. This is particularly useful when writing alternative constructors for *data* types by overwriting __new__, since it allows direct access to the base constructor of the data type created with the Coconut data statement.

makedata takes the data type to construct as the first argument, and the objects to put in that container as the rest of the arguments.

makedata can also be used to extract the underlying constructor for *match data* types that bypasses the normal pattern-matching constructor.

Additionally, makedata can also be called with non-data type as the first argument, in which case it will do its best to construct the given type of object with the given arguments. This functionality is used internally by fmap.

datamaker

DEPRECATED: Coconut also has a datamaker built-in, which partially applies makedata; datamaker is defined as:

```
def datamaker(data_type):
    """Get the original constructor of the given data type or class."""
    return makedata$(data_type)
```

Note: Passing --strict disables deprecated features.

Example

Coconut:

```
data Tuple(elems):
    def __new__(cls, *elems):
        return elems |> makedata$(cls)
```

Python: *Can't be done without a series of method definitions for each data type. See the compiled code for the Python syntax.*

fmap

fmap(*func*, *obj*)

In Haskell, fmap(func, obj) takes a data type obj and returns a new data type with func mapped over the contents. Coconut's fmap function does the exact same thing for Coconut's *data types*.

fmap can also be used on the built-in objects str, dict, list, tuple, set, frozenset, bytes, bytearray, and dict as a variant of map that returns back an object of the same type.

For dict, or any other collections.abc.Mapping, fmap will map over the mapping's .items() instead of the default iteration through its .keys(), with the new mapping reconstructed from the mapped over items. *Deprecated: fmap$(starmap_over_mappings=True) will starmap over the .items() instead of map over them.*

For asynchronous iterables, fmap will map asynchronously, making fmap equivalent in that case to

```
async def fmap_over_async_iters(func, async_iter):
    async for item in async_iter:
        yield func(item)
```

such that `fmap` can effectively be used as an async map.

Some objects from external libraries are also given special support:

- For *numpy* objects, `fmap` will use `np.vectorize` to produce the result.
- For pandas objects, `fmap` will use `.apply` along the last axis (so row-wise for `DataFrame`'s, element-wise for `Series`'s).
- For `xarray` objects, `fmap` will first convert them into `pandas` objects, apply `fmap`, then convert them back.

The behavior of `fmap` for a given object can be overridden by defining an `__fmap__(self, func)` magic method that will be called whenever `fmap` is invoked on that object. Note that `__fmap__` implementations should always satisfy the Functor Laws.

Deprecated: `fmap(func, obj, fallback_to_init=True)` *will fall back to* `obj.__class__(map(func, obj))` *if no* `fmap` *implementation is available rather than raise* `TypeError`*.*

Example

Coconut:

```
[1, 2, 3] |> fmap$(x => x+1) == [2, 3, 4]

class Maybe
data Nothing() from Maybe
data Just(n) from Maybe

Just(3) |> fmap$(x => x*2) == Just(6)
Nothing() |> fmap$(x => x*2) == Nothing()
```

Python: *Can't be done without a series of method definitions for each data type. See the compiled code for the Python syntax.*

`call`

call(*func*, /, **args*, ***kwargs*)

Coconut's `call` simply implements function application. Thus, `call` is effectively equivalent to

```
def call(f, /, *args, **kwargs) = f(*args, **kwargs)
```

`call` is primarily useful as an *operator function* for function application when writing in a point-free style.

Deprecated: `of` *is available as a deprecated alias for* `call`*. Note that deprecated features are disabled in* `--strict` *mode.*

`safe_call`

safe_call(*func*, /, **args*, ***kwargs*)

Coconut's `safe_call` is a version of *call* that catches any `Exception`s and returns an *Expected* containing either the result or the error.

`safe_call` is effectively equivalent to:

```
def safe_call(f, /, *args, **kwargs):
    try:
        return Expected(f(*args, **kwargs))
    except Exception as err:
        return Expected(error=err)
```

To define a function that always returns an `Expected` rather than raising any errors, simply decorate it with `@safe_call$`.

Example

Coconut:

```
res, err = safe_call(=> 1 / 0) |> fmap$(.+1)
```

Python: *Can't be done without a complex* `Expected` *definition. See the compiled code for the Python syntax.*

`ident`

ident(*x*, *, *side_effect*=None)

Coconut's `ident` is the identity function, generally equivalent to `x => x`.

`ident` also accepts one keyword-only argument, `side_effect`, which specifies a function to call on the argument before it is returned. Thus, `ident` is effectively equivalent to:

```
def ident(x, *, side_effect=None):
    if side_effect is not None:
        side_effect(x)
    return x
```

`ident` is primarily useful when writing in a point-free style (e.g. in combination with *lift*) or for debugging *pipes* where `ident$(side_effect=print)` can let you see what is being piped.

`const`

const(*value*)

Coconut's `const` simply constructs a function that, whatever its arguments, just returns the given value. Thus, `const` is equivalent to a pickleable version of

```
def const(value) = (*args, **kwargs) => value
```

`const` is primarily useful when writing in a point-free style (e.g. in combination with *lift*).

`flip`

flip(*func*, *nargs*=None)

Coconut's `flip(f, nargs=None)` is a higher-order function that, given a function `f`, returns a new function with reversed argument order. If `nargs` is passed, only the first `nargs` arguments are reversed.

For the binary case, `flip` works as

```
flip(f, 2)(x, y) == f(y, x)
```

such that `flip$(?, 2)` implements the `C` combinator (`flip` in Haskell).

In the general case, `flip` is equivalent to a pickleable version of

```
def flip(f, nargs=None) =
    (*args, **kwargs) => (
        f(*args[::-1], **kwargs) if nargs is None
        else f(*(args[nargs-1::-1] + args[nargs:]), **kwargs)
    )
```

`lift` and `lift_apart`

lift(*func*)

lift(*func*, **func_args*, ***func_kwargs*)

Coconut's `lift` built-in is a higher-order function that takes in a function and "lifts" it up so that all of its arguments are functions.

As a simple example, for a binary function `f(x, y)` and two unary functions `g(z)` and `h(z)`, `lift` works as

```
lift(f)(g, h)(z) == f(g(z), h(z))
```

such that in this case `lift` implements the `S'` combinator (`liftA2` or `liftM2` in Haskell).

In the general case, `lift` is equivalent to a pickleable version of

```
def lift(f) = (
    (*func_args, **func_kwargs) =>
        (*args, **kwargs) =>
            f(
                *(g(*args, **kwargs) for g in func_args),
                **{k: h(*args, **kwargs) for k, h in func_kwargs.items()}
            )
)
```

`lift` also supports a shortcut form such that `lift(f, *func_args, **func_kwargs)` is equivalent to `lift(f)(*func_args, **func_kwargs)`.

lift_apart(*func*)

lift_apart(*func*, **func_args*, ***func_kwargs*)

Coconut's `lift_apart` built-in is very similar to `lift`, except instead of duplicating the final arguments to each function, it separates them out.

For a binary function `f(x, y)` and two unary functions `g(z)` and `h(z)`, `lift_apart` works as

```
lift_apart(f)(g, h)(z, w) == f(g(z), h(w))
```

such that in this case `lift_apart` implements the `D2` combinator.

In the general case, `lift_apart` is equivalent to a pickleable version of

```
def lift_apart(f) = (
    (*func_args, **func_kwargs) =>
        (*args, **kwargs) =>
            f(
                *(f(x) for f, x in zip(func_args, args, strict=True)),
                **{k: func_kwargs[k](kwargs[k]) for k in func_kwargs.keys() | kwargs.
↪keys()},
            )
)
```

`lift_apart` supports the same shortcut form as `lift`.

Examples

Coconut:

```
xs_and_xsp1 = ident `lift(zip)` map$(=>_+1)
min_and_max = lift(,)(min, max)
plus_and_times = (+) `lift(,)` (*)
```

Python:

```
def xs_and_xsp1(xs):
    return zip(xs, map(lambda x: x + 1, xs))
def min_and_max(xs):
    return min(xs), max(xs)
def plus_and_times(x, y):
    return x + y, x * y
```

Coconut:

```
first_false_and_last_true = (
    lift(,)(ident, reversed)
    ..*> lift_apart(,)(dropwhile$(bool), dropwhile$(not))
    ..*> lift_apart(,)(.$[0], .$[0])
)
```

Python:

```
from itertools import dropwhile

def first_false_and_last_true(xs):
    rev_xs = reversed(xs)
    return (
        next(dropwhile(bool, xs)),
        next(dropwhile(lambda x: not x, rev_xs)),
    )
```

`and_then` and `and_then_await`

and_then(*first_async_func*, *second_func*)

and_then_await(*first_async_func*, *second_async_func*)

Coconut provides the `and_then` and `and_then_await` built-ins for composing `async` functions. Specifically:

- To forwards compose an async function `async_f` with a normal function `g` (such that `g` is called on the result of `await`ing `async_f`), write ``async_f `and_then` g``.
- To forwards compose an async function `async_f` with another async function `async_g` (such that `async_g` is called on the result of `await`ing `async_f`, and then `async_g` is itself awaited), write ``async_f `and_then_await` async_g``.
- To forwards compose a normal function `f` with an async function `async_g` (such that `async_g` is called on the result of `f`), just write `f ..> async_g`.

Note that all of the above will always result in the resulting composition being an `async` function.

The built-ins are effectively equivalent to:

```
def and_then[**T, U, V](
    first_async_func: async (**T) -> U,
    second_func: U -> V,
) -> async (**T) -> V =
    async def (*args, **kwargs) => (
        first_async_func(*args, **kwargs)
        |> await
        |> second_func
    )

def and_then_await[**T, U, V](
    first_async_func: async (**T) -> U,
    second_async_func: async U -> V,
) -> async (**T) -> V =
    async def (*args, **kwargs) => (
        first_async_func(*args, **kwargs)
        |> await
        |> second_async_func
        |> await
    )
```

Like normal *function composition*, `and_then` and `and_then_await` will preserve all metadata attached to the first function in the composition.

Example

Coconut:

```
load_and_send_data = (
    load_data_async()
    `and_then` proc_data
    `and_then_await` send_data
)
```

Python:

```
async def load_and_send_data():
    return await send_data(proc_data(await load_data_async()))
```

3.10.5 Built-Ins for Working with Iterators

Enhanced Built-Ins

Coconut's `map`, `zip`, `filter`, `reversed`, and `enumerate` objects are enhanced versions of their Python equivalents that support:

- The ability to be iterated over multiple times if the underlying iterators can be iterated over multiple times.
  - *Note: This can lead to different behavior between Coconut built-ins and Python built-ins. Use* `py_` *versions if the Python behavior is necessary.*
- `reversed`
- `repr`
- Optimized normal (and iterator) indexing/slicing (`map`, `zip`, `reversed`, and `enumerate` but not `filter`).
- `len` (all but `filter`) (though `bool` will still always yield `True`).
- PEP 618 `zip(..., strict=True)` support on all Python versions.
- Added `strict=True` support to `map` as well (enforces that iterables are the same length in the multi-iterable case; uses `zip` under the hood such that errors will show up as `zip(..., strict=True)` errors).
- Added attributes which subclasses can make use of to get at the original arguments to the object:
  - `map`: `func`, `iters`
  - `zip`: `iters`
  - `filter`: `func`, `iter`
  - `reversed`: `iter`
  - `enumerate`: `iter`, `start`

Indexing into other built-ins

Though Coconut provides random access indexing/slicing to `range`, `map`, `zip`, `reversed`, and `enumerate`, Coconut cannot index into built-ins like `filter`, `takewhile`, or `dropwhile` directly, as there is no efficient way to do so.

```
range(10) |> filter$(i => i>3) |> .[0]  # doesn't work
```

In order to make this work, you can explicitly use iterator slicing, which is less efficient in the general case:

```
range(10) |> filter$(i => i>3) |> .$[0]  # works
```

For more information on Coconut's iterator slicing, see *here*.

Examples

Coconut:

```
map((+), range(5), range(6)) |> len |> print
range(10) |> filter$((x) => x < 5) |> reversed |> tuple |> print
```

Python: *Can't be done without defining a custom* `map` *type. The full definition of* `map` *can be found in the Coconut header.*

Coconut:

```
range(0, 12, 2)[4]  # 8

map((i => i*2), range(10))[2]  # 4
```

Python: *Can't be done quickly without Coconut's iterable indexing, which requires many complicated pieces. The necessary definitions in Python can be found in the Coconut header.*

`reduce`

reduce(*function*, *iterable*[, *initial*], /)

Coconut re-introduces Python 2's `reduce` built-in, using the `functools.reduce` version.

Python Docs

reduce(*function, iterable*[, *initial*])

Apply *function* of two arguments cumulatively to the items of *sequence*, from left to right, so as to reduce the sequence to a single value. For example, `reduce((x, y) => x+y, [1, 2, 3, 4, 5])` calculates `((((1+2)+3)+4)+5)`. The left argument, *x*, is the accumulated value and the right argument, *y*, is the update value from the *sequence*. If the optional *initial* is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If *initial* is not given and *sequence* contains only one item, the first item is returned.

Example

Coconut:

```
product = reduce$(*)
range(1, 10) |> product |> print
```

Python:

```
import operator
import functools
product = functools.partial(functools.reduce, operator.mul)
print(product(range(1, 10)))
```

`reiterable`

reiterable(*iterable*)

`reiterable` wraps the given iterable to ensure that every time the `reiterable` is iterated over, it produces the same results. Note that the result need not be a `reiterable` object if the given iterable is already reiterable. `reiterable` uses `tee` under the hood and `tee` can be used in its place, though `reiterable` is generally recommended over `tee`.

Example

Coconut:

```
def list_type(xs):
    match reiterable(xs):
        case [fst, snd] :: tail:
            return "at least 2"
        case [fst] :: tail:
            return "at least 1"
        case (| |):
            return "empty"
```

Python: *Can't be done without a long series of checks for each* `match` *statement. See the compiled code for the Python syntax.*

`starmap`

starmap(*function*, *iterable*)

Coconut provides a modified version of `itertools.starmap` that supports `reversed`, `repr`, optimized normal (and iterator) slicing, `len`, and `func/iter` attributes.

Python Docs

starmap(*function, iterable*)

Make an iterator that computes the function using arguments obtained from the iterable. Used instead of `map()` when argument parameters are already grouped in tuples from a single iterable (the data has been "pre-zipped"). The difference between `map()` and `starmap()` parallels the distinction between `function(a,b)` and `function(*c)`. Roughly equivalent to:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

Example

Coconut:

```
range(1, 5) |> map$(range) |> starmap$(print) |> consume
```

Python:

```
import itertools, collections
collections.deque(itertools.starmap(print, map(range, range(1, 5))), maxlen=0)
```

zip_longest

zip_longest(**iterables*, *fillvalue*=None)

Coconut provides an enhanced version of `itertools.zip_longest` as a built-in under the name `zip_longest`. `zip_longest` supports all the same features as Coconut's *enhanced zip* as well as the additional attribute `fillvalue`.

Python Docs

zip_longest(**iterables, fillvalue=None*)

Make an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with *fillvalue*. Iteration continues until the longest iterable is exhausted. Roughly equivalent to:

```
def zip_longest(*args, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
    iterators = [iter(it) for it in args]
    num_active = len(iterators)
    if not num_active:
        return
    while True:
        values = []
        for i, it in enumerate(iterators):
            try:
                value = next(it)
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
                iterators[i] = repeat(fillvalue)
                value = fillvalue
            values.append(value)
        yield tuple(values)
```

If one of the iterables is potentially infinite, then the `zip_longest()` function should be wrapped with something that limits the number of calls (for example iterator slicing or `takewhile`). If not specified, *fillvalue* defaults to `None`.

Example

Coconut:

```
result = zip_longest(range(5), range(10))
```

Python:

```
import itertools
result = itertools.zip_longest(range(5), range(10))
```

`takewhile`

takewhile(*predicate*, *iterable*, */*)

Coconut provides `itertools.takewhile` as a built-in under the name `takewhile`.

Python Docs

takewhile(*predicate, iterable*)

Make an iterator that returns elements from the *iterable* as long as the *predicate* is true. Equivalent to:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

Example

Coconut:

```
negatives = numiter |> takewhile$(x => x < 0)
```

Python:

```
import itertools
negatives = itertools.takewhile(lambda x: x < 0, numiter)
```

`dropwhile`

dropwhile(*predicate*, *iterable*, */*)

Coconut provides `itertools.dropwhile` as a built-in under the name `dropwhile`.

Python Docs

dropwhile(*predicate, iterable*)

Make an iterator that drops elements from the *iterable* as long as the *predicate* is true; afterwards, returns every element. Note: the iterator does not produce any output until the predicate first becomes false, so it may have a lengthy start-up time. Equivalent to:

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
```

(continues on next page)

(continued from previous page)

```
            break
    for x in iterable:
        yield x
```

Example

Coconut:

```
positives = numiter |> dropwhile$(x => x < 0)
```

Python:

```
import itertools
positives = itertools.dropwhile(lambda x: x < 0, numiter)
```

`flatten`

flatten(*iterable*, *levels*=1)

Coconut provides an enhanced version of `itertools.chain.from_iterable` as a built-in under the name `flatten` with added support for `reversed`, `repr`, `in`, `.count()`, `.index()`, and `fmap`.

By default, `flatten` only flattens the top level of the given iterable/array. If *levels* is passed, however, it can be used to control the number of levels flattened, with `0` meaning no flattening and `None` flattening as many iterables as are found. Note that if *levels* is set to any non-`None` value, the first *levels* levels must be iterables, or else an error will be raised.

Python Docs

chain.**from_iterable**(*iterable*)

Alternate constructor for `chain()`. Gets chained inputs from a single iterable argument that is evaluated lazily. Roughly equivalent to:

```
def flatten(iterables):
    # flatten(['ABC', 'DEF']) --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

Example

Coconut:

```
iter_of_iters = [[1, 2], [3, 4]]
flat_it = iter_of_iters |> flatten |> list
```

Python:

```
from itertools import chain
iter_of_iters = [[1, 2], [3, 4]]
flat_it = list(chain.from_iterable(iter_of_iters))
```

scan

scan(*function*, *iterable*[, *initial*])

Coconut provides a modified version of `itertools.accumulate` with opposite argument order as `scan` that also supports `repr`, `len`, and `func/iter/initial` attributes. `scan` works exactly like *reduce*, except that instead of only returning the last accumulated value, it returns an iterator of all the intermediate values.

Python Docs

scan(*function, iterable*[, *initial*])

Make an iterator that returns accumulated results of some function of two arguments. Elements of the input iterable may be any type that can be accepted as arguments to *function*. (For example, with the operation of addition, elements may be any addable type including Decimal or Fraction.) If the input iterable is empty, the output iterable will also be empty.

If no *initial* is given, roughly equivalent to:

```
def scan(function, iterable):
    'Return running totals'
    # scan(operator.add, [1,2,3,4,5]) --> 1 3 6 10 15
    # scan(operator.mul, [1,2,3,4,5]) --> 1 2 6 24 120
    it = iter(iterable)
    try:
        total = next(it)
    except StopIteration:
        return
    yield total
    for element in it:
        total = function(total, element)
        yield total
```

Example

Coconut:

```
input_data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
running_max = input_data |> scan$(max) |> list
```

Python:

```
input_data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
running_max = []
max_so_far = input_data[0]
for x in input_data:
    if x > max_so_far:
```

(continues on next page)

(continued from previous page)

```
        max_so_far = x
    running_max.append(max_so_far)
```

`count`

count(*start*=0, *step*=1)

Coconut provides a modified version of `itertools.count` that supports `in`, normal slicing, optimized iterator slicing, the standard `count` and `index` sequence methods, `repr`, and `start`/`step` attributes as a built-in under the name `count`. If the *step* parameter is set to `None`, `count` will behave like `itertools.repeat` instead.

Since `count` supports slicing, `count()[...]` can be used as a version of `range` that can in some cases be more readable. In particular, it is easy to accidentally write `range(10, 2)` when you meant `range(0, 10, 2)`, but it is hard to accidentally write `count()[10:2]` when you mean `count()[:10:2]`.

Python Docs

count(*start=0, step=1*)

Make an iterator that returns evenly spaced values starting with number *start*. Often used as an argument to `map()` to generate consecutive data points. Also, used with `zip()` to add sequence numbers. Roughly equivalent to:

```
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        if step:
          n += step
```

Example

Coconut:

```
count()$[10**100] |> print
```

Python: *Can't be done quickly without Coconut's iterator slicing, which requires many complicated pieces. The necessary definitions in Python can be found in the Coconut header.*

`cycle`

cycle(*iterable*, *times*=None)

Coconut's `cycle` is a modified version of `itertools.cycle` with a `times` parameter that controls the number of times to cycle through *iterable* before stopping. `cycle` also supports `in`, slicing, `len`, `reversed`, `.count()`, `.index()`, and `repr`.

Python Docs

cycle(*iterable*)

Make an iterator returning elements from the iterable and saving a copy of each. When the iterable is exhausted, return elements from the saved copy. Repeats indefinitely. Roughly equivalent to:

```
def cycle(iterable):
    # cycle('ABCD') --> A B C D A B C D A B C D ...
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
            yield element
```

Note, this member of the toolkit may require significant auxiliary storage (depending on the length of the iterable).

Example

Coconut:

```
cycle(range(2), 2) |> list |> print
```

Python:

```
from itertools import cycle, islice
print(list(islice(cycle(range(2)), 4)))
```

`cartesian_product`

cartesian_product(**iterables*, *repeat*=1)

Coconut provides an enhanced version of `itertools.product` as a built-in under the name `cartesian_product` with added support for `len`, `repr`, `in`, `.count()`, and `fmap`.

Additionally, `cartesian_product` includes special support for *numpy* objects, in which case a multidimensional array is returned instead of an iterator.

Python Docs

cartesian_product(**iterables, repeat=1*)

Cartesian product of input iterables.

Roughly equivalent to nested for-loops in a generator expression. For example, `cartesian_product(A, B)` returns the same as `((x,y) for x in A for y in B)`.

The nested loops cycle like an odometer with the rightmost element advancing on every iteration. This pattern creates a lexicographic ordering so that if the input's iterables are sorted, the product tuples are emitted in sorted order.

To compute the product of an iterable with itself, specify the number of repetitions with the optional repeat keyword argument. For example, `product(A, repeat=4)` means the same as `cartesian_product(A, A, A, A)`.

This function is roughly equivalent to the following code, except that the actual implementation does not build up intermediate results in memory:

```
def cartesian_product(*args, repeat=1):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
    pools = [tuple(pool) for pool in args] * repeat
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

Before `cartesian_product()` runs, it completely consumes the input iterables, keeping pools of values in memory to generate the products. Accordingly, it is only useful with finite inputs.

Example

Coconut:

```
v = [1, 2]
assert cartesian_product(v, v) |> list == [(1, 1), (1, 2), (2, 1), (2, 2)]
```

Python:

```
from itertools import product
v = [1, 2]
assert list(product(v, v)) == [(1, 1), (1, 2), (2, 1), (2, 2)]
```

`multi_enumerate`

multi_enumerate(*iterable*)

Coconut's `multi_enumerate` enumerates through an iterable of iterables. `multi_enumerate` works like enumerate, but indexes through inner iterables and produces a tuple index representing the index in each inner iterable. Supports indexing.

For *numpy* objects, uses `np.nditer` under the hood. Also supports `len` for *numpy* arrays.

Example

Coconut:

```
>>> [1, 2;; 3, 4] |> multi_enumerate |> list
[((0, 0), 1), ((0, 1), 2), ((1, 0), 3), ((1, 1), 4)]
```

Python:

```
array = [[1, 2], [3, 4]]
enumerated_array = []
for i in range(len(array)):
    for j in range(len(array[i])):
        enumerated_array.append(((i, j), array[i][j]))
```

groupsof

groupsof(*n*, *iterable*, *fillvalue*=`...`)

Coconut provides the `groupsof` built-in to split an iterable into groups of a specific length. Specifically, `groupsof(n, iterable)` will split `iterable` into tuples of length `n`, with only the last tuple potentially of size < `n` if the length of `iterable` is not divisible by `n`. If that is not the desired behavior, *fillvalue* can be passed and will be used to pad the end of the last tuple to length `n`.

Additionally, `groupsof` supports `len` when `iterable` supports `len`.

Example

Coconut:

```
pairs = range(1, 11) |> groupsof$(2)
```

Python:

```
pairs = []
group = []
for item in range(1, 11):
    group.append(item)
    if len(group) == 2:
        pairs.append(tuple(group))
        group = []
if group:
    pairs.append(tuple(group))
```

windowsof

windowsof(*size*, *iterable*, *fillvalue*=`...`, *step*=1)

`windowsof` produces an iterable that effectively mimics a sliding window over *iterable* of size *size*. *step* determines the spacing between windows.

If *size* is larger than *iterable*, `windowsof` will produce an empty iterable. If that is not the desired behavior, *fillvalue* can be passed and will be used in place of missing values. Also, if *fillvalue* is passed and the length of the *iterable* is not divisible by *step*, *fillvalue* will be used in that case to pad the last window as well. Note that *fillvalue* will only ever appear in the last window.

Additionally, `windowsof` supports `len` when `iterable` supports `len`.

Example

Coconut:

```
assert "12345" |> windowsof$(3) |> list == [("1", "2", "3"), ("2", "3", "4"), ("3", "4",
↪"5")]
```

Python: *Can't be done without the definition of* `windowsof`*; see the compiled header for the full definition.*

all_equal

all_equal(*iterable*, *to*=...)

Coconut's all_equal built-in takes in an iterable and determines whether all of its elements are equal to each other.

If *to* is passed, all_equal will check that all the elements are specifically equal to that value, rather than just equal to each other.

Note that all_equal assumes transitivity of equality, that != is the negation of ==, and that empty arrays always have all their elements equal.

Special support is provided for *numpy* objects.

Example

Coconut:

```
all_equal([1, 1, 1])
all_equal([1, 1, 2])
```

Python:

```
sentinel = object()
def all_equal(iterable):
    first_item = sentinel
    for item in iterable:
        if first_item is sentinel:
            first_item = item
        elif first_item != item:
            return False
    return True
all_equal([1, 1, 1])
all_equal([1, 1, 2])
```

tee

tee(*iterable*, *n*=2)

Coconut provides an optimized version of itertools.tee as a built-in under the name tee.

Though tee is not deprecated, *reiterable* is generally recommended over tee.

Custom tee/reiterable implementations for custom Containers/Collections should be put in the __copy__ method. Note that all Sequences/Mappings/Sets are always assumed to be reiterable even without calling __copy__.

Python Docs

tee(*iterable, n=2*)

Return *n* independent iterators from a single iterable. Equivalent to:

```
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:             # when the local deque is empty
                newval = next(it)       # fetch a new value and
                for d in deques:        # load it to all the deques
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

Once `tee()` has made a split, the original *iterable* should not be used anywhere else; otherwise, the *iterable* could get advanced without the tee objects being informed.

This itertool may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator uses most or all of the data before another iterator starts, it is faster to use `list()` instead of `tee()`.

Example

Coconut:

```
original, temp = tee(original)
sliced = temp$[5:]
```

Python:

```
import itertools
original, temp = itertools.tee(original)
sliced = itertools.islice(temp, 5, None)
```

`consume`

consume(*iterable*, *keep_last*=0)

Coconut provides the `consume` function to efficiently exhaust an iterator and thus perform any lazy evaluation contained within it. `consume` takes one optional argument, `keep_last`, that defaults to 0 and specifies how many, if any, items from the end to return as a sequence (`None` will keep all elements).

Equivalent to:

```
def consume(iterable, keep_last=0):
    """Fully exhaust iterable and return the last keep_last elements."""
    return collections.deque(iterable, maxlen=keep_last)  # fastest way to exhaust an␣
↪iterator
```

Rationale

In the process of lazily applying operations to iterators, eventually a point is reached where evaluation of the iterator is necessary. To do this efficiently, Coconut provides the `consume` function, which will fully exhaust the iterator given to it.

Example

Coconut:

```
range(10) |> map$((x) => x**2) |> map$(print) |> consume
```

Python:

```
collections.deque(map(print, map(lambda x: x**2, range(10))), maxlen=0)
```

3.10.6 Built-Ins for Parallelization

`process_map` and `thread_map`

process_map(*function*, **iterables*, *, *chunksize*=`1`, *strict*=`False`, *stream*=`False`, *ordered*=`True`)

Coconut provides a `multiprocessing`-based version of `map` under the name `process_map`. `process_map` makes use of multiple processes, and is therefore much faster than `map` for CPU-bound tasks. If any exceptions are raised inside of `process_map`, a traceback will be printed as soon as they are encountered. Results will be in the same order as the input unless *ordered*=`False`.

`process_map` never loads the entire input iterator into memory, though by default it does consume the entire input iterator as soon as a single output is requested. Results can be streamed one at a time when iterating by passing *stream*=`True`, however note that *stream*=`True` requires that the resulting iterator only be iterated over inside of a `process_map.multiple_sequential_calls` block (see below).

Because `process_map` uses multiple processes for its execution, it is necessary that all of its arguments be pickleable. Only objects defined at the module level, and not lambdas, objects defined inside of a function, or objects defined inside of the interpreter, are pickleable. Furthermore, on Windows, it is necessary that all calls to `process_map` occur inside of an `if __name__ == "__main__"` guard.

`process_map` supports a `chunksize` argument, which determines how many items are passed to each process at a time. Larger values of *chunksize* are recommended when dealing with very long iterables. Additionally, in the multi-iterable case, *strict* can be set to `True` to ensure that all iterables are the same length.

Deprecated: `parallel_map` *is available as a deprecated alias for* `process_map`*. Note that deprecated features are disabled in* `--strict` *mode.*

process_map.multiple_sequential_calls(*max_workers*=None)

If multiple sequential calls to `process_map` need to be made, it is highly recommended that they be done inside of a `with process_map.multiple_sequential_calls():` block, which will cause the different calls to use the same process pool and result in `process_map` immediately returning a list rather than a `process_map` object. If multiple sequential calls are necessary and the laziness of process_map is required, then the `process_map` objects should be constructed before the `multiple_sequential_calls` block and then only iterated over once inside the block.

`process_map.multiple_sequential_calls` also supports a *max_workers* argument to set the number of processes. If `max_workers=None`, Coconut will pick a suitable *max_workers*, including reusing worker pools from higher up in the call stack.

thread_map(*function*, **iterables*, *, *chunksize*=1, *strict*=False, *stream*=False, *ordered*=True)

thread_map.multiple_sequential_calls(*max_workers*=None)

Coconut provides a `multithreading`-based version of *`process_map`* under the name `thread_map`. `thread_map` and `thread_map.multiple_sequential_calls` behave identically to `process_map` except that they use multithreading instead of multiprocessing, and are therefore primarily useful only for IO-bound tasks due to CPython's Global Interpreter Lock.

Deprecated: `concurrent_map` is available as a deprecated alias for `thread_map`. Note that deprecated features are disabled in `--strict` mode.

Python Docs

process_map(*func, *iterables*, *chunksize*=1)

Equivalent to `map(func, *iterables)` except *func* is executed asynchronously and several calls to *func* may be made concurrently. If a call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

`process_map` chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer. For very long iterables using a large value for *chunksize* can make the job complete **much** faster than using the default value of 1.

thread_map(*func, *iterables*, *chunksize*=1)

Equivalent to `map(func, *iterables)` except *func* is executed asynchronously and several calls to *func* may be made concurrently. If a call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

`thread_map` chops the iterable into a number of chunks which it submits to the thread pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer. For very long iterables using a large value for *chunksize* can make the job complete **much** faster than using the default value of 1.

Examples

Coconut:

```
process_map(pow$(2), range(100)) |> list |> print
```

Python:

```
import functools
from multiprocessing import Pool
with Pool() as pool:
    print(list(pool.imap(functools.partial(pow, 2), range(100))))
```

Coconut:

```
thread_map(get_data_for_user, get_all_users()) |> list |> print
```

Python:

```
import functools
import concurrent.futures
with concurrent.futures.ThreadPoolExecutor() as executor:
    print(list(executor.map(get_data_for_user, get_all_users())))
```

`collectby` and `mapreduce`

collectby(*key_func*, *iterable*, *value_func*=None, *, *reduce_func*=None, *collect_in*=None, *reduce_func_init*=`...`, *map_using*=None)

`collectby(key_func, iterable)` collects the items in `iterable` into a dictionary of lists keyed by `key_func(item)`.

If *value_func* is passed, instead collects `value_func(item)` into each list instead of `item`.

If *reduce_func* is passed, instead of collecting the items into lists, *reduce* over the items of each key with *reduce_func*, effectively implementing a MapReduce operation. If keys are intended to be unique, set `reduce_func=False` (`reduce_func=False` is also the default if *collect_in* is passed). If *reduce_func* is passed, then *reduce_func_init* may also be passed, and will determine the initial value when reducing with *reduce_func*.

If *collect_in* is passed, initializes the collection from *collect_in* rather than as a `collections.defaultdict` (if `reduce_func=None`) or an empty `dict` (otherwise). Additionally, *reduce_func* defaults to `False` rather than `None` when *collect_in* is passed. Useful when you want to collect the results into a `pandas.DataFrame`.

If *map_using* is passed, calculates `key_func` and `value_func` by mapping them over the iterable using `map_using` as `map`. Useful with *process_map*/*thread_map*. See `.using_threads` and `.using_processes` methods below for simple shortcut methods that make use of `map_using` internally.

`collectby` is similar to `itertools.groupby` except that `collectby` aggregates common elements regardless of their order in the input iterable, whereas `groupby` only aggregates common elements that are adjacent in the input iterable.

mapreduce(*key_value_func*, *iterable*, *, *reduce_func*=None, *collect_in*=None, *reduce_func_init*=`...`, *map_using*=None)

`mapreduce(key_value_func, iterable)` functions the same as `collectby`, but allows calculating the keys and values together in one function. *key_value_func* must return a 2-tuple of `(key, value)`.

collectby.using_threads(*key_func*, *iterable*, *value_func*=None, *, *reduce_func*=None, *collect_in*=None, *reduce_func_init*=..., *ordered*=False, *chunksize*=1, *max_workers*=None)

collectby.using_processes(*key_func*, *iterable*, *value_func*=None, *, *reduce_func*=None, *collect_in*=None, *reduce_func_init*=..., *ordered*=False, *chunksize*=1, *max_workers*=None)

mapreduce.using_threads(*key_value_func*, *iterable*, *, *reduce_func*=None, *collect_in*=None, *reduce_func_init*=..., *ordered*=False, *chunksize*=1, *max_workers*=None)

mapreduce.using_processes(*key_value_func*, *iterable*, *, *reduce_func*=None, *collect_in*=None, *reduce_func_init*=..., *ordered*=False, *chunksize*=1, *max_workers*=None)

These shortcut methods call `collectby/mapreduce` with `map_using` set to *process_map/thread_map*, properly managed using the `.multiple_sequential_calls` method and the `stream=True` argument of *process_map/thread_map*. `reduce_func` will be called as soon as results arrive, and by default in whatever order they arrive in (to enforce the original order, pass *ordered*=True).

To make multiple sequential calls to `collectby.using_threads()/mapreduce.using_threads()`, manage them using `thread_map.multiple_sequential_calls()`. Similarly, use `process_map.multiple_sequential_calls()` to manage `.using_processes()`.

Note that, for very long iterables, it is highly recommended to pass a value other than the default 1 for *chunksize*.

As an example, `mapreduce.using_processes` is effectively equivalent to:

```
def mapreduce.using_processes(key_value_func, iterable, *, reduce_func=None,␣
↪ordered=False, chunksize=1, max_workers=None):
    with process_map.multiple_sequential_calls(max_workers=max_workers):
        return mapreduce(
            key_value_func,
            iterable,
            reduce_func=reduce_func,
            map_using=process_map$(
                stream=True,
                ordered=ordered,
                chunksize=chunksize,
            ),
        )
```

Example

Coconut:

```
user_balances = (
    balance_data
    |> collectby$(.user, value_func=.balance, reduce_func=(+))
)
```

Python:

```
from collections import defaultdict
```

(continues on next page)

(continued from previous page)

```
user_balances = defaultdict(int)
for item in balance_data:
    user_balances[item.user] += item.balance
```

async_map

async_map(*async_func*, **iters*, *strict*=False)

async_map maps *async_func* over *iters* asynchronously using anyio, which must be installed for *async_func* to work. *strict* functions as in *map/zip*, enforcing that all the *iters* must have the same length.

Equivalent to:

```
async def async_map[T, U](
    async_func: async T -> U,
    *iters: T$[],
    strict: bool = False
) -> U[]:
    """Map async_func over iters asynchronously using anyio."""
    import anyio
    results = []
    async def store_func_in_of(i, args):
        got = await async_func(*args)
        results.extend([None] * (1 + i - len(results)))
        results[i] = got
    async with anyio.create_task_group() as nursery:
        for i, args in enumerate(zip(*iters, strict=strict)):
            nursery.start_soon(store_func_in_of, i, args)
    return results
```

Example

Coconut:

```
async def load_pages(urls) = (
    urls
    |> async_map$(load_page)
    |> await
)
```

Python:

```
import anyio

async def load_pages(urls):
    results = [None] * len(urls)
    async def proc_url(i, url):
        results[i] = await load_page(url)
    async with anyio.create_task_group() as nursery:
        for i, url in enumerate(urls)
```

(continues on next page)

(continued from previous page)

```
        nursery.start_soon(proc_url, i, url)
    return results
```

3.10.7 Typing-Specific Built-Ins

TYPE_CHECKING

The TYPE_CHECKING variable is set to False at runtime and True during type_checking, allowing you to prevent your typing imports and TypeVar definitions from being executed at runtime. By wrapping your typing imports in an if TYPE_CHECKING: block, you can even use the typing module on Python versions that don't natively support it. Furthermore, TYPE_CHECKING can also be used to hide code that is mistyped by default.

Python Docs

A special constant that is assumed to be True by 3rd party static type checkers. It is False at runtime. Usage:

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: expensive_mod.SomeType) -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

Examples

Coconut:

```
if TYPE_CHECKING:
    from typing import List
x: List[str] = ["a", "b"]
```

```
if TYPE_CHECKING:
    def factorial(n: int) -> int: ...
else:
    def factorial(0) = 1
    addpattern def factorial(n) = n * factorial(n-1)
```

Python:

```
try:
    from typing import TYPE_CHECKING
except ImportError:
    TYPE_CHECKING = False
```

(continues on next page)

(continued from previous page)

```
if TYPE_CHECKING:
    from typing import List
x: List[str] = ["a", "b"]
```

```
try:
    from typing import TYPE_CHECKING
except ImportError:
    TYPE_CHECKING = False

if TYPE_CHECKING:
    def factorial(n: int) -> int: ...
else:
    def factorial(n):
        if n == 0:
            return 1
        else:
            return n * factorial(n-1)
```

reveal_type and reveal_locals

When using MyPy, `reveal_type(<expr>)` will cause MyPy to print the type of `<expr>` and `reveal_locals()` will cause MyPy to print the types of the current `locals()`. At runtime, `reveal_type(x)` is always the identity function and `reveal_locals()` always returns `None`. See the MyPy documentation for more information.

Example

Coconut:

```
> coconut --mypy
Coconut Interpreter vX.X.X:
(enter 'exit()' or press Ctrl-D to end)
>>> reveal_type(fmap)
<function fmap at 0x00000239B06E2040>
<string>:17: note: Revealed type is 'def [_T, _U] (func: def (_T`-1) -> _U`-2, obj:␣
↪typing.Iterable[_T`-1]) -> typing.Iterable[_U`-2]'
>>>
```

Python

```
try:
    from typing import TYPE_CHECKING
except ImportError:
    TYPE_CHECKING = False

if not TYPE_CHECKING:
    def reveal_type(x):
        return x
```

(continues on next page)

(continued from previous page)

```
from coconut.__coconut__ import fmap
reveal_type(fmap)
```

3.11 Coconut API

3.11.1 coconut.embed

coconut.embed(*kernel*=None, *depth*=0, ***kwargs*)

If *kernel*=False (default), embeds a Coconut Jupyter console initialized from the current local namespace. If *kernel*=True, launches a Coconut Jupyter kernel initialized from the local namespace that can then be attached to. The *depth* indicates how many additional call frames to ignore. *kwargs* are as in IPython.embed or IPython.embed_kernel based on *kernel*.

Recommended usage is as a debugging tool, where the code `from coconut import embed; embed()` can be inserted to launch an interactive Coconut shell initialized from that point.

3.11.2 Automatic Compilation

Automatic compilation lets you simply import Coconut files directly without having to go through a compilation step first. Automatic compilation can be enabled either by importing `coconut.api` before you import anything else, or by running `coconut --site-install`.

Once automatic compilation is enabled, Coconut will check each of your imports to see if you are attempting to import a `.coco` file and, if so, automatically compile it for you. Note that, for Coconut to know what file you are trying to import, it will need to be accessible via `sys.path`, just like a normal import.

Automatic compilation always compiles with `--target sys --line-numbers --keep-lines` by default. On Python 3.4+, automatic compilation will use a `__coconut_cache__` directory to cache the compiled Python. Note that `__coconut_cache__` will always be removed from `__file__`.

Automatic compilation is always available in the Coconut interpreter or when using `coconut-run`. When using auto compilation through the Coconut interpreter, any compilation options passed in will also be used for auto compilation. Additionally, the interpreter always allows importing from the current working directory, letting you easily compile and play around with a `.coco` file simply by running the Coconut interpreter and importing it.

If using the Coconut interpreter, a `reload` built-in is always provided to easily reload (and thus recompile) imported modules.

3.11.3 Coconut Encoding

While automatic compilation is the preferred method for dynamically compiling Coconut files, as it caches the compiled code as a `.py` file to prevent recompilation, Coconut also supports a special

```
# coding: coconut
```

declaration which can be added to `.py` files to have them treated as Coconut files instead. To use such a coding declaration, you'll need to either run `coconut --site-install` or `import coconut.api` at some point before you first attempt to import a file with a `# coding:  coconut` declaration. Like automatic compilation, the Coconut encoding is always available from the Coconut interpreter. Compilation always uses the same parameters as in the *Coconut Jupyter kernel*.

3.11.4 `coconut.api`

In addition to enabling automatic compilation, `coconut.api` can also be used to call the Coconut compiler from code instead of from the command line. See below for specifications of the different api functions.

Deprecated: `coconut.convenience` *is a deprecated alias for* `coconut.api`.

`get_state`

coconut.api.get_state(*state*=`None`)

Gets a state object which stores the current compilation parameters. State objects can be configured with ***setup*** or ***cmd*** and then used in ***parse*** or other endpoints.

If *state* is `None`, gets a new state object, whereas if *state* is `False`, the global state object is returned.

parse

coconut.api.parse(*code*=`""`, *mode*=`"sys"`, *state*=`False`, *keep_internal_state*=`None`)

Likely the most useful of the api functions, `parse` takes Coconut code as input and outputs the equivalent compiled Python code. *mode* is used to indicate the context for the parsing and *state* is the state object storing the compilation parameters to use as obtained from ***get_state*** (if `False`, uses the global state object). *keep_internal_state* determines whether the state object will keep internal state (such as what *custom operators* have been declared)—if `None`, internal state will be kept iff you are not using the global *state*.

If *code* is not passed, `parse` will output just the given *mode*'s header, which can be executed to set up an execution environment in which future code can be parsed and executed without a header.

Each *mode* has two components: what parser it uses, and what header it prepends. The parser determines what Coconut code is allowed as input, and the header determines how the compiled Python can be used. Possible values of *mode* are:

- `"sys"`: (the default)
  - parser: file
    - The file parser can parse any Coconut code.
  - header: sys
    - This header imports `coconut.__coconut__` to access the necessary Coconut objects.
- `"exec"`:
  - parser: file
  - header: exec
    - When passed to `exec` at the global level, this header will create all the necessary Coconut objects itself instead of importing them.
- `"file"`:
  - parser: file
  - header: file
    - This header is meant to be written to a `--standalone` file and should not be passed to `exec`.
- `"package"`:
  - parser: file
  - header: package
    - This header is meant to be written to a `--package` file and should not be passed to `exec`.
- `"block"`:
  - parser: file
  - header: none
    - No header is included, thus this can only be passed to `exec` if code with a header has already been executed at the global level.
- `"single"`:
  - parser: single
    - Can only parse one line of Coconut code.
  - header: none

- "eval":
  - parser: eval
    - Can only parse a Coconut expression, not a statement.
  - header: none
- "lenient":
  - parser: lenient
    - Can parse any Coconut code, allows leading whitespace, and has no trailing newline.
  - header: none
- "xonsh":
  - parser: xonsh
    - Parses Coconut xonsh code for use in *Coconut's xonsh support*.
  - header: none

Example

```
from coconut.api import parse
exec(parse())
while True:
    exec(parse(input(), mode="block"))
```

setup

coconut.api.setup(*target*=None, *strict*=False, *minify*=False, *line_numbers*=True, *keep_lines*=False, *no_tco*=False, *no_wrap*=False, *, *state*=False)

setup can be used to set up the given state object with the given compilation parameters, each corresponding to the command-line flag of the same name. *target* should be either None for the default target or a string of any *allowable target*.

If *state* is False, the global state object is used.

warm_up

coconut.api.warm_up(*streamline*=True, *enable_incremental_mode*=False, *, *state*=False)

Can optionally be called to warm up the compiler and get it ready for parsing. Passing *streamline* will cause the warm up to take longer but will substantially reduce parsing times (by default, this level of warm up is only done when the compiler encounters a large file). Passing *enable_incremental_mode* will enable the compiler's incremental mdoe, where parsing some string, then later parsing a continuation of that string, will yield substantial performance improvements.

`cmd`

coconut.api.cmd(*args*=`None`, *, *argv*=`None`, *interact*=`False`, *default_target*=`None`, *default_jobs*=`None`, *state*=`False`)

Executes the given *args* as if they were fed to `coconut` on the command-line, with the exception that unless *interact* is true or `-i` is passed, the interpreter will not be started. Additionally, *argv* can be used to pass in arguments as in `--argv` and *default_target* can be used to set the default `--target`.

Has the same effect of setting the command-line flags on the given *state* object as `setup` (with the global `state` object used when *state* is `False`).

`cmd_sys`

coconut.api.cmd_sys(*args*=`None`, *, *argv*=`None`, *interact*=`False`, *default_target*=`"sys"`, *default_jobs*=`"0"`, *state*=`False`)

Same as `coconut.api.cmd` but *default_target* is `"sys"` rather than `None` (universal) and *default_jobs*=`"0"` rather than `None` (`"sys"`). Since `cmd_sys` defaults to not using `multiprocessing`, it is preferred whenever that might be a problem, e.g. if you're not inside an `if __name__ == "__main__"` block on Windows.

`coconut_exec`

coconut.api.coconut_exec(*expression*, *globals*=`None`, *locals*=`None`, *state*=`False`, *keep_internal_state*=`None`)

Version of `exec` which can execute Coconut code.

`coconut_eval`

coconut.api.coconut_eval(*expression*, *globals*=`None`, *locals*=`None`, *state*=`False`, *keep_internal_state*=`None`)

Version of `eval` which can evaluate Coconut code.

`auto_compilation`

coconut.api.auto_compilation(*on*=`True`, *args*=`None`, *use_cache_dir*=`None`)

Turns *automatic compilation* on or off. This function is called automatically when `coconut.api` is imported.

If *args* is passed, it will set the Coconut command-line arguments to use for automatic compilation. Arguments will be processed the same way as with *`coconut-run`* such that `--quiet --target sys --keep-lines` will all be set by default.

If *use_cache_dir* is passed, it will turn on or off the usage of a `__coconut_cache__` directory to put compile files in rather than compiling them in-place. Note that `__coconut_cache__` will always be removed from `__file__`.

use_coconut_breakpoint

coconut.api.use_coconut_breakpoint(*on*=True)

Switches the breakpoint built-in which Coconut makes universally available to use *coconut.embed* instead of pdb.set_trace (or undoes that switch if on=False). This function is called automatically when coconut.api is imported.

find_packages and find_and_compile_packages

coconut.api.find_packages(*where*=".", *exclude*=(), *include*=("*",))

coconut.api.find_and_compile_packages(*where*=".", *exclude*=(), *include*=("*",))

Both functions behave identically to setuptools.find_packages, except that they find Coconut packages rather than Python packages. find_and_compile_packages additionally compiles any Coconut packages that it finds in-place.

Note that if you want to use either of these functions in your setup.py, you'll need to include coconut as a build-time dependency in your pyproject.toml. If you want setuptools to package your Coconut files, you'll also need to add global-include *.coco to your MANIFEST.in and pass include_package_data=True to setuptools.setup.

Example

```
# if you put this in your setup.py, your Coconut package will be compiled in-place␣
↪whenever it is installed

from setuptools import setup
from coconut.api import find_and_compile_packages

setup(
    name=...,
    version=...,
    packages=find_and_compile_packages(),
)
```

version

coconut.api.version([*which*])

Retrieves a string containing information about the Coconut version. The optional argument *which* is the type of version information desired. Possible values of *which* are:

- "num": the numerical version (the default)
- "name": the version codename
- "spec": the numerical version with the codename attached
- "tag": the version tag used in GitHub and documentation URLs
- "-v": the full string printed by coconut -v

`CoconutException`

If an error is encountered in a api function, a `CoconutException` instance may be raised. `coconut.api.CoconutException` is provided to allow catching such errors.

3.11.5 `coconut.__coconut__`

It is sometimes useful to be able to access Coconut built-ins from pure Python. To accomplish this, Coconut provides `coconut.__coconut__`, which behaves exactly like the `__coconut__.py` header file included when Coconut is compiled in package mode.

All Coconut built-ins are accessible from `coconut.__coconut__`. The recommended way to import them is to use `from coconut.__coconut__ import` and import whatever built-ins you'll be using.

Example

```
from coconut.__coconut__ import process_map
```

CHAPTER

FOUR

COCONUT CONTRIBUTING GUIDELINES

By contributing to Coconut, you agree to your contribution being released under Coconut's Apache 2.0 license.

Anyone is welcome to submit an issue or pull request! The purpose of this document is simply to explain the contribution process and the internals of how Coconut works to make contributing easier.

If you are considering contributing to Coconut, you'll be doing so on the `develop` branch, which means you should be viewing the `develop` version of the Contributing Guidelines, if you aren't doing so already.

4.1 Asking Questions

If you are thinking about contributing to Coconut, please don't hesitate to ask questions at Coconut's Gitter! That includes any questions at all about contributing, including understanding the source code, figuring out how to implement a specific change, or just trying to figure out what needs to be done.

4.2 Good First Issues

Want to help out, but don't know what to work on? Head over to Coconut's open issues and look for ones labeled "good first issue." These issues are those that require less intimate knowledge of Coconut's inner workings, and are thus possible for new contributors to work on.

4.3 Contribution Process

Contributing to Coconut is as simple as

1. forking Coconut on GitHub,
2. making changes to the `develop` branch, and
3. proposing a pull request.

Note: Don't forget to add yourself to the "Authors:" section in the moduledocs of any files you modify!

4.4 Testing New Changes

First, you'll want to set up a local copy of Coconut's recommended development environment. For that, just run `git checkout develop`, make sure your default `python` installation is some variant of Python 3, and run `make dev`. That should switch you to the `develop` branch, install all possible dependencies, bind the `coconut` command to your local copy, and set up pre-commit, which will check your code for errors for you whenever you `git commit`.

Then, you should be able to use the Coconut command-line for trying out simple things, and to run a paired-down version of the test suite locally, just `make test-univ`.

After you've tested your changes locally, you'll want to add more permanent tests to Coconut's test suite. Coconut's test suite is primarily written in Coconut itself, so testing new features just means using them inside of one of Coconut's `.coco` test files, with some `assert` statements to check validity.

4.5 File Layout

- `DOCS.md`
  - Markdown file containing detailed documentation on every Coconut feature. If you are adding a new feature, you should also add documentation on it to this file.
- `FAQ.md`
  - Markdown file containing frequently asked questions and their answers. If you had a question you wished was answered earlier when learning Coconut, you should add it to this file.
- `HELP.md`
  - Markdown file containing Coconut's tutorial. The tutorial should be a streamlined introduction to Coconut and all of its most important features.
- `Makefile`
  - Contains targets for installing Coconut, building the documentation, checking for dependency updates, etc.
- `setup.py`
  - Using information from `requirements.py` and `constants.py` to install Coconut. Also reads `README.rst` to generate the PyPI description.
- `conf.py`
  - Sphinx configuration file for Coconut's documentation.
- coconut
  - `__coconut__.py`
    - Mimics the Coconut header by generating and executing it when imported. Used by the REPL.
  - `__init__.py`
    - Includes the implementation of the `%coconut` IPython magic.
  - `__main__.py`
    - Imports and runs `main` from `main.py`.
  - `constants.py`
    - All constants used across Coconut are defined here, including dependencies, magic numbers/strings, etc.

- convenience.py
  * Contains cmd, version, setup, and parse functions as convenience utilities when using Coconut as a module. Documented in DOCS.md.
- exceptions.py
  * All of the exceptions raised by Coconut are defined here, both those shown to the user and those used only internally.
- highlighter.py
  * Contains Coconut's Pygments syntax highlighter, as well as modified Python highlighters that don't fail if they encounter unknown syntax.
- main.py
  * Contains main and main_run, the entry points for the coconut and coconut-run commands, respectively.
- requirements.py
  * Processes Coconut's requirements from constants.py into a form setup.py can use, as well as checks for updates to Coconut's dependencies.
- root.py
  * root.py creates and executes the part of Coconut's header that normalizes Python built-ins across versions. Whenever you are writing a new file, you should always add from coconut.root import * to ensure compatibility with different Python versions. root.py also sets basic version-related constants.
- terminal.py
  * Contains utilities for displaying messages to the console, mainly logger, which is Coconut's primary method of logging a message from anywhere.
- command
  * __init__.py
    · Imports everything in command.py.
  * cli.py
    · Creates the ArgumentParser object used to parse Coconut command-line arguments.
  * command.py
    · Contains Command, whose start method is the main entry point for the Coconut command-line utility.
  * mypy.py
    · Contains objects necessary for Coconut's --mypy flag.
  * util.py
    · Contains utilities used by command.py, including Prompt for getting syntax-highlighted input, and Runner for executing compiled Python.
  * watch.py
    · Contains objects necessary for Coconut's --watch flag.
- compiler
  * __init__.py

        - Imports everything in `compiler.py`.

    * `compiler.py`
        - Contains `Compiler`, the class that actually compiles Coconut code. `Compiler` inherits from `Grammar` in `grammar.py` to get all of the basic grammatical definitions, then extends them with all of the handlers that depend on the compiler's options (e.g. the current `--target`). `Compiler` also does pre- and post-processing, including replacing strings with markers (pre-processing) and adding the header (post-processing).
    * `grammar.py`
        - Contains `Grammar`, the class that specifies Coconut's grammar in PyParsing. Coconut performs one-pass compilation by attaching "handlers" to specific grammar objects to transform them into compiled Python. `grammar.py` contains all basic (non-option-dependent) handlers.
    * `header.py`
        - Contains `getheader`, which generates the header at the top of all compiled Coconut files.
    * `matching.py`
        - Contains `Matcher`, which handles the compilation of all Coconut pattern-matching, including `match` statements, destructuring assignment, and pattern-matching functions.
    * `util.py`
        - Contains utilities for working with PyParsing objects that are primarily used by `grammar.py`.
    * templates
        - `header.py_template`
        - Template for the main body of Coconut's header; use and formatting of this file is all in `header.py`.
- icoconut
    * `__init__.py`
        - Imports everything from `icoconut/root.py`.
    * `__main__.py`
        - Contains the main entry point for Coconut's Jupyter kernel.
    * `root.py`
        - Contains the implementation of Coconut's Jupyter kernel, made by subclassing the IPython kernel.
- tests
    * `__init__.py`
        - Imports everything in `main_test.py`.
    * `__main__.py`
        - When run, compiles all of the test source code, but *does not run any tests*. To run the tests, the command `make test`, or a `pytest` command to run a specific test, is necessary.
    * `main_test.py`
        - Contains `TestCase` subclasses that run all of the commands for testing the Coconut files in `src`.
    * src

- `extras.coco`
- Directly imports and calls functions in the Coconut package, including from `convenience.py` and icoconut.
- `runnable.coco`
- Makes sure the argument `--arg` was passed when running the file.
- `runner.coco`
- Runs `main` from `cocotest/agnostic/main.py`.
- cocotest
- *Note: Files in the folders below all get compiled into the top-level cocotest directory. The folders are only for differentiating what files to compile on what Python version.*
- agnostic
- `__init__.coco`
- Contains a docstring that `main.coco` asserts exists.
- `main.coco`
- Contains the main test entry point as well as many simple, one-line tests.
- `specific.coco`
- Tests to be run only on a specific Python version, but not necessarily only under a specific `--target`.
- `suite.coco`
- Tests objects defined in `util.coco`.
- `tutorial.coco`
- Tests all the examples in `TUTORIAL.md`.
- `util.coco`
- Contains objects used in `suite.coco`.
- python2
- `py2_test.coco`
- Tests to be run only on Python 2 with `--target 2`.
- python3
- `py3_test.coco`
- Tests to be run only on Python 3 with `--target 3`.
- python35
- `py35_test.coco`
- Tests to be run only on Python 3.5 with `--target 3.5`.
- python36
- `py36_test.coco`
- Tests to be run only on Python 3.6 with `--target 3.6`.

– coconut-stubs

- `__coconut__.pyi`
  - A MyPy stub file for specifying the type of all the objects defined in Coconut's package header (which is saved as `__coconut__.py`).

4.6 Release Process

1. Preparation:
   1. Run `make check-reqs` and update dependencies as necessary
   2. Run `sudo make format`
   3. Make sure `make test`, `make test-py2`, and `make test-easter-eggs` are passing
   4. Ensure that `coconut --watch` can successfully compile files when they're modified
   5. Check changes in `compiled-cocotest`, `pyprover`, and `coconut-prelude`
   6. Check Codebeat and LGTM for `coconut` and `compiled-cocotest`
   7. Make sure `coconut-develop` package looks good
   8. Run `make docs` and ensure local documentation looks good
   9. Make sure all of the following are passing:
      1. Github Actions
      2. AppVeyor
      3. readthedocs
   10. Make sure develop documentation looks good
   11. Turn off `develop` in `root.py`
   12. Set `root.py` to new version number
   13. If major release, set `root.py` to new version name
2. Pull Request:
   1. Move unresolved issues to new milestone
   2. Create a pull request to merge `develop` into `master`
   3. Link contributors on pull request
   4. Wait until everything is passing
3. Release:
   1. Release a new version of `sublime-coconut` if applicable
      1. Edit the `package.json` with the new version
      2. Run `make publish`
      3. Release a new version on GitHub
   2. Merge pull request and mark as resolved
   3. Release `master` on GitHub
   4. `git fetch`, `git checkout master`, and `git pull`
   5. Run `sudo make upload`

6. `git checkout develop`, `git rebase master`, and `git push`
7. Turn on `develop` in `root`
8. Run `sudo make dev`
9. Push to `develop`
10. Wipe all updated versions on readthedocs
11. Build all updated versions on readthedocs
12. Copy PyPI keywords to readthedocs tags
13. Get SHA-256 hash from PyPI `.tar.gz` file and use that as well as the current version requirements in `constants.py` to update the local feedstock
14. Submit PR to update Coconut's `conda-forge` feedstock
15. Update website if it needs updating
16. Wait until feedstock PR is passing then merge it
17. Close release milestone

Coconut (coconut-lang.org) is a variant of Python that **adds on top of Python syntax** new features for simple, elegant, Pythonic **functional programming**.

Coconut is developed on GitHub and hosted on PyPI. Installing Coconut is as easy as opening a command prompt and entering:

```
pip install coconut
```

To help you get started, check out these links for more information about Coconut:

- Tutorial: If you're new to Coconut, a good place to start is Coconut's **tutorial**.
- Documentation: If you're looking for info about a specific feature, check out Coconut's **documentation**.
- Online Interpreter: If you want to try Coconut in your browser, check out Coconut's **online interpreter**.
- FAQ: If you have general questions about Coconut—like who Coconut is built for and whether or not you should use it—Coconut's frequently asked questions are often the best place to start.
- Create a New Issue: If you're having a problem with Coconut, creating a new issue detailing the problem will allow it to be addressed as soon as possible.
- Gitter: For any questions, concerns, or comments about anything Coconut-related, ask around at Coconut's Gitter, a GitHub-integrated chat room for Coconut developers.
- Releases: Want to know what's been added in recent Coconut versions? Check out the release log for all the new features and fixes.

CHAPTER

FIVE

CREDITS

5.1 Contributors

This project exists thanks to all the people who contribute! Become a contributor.

5.2 Backers

Thank you to all our backers! Become a backer.

5.3 Sponsors

Support Coconut by becoming a sponsor. Your logo will show up here with a link to your website. Become a sponsor.