# Contents

1 Introduction 3
   1.1 Python ................................................................. 3
   1.2 Hello World ........................................................... 3
   1.3 Errors ....................................................................... 4
   1.4 Object oriented programming ........................................ 5
   1.5 More about Python ..................................................... 6
   1.6 Language goals .......................................................... 6

2 Turtles 9
   2.1 Interactive interpreter .................................................. 9
   2.2 Turtles ..................................................................... 11
   2.3 Code in files ............................................................... 12
   2.4 Shape Exercises .......................................................... 13

3 Numbers 17
   3.1 Integers & Floats .......................................................... 17
   3.2 Number Operators ......................................................... 18
   3.3 The if conditional ......................................................... 18
   3.4 Number Exercises ........................................................ 19

4 Text 21
   4.1 Strings ..................................................................... 21
   4.2 User Input .................................................................... 22
   4.3 if and elif ................................................................. 22
   4.4 Exercises ................................................................. 22

5 Names 25
   5.1 Assignment ............................................................... 25
   5.2 Reusability ................................................................. 26
   5.3 Exercises ................................................................. 27
   5.4 Objects & Types Q&A .................................................. 28

6 Conditionals 29
   6.1 Code Blocks ............................................................. 29
   6.2 Equality ................................................................. 30
   6.3 The while loop ........................................................... 30
   6.4 Practicals ................................................................. 31

7 Functions 33
7.1 Defining and using functions ........................................... 33
7.2 Function parameters ...................................................... 34
7.3 Exercises ........................................................................ 35

8 Data Structures ............................................................... 37
8.1 Lists .............................................................................. 37
8.2 Dictionaries .................................................................... 38
8.3 Nesting ........................................................................... 39
8.4 The for loop .................................................................... 39
8.5 Exercises ........................................................................ 40

9 Conclusions ................................................................. 45
9.1 Programming ................................................................. 45
9.2 Abstractions .................................................................... 45
9.3 Design ............................................................................ 46
9.4 Exercises ........................................................................ 46

10 Appendix A: Mac OSX .................................................. 47
10.1 Terminal 101 .................................................................. 47
10.2 The Python Interpreter .................................................. 48

11 Appendix B: Debugging ................................................ 49
11.1 Errors ............................................................................. 49
11.2 pdb ................................................................................. 50

12 Appendix C: Classes ...................................................... 53
12.1 Defining & usage .......................................................... 53
12.2 snakes .......................................................................... 54
12.3 special methods ........................................................... 55
12.4 Exercises ....................................................................... 55

13 Appendix D: Resources .................................................. 57
13.1 Documentation ............................................................. 57
13.2 Tutorials ........................................................................ 57
13.3 Online Courses ............................................................ 58
13.4 Applied ......................................................................... 58
13.5 Course References ........................................................ 60

14 Key Words ................................................................. 61
14.1 Variable ........................................................................ 61
14.2 Assignment .................................................................... 61
14.3 Interpreter ..................................................................... 61
14.4 Namespace ..................................................................... 61
14.5 Boolean ......................................................................... 61
14.6 Conditional Flow Control ............................................. 62
14.7 Syntax .......................................................................... 62
14.8 Function ........................................................................ 62
14.9 Calling a Function ........................................................ 62
14.10 Parameters ................................................................... 63
14.11 Bugs ............................................................................. 63
14.12 Debugging .................................................................... 63
14.13 Parentheses .................................................................. 63
14.14 Terminal ....................................................................... 63
14.15 Execute ........................................................................ 63
14.16 Terminate ..................................................................... 64
14.17 Types ................................................................. 64
14.18 Objects ............................................................... 64
14.19 Class ................................................................. 64
14.20 Operators ............................................................ 64
Table of Contents:
Learning to program requires practice but is also very rewarding and creative.

The purpose of this course is to give you enough programming vocabulary to get a taste of what it is and to lay the foundational knowledge for you to build upon in the future.

Python

This introductory course will use a programming language called Python.

Python is popular both in academia, science and other industries. It is used for the Google search engine, Dropbox and at the Guardian, in games like Battlefield 2 and by NASA.

Together we will learn the correct syntax and grammar of the Python language.

Questions

1. What other languages have you heard of?

Hello World

All programming languages have a ‘Hello World’ tutorial - these are the most basic example of checking we have our programming language set-up and running correctly.
This is our Python ‘Hellow World’ tutorial.

First we need to open the ‘terminal’ - An application that provides text-based access to the operating system.

1. Press `Cmd(apple) + Spacebar` (the two keys together). A search box pops up.
2. Type `terminal` and press enter.

The program `terminal` will launch and you should see a prompt:

The line tells you your current location followed by `>.

**Tip:** We want to change folders (directories) into ‘documents’.

Take a look at the *Terminal 101* section to work out how to change folders.

`terminal` is a program often called a shell. It is an alternative to the point and click ‘graphical user interface’ that we are all used to.

Typing `python` and you enter the python interpreter:

```
~> python
Python 2.7.11 (default, Dec 26 2015, 17:47:15)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 {clang-700.1.81}] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Note the prompt has changed to `>>>`. Python is waiting for your instructions.

**Tip:** Do not confuse the shell with the python interpreter.

If the prompt is `>` you are in terminal If the prompt is `>>>` you are in the python interpreter.

In the interpreter type the following and press enter:

```
>>> print('Hello World')
```

Congratulations you have just written and executed a line of python!

**Exercise**

1. Explore and experiment with the interpreter. Try printing other words. Try printing sentences with punctuation.
2. What happens if you miss off the end bracket `print('hello')`? How do you continue?
3. Can you make Errors appear (it shouldn’t be too difficult)? How many different ones can you make? Make a list and google each one.
4. List the commands for printing ‘Hello World’ in other languages.

**Errors**

Troubleshooting errors is a large part of programming.
Typically given a problem to solve a programmer thinks up an idea that may work then battles through errors until it does work.

Often beginners understandably get frustrated with them. Don’t. Instead build up resilience by taking time to understand them. They are always correct and trying to guide you to a solution.

Here are three you will see a lot:

```python
>>> def asdfwe:
    File "<stdin>", line 1
    def asdfwe:
    ^
SyntaxError: invalid syntax
```

```python
>>> if 5 > 6:
... print('yes')
    File "<stdin>", line 2
    print('yes')
    ^
IndentationError: expected an indented block
```

```python
>>> def asdfwe:
    File "<stdin>", line 1
    def asdfwe:
    ^
SyntaxError: invalid syntax
```

By the end of this course, you should be able to instantly understand and map the above errors to solutions.

**Tip:** Troubleshooting Errors:

1. Reading error messages. Try to intuitively solve them.
2. Google errors. There isn’t a single error someone hasn’t already had.
3. Ask an expert. If really stuck ask someone for help.

**Object oriented programming**

We can see our world as containing different types of objects that we can classify according to common attributes and behaviours.

For example in a classroom there are many objects that are instances of the type `Chair` and many other objects that are instances of the type `Person`.

Objects have:

- Attributes - Chairs have four legs, Persons have two.
- Behaviours - Persons can walk. Persons can move chairs.

Objects can interact with other objects of different types. An object of type `Person` can sit on an object of type `Chair`. 
This is the essence of object oriented thinking. It is about using programmatic objects to model a domain of interest to a programmer.

This style of programming provides a clear, simple, and consistent model of computation that maps well to our intuitions about the world.

**More about Python**

Python is a simple to learn yet fully featured, high-level, object oriented programming language.

Two terms to bear in mind:

- **A programming language** A programming language is a formal computer language or constructed language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs to control the behavior of a machine or to express algorithms.

In other words, the Python language is the textual instructions you type.

- **An interpreter** A program (called python) that reads and executes (runs) a programming language, line-by-line.

When we ask the interpreter to execute the Python we write, it understands how to translate Python scripts, the instructions that are easy for a human to read and write, into the creation and manipulation of objects that a computer can understand.

This course introduces different types of Python objects: *String, Integer, Turtle, lists...*

You will discover what attributes and behaviours these objects have and how to use these to write programs to get stuff done.

Many of the concepts you learn about Python will apply to most other object oriented languages.

**Questions**

1. Explain in your own words but using the concepts *interpreter* and *language* what happened above when you ran the command `print('Hello World')`.

Start your answer “Using the Python language...”

**Language goals**

In this course, we will teach you about numbers, text, names, how conditional code works, how to write and execute functions and different data structures.

Our goal is to learn how we can make our code leaner, more re-usable and powerful such as moving from this:

```
turtle.forward(100)
turtle.left(55)
turtle.forward(100)
turtle.left(55)
turtle.forward(100)
turtle.left(55)
turtle.forward(100)
turtle.left(55)
turtle.forward(100)
turtle.left(55)
```

...to this:
The two code examples above perform the exact same function.

**Questions**

Amongst yourselves:

- What does the first piece of code above do?
- What does the second code extract do?
- Which do you prefer and why?
Turtles

Turtle objects know how to draw. Here we explore creating and manipulating them to draw on the screen. We also look at the two ways the python interpreter can execute your Python code:

- The interactive interpreter (typing directly into the terminal).
- Calling the python interpreter on a file that contains code.

Tip: Don’t just read! Type everything and experiment.

Interactive interpreter

We launch the python interpreter through terminal:

1. Press Cmd(apple) + Spacebar (the two keys together). A search box pops up.
2. Type terminal and press enter.

A window will appear with a prompt:

Type python to enter the interactive shell:

```
~ >python
...
>>>
```

The interpreter is awaiting your commands. Now type each of the lines, pressing enter after each line:

```
>>> from turtle import Turtle
>>> tess = Turtle()
```

A window will pop up called Python Turtle Graphics containing tess the turtle.
Try typing some different commands - forward and left are methods on our tess turtle object
Forward takes pixels as it's input - this will move tess forward 100 pixels:

```python
>>> tess.forward(100)
```

The left and right methods take degrees as their input - this will turn tess left 30 degrees:

```python
>>> tess.left(30)
```

Lets call some more methods on the tess our turtle object
Shape and colour take strings as their inputs:

```python
>>> tess.shape('turtle')
>>> tess.color('green')
```

Lets create 'bob' - a new turtle object:

```python
>>> bob = Turtle()
>>> bob.shape('circle')
>>> bob.color('red')
>>> bob.backward(100)
```
Clear your canvas and move both turtles back to the middle

```python
>>> bob.reset()
>>> tess.reset()
```

**Exercise**

Experiment drawing shapes in different colours.

1. Can you make bob a triangle so you can see which way he is pointing?
2. Can you draw a triangle?
3. What shapes can you draw using both turtles in different colours?

**Documentation**

Visit the *turtle* online documentation and explore what Turtle objects can do.

Questions:

- What different colors does a turtle’s `color` method recognise?
- What shapes does a turtle’s `shape` method recognise?

Find some new turtle object methods and experiment.

**Tip:** As you experiment you will want to do know how to do new things. Get into the habit of exploring the documentation to see what you can do.

**Turtles**

Let’s revise what we have learnt in the light of object oriented terminology.

An object can be created. It has a type, and this type determines its methods (behaviours).

**Creation**

```python
>>> from turtle import Turtle
>>> tess = Turtle()
```

Breakdown:

1. We import an object called Turtle from somewhere called turtle.
2. Turtle is called, creates a new object of type turtle, and returns it.
3. This returned object is assigned to the name tess.

**Tip:** We call an object by adding parenthesis (brackets) at the end of its name. Here the parenthesis are empty but they often aren’t.
Run this command to find the object type of tess:

```python
>>> type(tess)
```

What type of object is tess?

**Tip:** The function `type` returns the type of the object passed in.

Turtle is a special kind of object in that it produces new objects. We call it a constructor object.

**Methods**

Methods are functions attached to objects. We will explore functions later.

```python
>>> tess.forward(100)
```

Braces `{}` have a special meaning. They indicate calling. You can think of this as effecting an action.

The effect of calling the method `forward` on an object of type `Turtle` is to draw a line.

What other methods (behaviours) do turtle objects have?

**Code in files**

Most code is written and executed from a file.

Use SublimeText to create a file named `my_turtle_file.py` in the documents folder with this code:

```python
from turtle import Turtle, exitonclick

tess = Turtle()
tess.shape("turtle")
tess.forward(100)
exitonclick()
```

**Tip:** All file names with python code must end with `.py`

We can comment out lines of code using `#`

Open a terminal and make sure you are in the documents directory (*See Terminal 101*). If you still have the Python interpreter open in your terminal you can exit using `cmd + d`.

In the terminal call the python interpreter command with the filename `my_turtle_file.py` as parameter:

```bash
python my_turtle_file.py
```

**Tip:** Make sure the file you created exists in the location where you execute this command. The location is given by the prompt. You can use the ‘Change Directory’ command `cd` to get to the right directory E.g: `cd /folder` will move the current directory to ‘folder’
Questions/Practicals

1. What happens if you delete or comment out `exitonclick()` in your script and re-run it? Explain why you think this happens - discuss with your team and mentor.

2. What are the differences between using `python` interactively and using files? When would you use one or the other?

3. Challenge yourself to find as many different ways of drawing with a turtle object.

4. Take your time to draw something useful and/or crazy.

Shape Exercises

Lets program some shapes. We do this by breaking down the principles of geometry into step by step instructions.

Put all code inside a file named `shapes.py` to be executed using:

```
python shapes.py
```

Shapes:

- Draw a square as in the following picture.

**Tip:** Squares have right angles which are 90 degrees.

```
```

- Draw a rectangle.

```
```

- Draw an equilateral triangle.

**Tip:** An equilateral triangle has 3 sides of equal length and each corner has an angle of 60 degrees.

```
```

- Draw many squares. Each square should be tilted left of the previous.
Experiment with the angles between the individual squares. The picture shows three 20 degree turns. You could try 30 and 40...

- Draw a simple house.

**Tip:** Reuse the code you have already written.

- Really want a challenge? Draw the ‘flower’ shape below:
R=100, r=4, p=80
In this and the following section we examine three new types of objects

<table>
<thead>
<tr>
<th>Type</th>
<th>Used for</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>Numbers</td>
<td>-5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Floats</td>
<td>Decimal</td>
<td>3.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-23.001</td>
</tr>
<tr>
<td>Strings</td>
<td>Text</td>
<td>‘abc’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘bob’</td>
</tr>
</tbody>
</table>

We will look at how to create objects of these types and how to manipulate them.

**Integers & Floats**

*int* objects represent natural numbers. *float* objects represent rational numbers, numbers that have a decimal value. Both types can represent positive or negative numbers.

**Creation**

It is easier to create ints and floats than it was to create turtles.

You can just type them out:

```python
>>> 3
3
>>> type(-5)  # confirm type
int
>>> 3.4
3.4
>>> type(3.0)
<class 'float'>
```

**Questions**

1. What is the different between an *int* and a *float*?
2. Why do we have two different types to represent numbers?
3. Find some uses cases where you’d choose an *int* and others where a *float* is more suitable.
Number Operators

**Tip:** Unlike turtle object methods, we use operators to manipulate number objects. This special syntax exists as it maps to our expectations and so is more intuitive.

### Arithmetic operators

Two number objects separated by an arithmetic operator */ - +, actions behaviour we expect from basic arithmetic.

```python
>>> 5 + 4
9
>>> 5 - 6
-1
```

The behaviour is to compute and return the result as a new number object with the same type. In the above example 5 + 4 equals the 9 which is of type `int`.

### Questions

1. What is the difference between 2 * 5 + 3 and 2 * (5 + 3)? What are the answers of each expression?
2. What happens when you divide 5 by 2? What about 5 by 2.0. Is there a difference? If so, why?

### Comparison operators

Likewise two number objects separated by comparison operators == != >= <= < >, have the behaviour we expect.

```python
>>> 5 == 4
False # What is this?
>>> 5 < 6
True  # and this?
>>> 6 <= 6
True
```

**Tip:** `int` objects are used to solve problems that require manipulating numbers but with no decimal point such as age, days, and IDs.

These are expressions and these evaluate to `True` or `False`.

### The if conditional

This pattern:

```python
if <boolean expression>:
    <block of code>  # Note 4 space indent
```
mirrors the syntax required to define conditional behaviour.

Typically we use the result of comparison statements to make decisions on what code to execute:

```python
if 6 > 5:
    print('Greater Than')
```

`if` statements can combine with `else`:

```python
if 6 > 5:
    print('Greater Than')
else:
    print('Not Greater Than')
```

**Questions**

1. In an `if` conditional, what happens if the boolean is true?
2. When does the `else` block of code run in an `if else` statement?
3. What could you use an `if` conditional for?

**Number Exercises**

1. A bar wants to ensure only adults are allowed in. Write a program in a file named `bar.py` that prints ‘underaged’ or ‘ok’ depending on the age entered in the code.
2. A ride operator needs to ensure clients are taller than 150cm due to security. Write a program in a file named `ride.py` that will print ‘ok’ or ‘not tall enough’ given a height entered in the code.
3. A trader wants to algorithmically buy ‘ACME’ corp stock if they rise above 0.005$ but sell if they are below 0.001$. Write a script `trader.py` that prints ‘buy’, ‘sell’, ‘hold’ depending on a sale price entered in the script.
Strings

Text is represented and manipulated using objects of type `str`.

Creation

```python
>>> "hi"
hi
>>> type("hi")  # confirm type
str
```

When you execute the code "hi" or `str("hi")`, the python interpreter:

1. Creates an object of type `str`
2. Gives it the value “hi”
3. Returns this newly created object

Methods

Many! Consult the online documentation:

Questions

1. Using the documentation, how would you make a string all capitalized?
2. Given the string “I like chocolate”, how would you change the word chocolate to something else?
3. What other ways would you want to manipulate a string?
4. Try experimenting with the other methods that exist in the docs. How would you use the each of those methods to manipulate a string? Use the interpreter interactively to try them out.
Conversion functions

At some point, you might want convert between numbers and text.

You can use the `int` and `float` functions to convert `str` objects into number objects:

```python
>>> int('3')
3
>>> float('3.4')  # constructor can convert from str
3.4
>>> str(3)  # note the ''s that indicate a str object
'3'
>>> str(3.4)
'3.4'
```

User Input

To make programs interactive use a function named `input`:

```python
>>> name = input("Please enter your name: ")
Please enter your name:
```

When the interpreter meets `input` it:

1. Prints the string message passed as an argument to `input`,
2. Stores any characters typed
3. On `enter` returns the characters as a new String.

Here the new string is assigned to the name `name`.

So if the user types in `John` and then presses enter, a string object of value ‘John’ is assigned to name.

**if and `elif`**

We can define more complex conditional behaviour by combining `if` with `elif` and `else`:

```python
>>> x = input("Enter your age: ")  # input returns a str
Enter your age: 24
>>> x = int(x)  # convert to an int
>>> if x < 18:
...    print('You are a child')
... elif x == 18:
...    print('You have just turned into an adult')
... else:
...    print('You are an adult')
```

Exercises

1. Rewrite the number programs `bar.py`, `ride.py` and `trader.py` to take input from the user. Think of an appropriate question to print to screen to solicit a correct response.
What if the user enters nonsense? There is rarely a program without some form of validation. This is explored in the next exercise.

2. A sign up form on a website for the company ‘Very Big Corp. Of America’ requires information from its clients. The company wants to do gender based email marketing. Put this code in `big_corp.py`.

   a Write a program that asks clients their name, address, and gender. Ensure that gender is represented as either ‘m’, ‘f’. If it is not ask the user again. Once all information is inputted print ‘Hi Oliver, we have shaving blades reduced this week’ and for women ‘Hi Sarah, we have shampoo currently on sale’.

   b The same program now requires people to enter their email address. Add this but ensure it is in the right format. What would a well defined email address look like? How can you test that an email address is in the correct format?

As a challenge extend this program to handle uppercase M and F as well.

3. A mobile phone company bills clients on a certain plan differently depending on whether they have dialled a number containing 0845 or not. Write a program that asks the user which number they’d like to dial and informs the user whether it is ‘Free’ or ‘Paid’. Use `mobile.py`.

4. A geneticist needs your help identifying if a DNA sequence exists in a larger strand of DNA. A DNA sequence consists of a sequence of A, T, G, and Cs. Write a program that takes a DNA sequence from the user and confirms ‘Found’ or ‘Not Found’ depending on whether the input is contained in the target DNA strand. Use `dna.py`.

   DNA strand: ATTCGCCCTATGCTTAACC

   As a challenge, extend this program to check that the input is correct.
In this section we will examine names and assignment more closely.

**Tip:** Names and variables in this context mean the same thing. We use ‘name’ because Python uses this terminology.

**Assignment**

Assignment is one of the ways we associate names with objects. Names are how the interpreter knows what programmers are referring to.

Just like people have names so we can uniquely identify them, in Python the interpreter also needs to know what you are referring to when you give it instructions. For example you would not want it to be possible for there to be 2 methods called ‘forward’ which did different things as you would not know which method was going to be executed.

The namespace is a collection of all the available names that could be associated with an object. You can think of a namespace as like a register. A register is a list of all possible names a child in a class could have and each name is assigned to one child only and uniquely identifies them.

**Tip:** = the equals symbol means assignment and not equality (unlike in maths).

```python
>>> x = 5
```

The interpreter executes the above code as:

1. Create an `int` object of value 5.
2. Does `x` exist in the namespace?
   - True - update the name `x` to point to the new `int` object.
   - False - create a new name `x` in the namespace and make it point to the new object.

From the point of assignment onwards code can refer to that object by using the name `x`. The interpreter will know how to find it by looking up the value in the namespace.

A name is an expression and it evaluates to its object:

```python
>>> x
5
```
Names can be reassigned to any type of object:

```python
>>> x = 5  # x refers to an 'int' object
>>> x = 'greg'  # x refers now to a 'str' object
```

The mysterious `from ... import ...` that we saw earlier is just about adding names to the namespace so the interpreter knows what you are referring to:

```python
>>> from turtle import Turtle
>>> tess = Turtle()
```

**Visualising**

For each assignment:

- If the name already exists, the namespace (frame) is updated.
- If the name doesn’t exist, a new name is created pointing and it references the newly created object.

**NameError**

If the interpreter gets a name that hasn’t yet been defined through assignment it will complain by throwing a `NameError`.

**example:**

```python
>>> x = 5
>>> x
5
>>> the_holy_grail
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'the_holy_grail' is not defined
```

**Questions**

```python
five = "five"
```

What does each set of characters on either side of the equal sign mean?

**Reusability**

Names enhance a programmer’s expressivity. They permit generalising code thereby facilitating code reuse. Indeed they are often called variables.

Consider this code, which draws a square with side length 50:

```python
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
```
turtle.left(90)
turtle.forward(50)
turtle.left(90)

Now a decision is made that the sides be of length 100.
You have to go back and replace 50 with 100 four times.
Using names you can do this:

```python
side = 50
right_angle = 90

turtle.forward(side)
turtle.left(right_angle)
turtle.forward(side)
turtle.left(right_angle)
turtle.forward(side)
turtle.left(right_angle)
turtle.forward(side)
turtle.left(right_angle)
```

If you change your mind you need only update one value.
Mathematics tells us a square’s length can be of any size but that all sides must be the same length. Our new programmatic definition mirrors that.

**Tip:** If you find yourself needing to replace many similar values in order to update your code, using names is worth considering.

### Good Naming

The name `right_angle` was chosen to refer to an `int` of value 90.

We could have used `thirty_degree_angle`, `angle`, or `awef` and the code would work fine. However:

- `thirty_degree_angle` is misleading its 90 not 30 degrees.
- `angle` is perhaps OK but a little vague
- `awef` is nonsense and conveys no meaning

By choosing appropriate names you make the code more readable and intuitive. Readability is very important even for projects you are working on on your own. It is not always easy to remember what your intention was when you come back and look at your code. If you have used good variable and method names it will be easier to make changes to your code at a later date.

### Exercises

#### Age in 2050

Write a program that asks the user for her age and prints how old she will be in 2050.
Shapes
Refactor your code in shapes.py to use variables as much as possible.

Objects & Types Q&A
If you understand the answers to these you understand everything about objects and types!!

Tip: Use the interpreter to help you find answers

Describe in detail what the interpreter does when you type the following and enter:

```
>>> '5'
>>> 5
```

What is the result this line of code?:

```
3 < '5'
```

Instances of both str and int objects recognise the + symbol. What output would you expect of the following lines of code?

```
'1' + '2'
1 + 2
```

Try the same above but this time using * instead of +. What can you conclude of the meaning of *?
Conditionals give the python interpreter a choice of which lines of code to execute. Conditionals are a very powerful part of programming as they allow the program to carry out different behaviour depending on input rather than doing the same thing every time it is run.

Boolean expressions are lines of code that resolve to a boolean object. There are only two values that a boolean object can take: True or False.

Conditionals always base their decisions on the result of a boolean expression. They are always followed by a block of code.

Furthermore conditional loops enable us to harness logic relating to repetition.

**Code Blocks**

A block of code is one or more lines of code that will be executed one after the other. In Python a block is defined by the use of indentation.

All conditionals are followed by a code block whose execution depends on the outcome of the conditional expression above it. A conditional is often said to ‘guard’ the execution of a code block.

```python
a = 4
if a == 4:
    print('This code block will execute')
    result = 5 + a
else:
    print('This code block will not execute')
    result = a + 6
```

**Tip:** In other languages code blocks are defined by the use of braces: `{}`

**Questions**

1. Can you change the code above to make the else block execute?
Equality

Testing the equality of two objects returns True or False depending on how equality is defined on those two objects. Equality on strings is defined as follows:

```
>>> '5' == '5'
True
>>> '5' == '6'
False
```

Generally both objects have to - be of the same type and - have the same value in order to be equal:

```
>>> 5 == '5'
False
```

Questions

1. Write some equality statements of your own
2. Instead of == you can also use != <= < >= or >. Write some equality statements using these and if you’re not sure what they do see if you can work it out from whether you get a True or False response.

The while loop

The while <condition>: construct is a way of instructing the interpreter to repeatedly execute a code block indefinitely. The condition defines when the loop will terminate.

syntax

```
while <condition>:  # condition must evaluate to a boolean
    <code block>    # the indent defines the loop’s code block
```

example

```
>>> import random
>>> warm = 20
>>> temperature = random.randrange(5, 30)
>>> while temperature <= warm:
...     print('cold')
...     temperature = random.randrange(5, 30)
cold
cold
cold
```

Questions

1. What do you think random.randrange(5, 30) does?
visualising execution

**Tip:** A while loop is used to repeatedly execute an instruction until a condition is no longer true. You should make sure that the condition will eventually be false otherwise your program will run forever! If you do end up writing a loop like this use `ctrl + c` to terminate your program.

**loop keywords**

`break` is a keyword that instructs the interpreter to break out of a loop. `continue` instructs the interpreter to skip the rest of the loop code block and to return to the top of the loop.

**Practicals**

**Practical: Shoe Conversion**

A UK company wants to export shoes to continental Europe.

They hire you to write a program that prompts the user for a UK size and returns the equivalent size it would be in Europe.

Here is a conversion table:

<table>
<thead>
<tr>
<th>Europe</th>
<th>UK</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>5</td>
</tr>
<tr>
<td>39</td>
<td>6</td>
</tr>
<tr>
<td>40</td>
<td>7</td>
</tr>
<tr>
<td>42</td>
<td>8</td>
</tr>
</tbody>
</table>

Extension: Can you return an error message for the user if they enter an invalid shoe size?

**Practical: Loan**

A loan repayment plan consists of a balance, monthly interest and repayments.

The loan amount in question is £100. Repayments are made at £20. Interest is charged monthly at 10%.

Write a program that prints to screen the remaining balance after every month.

The program should terminate when the loan is completely paid off.

**Practical: BMI Calculator**

The NHS has hired you to create a BMI Calculator.

Write a command line program that asks a user for:

- Weight in Kilograms
- Height in Meters
and returns the BMI result, followed by the user’s BMI classification.

BMI Classification

**Tip:** You will have to do some research online for how to calculate a person’s BMI. Try working the maths out on paper first before you write the code.

<table>
<thead>
<tr>
<th>BMI</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.5 or less</td>
<td>Underweight</td>
</tr>
<tr>
<td>18.5 to 24.99</td>
<td>Normal Weight</td>
</tr>
<tr>
<td>25 to 29.99</td>
<td>Overweight</td>
</tr>
<tr>
<td>30 to 34.99</td>
<td>Obesity (Class 1)</td>
</tr>
<tr>
<td>35 to 39.99</td>
<td>Obesity (Class 2)</td>
</tr>
<tr>
<td>40 or greater</td>
<td>Morbid Obesity</td>
</tr>
</tbody>
</table>

**Practical: Turtles Joypad**

We want to control the movements of the turtle using instructions from the keyboard. Much like the way you’d control a character in a game.

Place this in a file called *turtle_joypad.py*:

```python
import turtle

tess = turtle.Turtle()

while True:
    move = input('
Type a w d s for left up right down (q to exit): ')
    if move == 'a':
        tess.setheading(180)  # west
        tess.forward(10)
        # [ ... put your code here ... ]
    if move == 'q':
        break
```
A function is some code which has been given a name and which can be used wherever you need it in your program. When we use a function we say we are “calling” it.

Some examples of functions we’ve already used are print, int and type.

```python
>>> print("hello")
hello
>>> int("13")
13
>>> type(True)
<type bool>
```

The print function simply prints out what is between the brackets. The int function turns something into an integer (in this example it’s a string “13”). The type function tells you the type of object something is.

All of these functions only require one value, or “parameter”, inside the brackets. But some functions can be called with zero parameters, for example the exit function:

```python
>>> exit()
```

Other functions require more than one parameter, for example the max function:

```python
>>> max(1, 2)
2
>>> max(-1000, 1000, 0)
1000
```

**Tip:** Like everything else in Python, functions are objects. Function objects contain code which you run using the brackets containing whatever parameters that function needs.

---

**Defining and using functions**

Functions help you to organise your code better. You can split up complicated tasks into several smaller, simpler ones. And they allow you to reuse code without repeating it.
Defining functions

Every function has to be “defined” before it can be used; functions have names like variables. To define a function you use `def`:

```python
>>> def say_hello(name):
...     print("hello " + name + "!")
...
```

The `def` keyword is followed by the function name, then the names of any parameters the function requires inside brackets, and then a colon. On the following lines you write the block of code for the function indented by four spaces, like it’s indented when you write an `if` or `while` block. We call this a “definition”.

Here’s an example of a definition of a function called “going_nowhere” which doesn’t have any parameters:

```python
>>> def going_nowhere():
...     turtle.forward(50)
...     turtle.backward(50)
...
```

**Tip:** It’s important to indent the code in a function definition by four spaces. If you don’t you’ll see an error saying: `IndentationError: expected an indented block`

Using functions

Once you’ve defined a function you can use, or “call”, it as many times as you need it by putting the function name followed by the brackets containing the parameters the function requires.

```python
>>> say_hello("Barry")
hello Barry!
>>> say_hello("Paul")
hello Paul!
>>> going_nowhere()
>>> exit()
```

Function parameters

Function parameters make functions more flexible and useful. For example compare this function for drawing a square without parameters:

```python
def draw_100_by_100_square():
    turtle.forward(100)
    turtle.left(90)
    turtle.forward(100)
    turtle.left(90)
    turtle.forward(100)
    turtle.left(90)
    turtle.forward(100)
    turtle.left(90)
```

to this one with an parameter called “size”:

```python
def draw_100_by_100_square(size):
    turtle.forward(size)
    turtle.left(90)
    turtle.forward(size)
    turtle.left(90)
    turtle.forward(size)
    turtle.left(90)
    turtle.forward(size)
    turtle.left(90)
```
```python
def draw_square(size):
    turtle.forward(size)
    turtle.left(90)
    turtle.forward(size)
    turtle.left(90)
    turtle.forward(size)
    turtle.left(90)
    turtle.forward(size)
    turtle.left(90)
```

The second function is more flexible; it can be used to draw a square of any size. Here are some more examples of functions with parameters:

```python
def left_diagonal(angle, length):
    turtle.left(angle)
    turtle.forward(length)
    turtle.right(angle)  # stay facing in the same direction

def draw_polygon(side_length, sides):
    for _ in range(sides):
        turtle.forward(side_length)
        turtle.left(360.0/sides)
```

**Tip:** See what happens if you define a function with a parameter which has the same name as a variable in your program.

```python
>>> name = "Miss Moneypenny"
>>> def witty_comeback(name):
...    print("Do you expect me to talk?")
...    print("No " + name + ", I expect you to die!")
...>>> witty_comeback("Mr Bond")
Do you expect me to talk?
No ??? I expect you to die!
```

## Exercises

### Shapes

1. Open your “shapes.py” file and define every shape as a function with parameters.
2. Write a simple program which uses your new functions to check that they work.
3. Rewrite your program for drawing a house using your shape functions.

### Conversions

4. Write a function called `celsius_to_fahrenheit` which has one parameter and converts a temperature in degrees celsius into the equivalent in degrees fahrenheit.
5. (Extension) Create a new program called “currency_converter.py” and copy the following code into it:

```python
import urllib2 import json

def get_conversion_rate(base, to):
    symbols=" + to)
    data = json.loads(response.read())
    return data["rates"][to]
```

This defines a function called `get_conversion_rates` which fetches the current exchange rate between two currencies from the Internet. For example:

```python
>>> get_conversion_rate("USD", "GBP")
0.751
```

Your task is to make the `currency_converter.py` program work like this:

```bash
$ python currency_converter.py
What currency do you want to convert from? USD
What currency do you want to convert to? GBP
How much do you want to convert? 100
100 USD = 75.1 GBP (exchange rate of 0.751)
```
Data structures are objects that contain or organise other objects in a useful way. We will look at two data structures in this lesson: lists and dictionaries (“dicts”).

We will learn how to create these data structures, and how to access and update their contents. We’ll also look at how to do something with every item in a data structure. To do this we use loops.

**Lists**

A list is a data structure which contains objects in a specific order.

**Creating lists**

To create a list from some other objects use square brackets like so:

```python
>>> monty_python = ['John', 'Terry G', 'Eric', 'Michael', 'Terry J', 'Graham']
>>> monty_python
['John', 'Terry G', 'Eric', 'Michael', 'Terry J', 'Graham']
>>> type(monty_python)
<type 'list'>
```

You can also create lists using some built-in functions such as `list` and `range`:

```python
>>> list('abc')
['a', 'b', 'c']
>>> range(4)
[0, 1, 2, 3]
```

`list` takes another object which can be represented as a list and returns that list. In this case it takes the string “abc” and turns it into a list of its letters.

`range` creates a list of numbers from zero up to some number.

**Accessing items**

You can access a specific item in a list like so:
>>> abc = ['a', 'b', 'c']
>>> abc[0]
'a'
>>> abc[2]
'c'

Tip: The first item in a list is at position zero, not position one.

Updating lists

<<<<<<< HEAD
You update lists in a similar way:

```python
>>> abc[2] = 'd'
>>> abc
['a', 'b', 'd']
```

AVX1101master

Tip: You can only access and update positions in the list which are already occupied. For example try doing

```
abc[25] = 'z';
```

the computer will say:

```
IndexError: list assignment index out of range
```

Dictionaries

A dictionary or “dict”, is another type of data structure which contains other objects. But unlike in a list the order of the objects is not important and we don’t access or update an item using its index.

Instead, each object has a name. These names are called “keys” and the objects are called “values”; a dictionary contains “key-value pairs”. If we have a key we can get access to the corresponding value.

Dictionaries are often used to contain information about a specific thing, in the examples below we use a dictionary to contain information a person.

Creating dicts

When creating a dict we use curly brackets. These enclose a series of pairs of keys and values like `key: value` separated by commas.

```python
>>> brian = {'name': 'Brian', 'age': 23, 'sex': 'M'}
>>> brian
{'age': 23, 'name': 'Brian', 'sex': 'M'}
>>> type(brian)
```

Tip: When `brian` is printed to the terminal, the keys and values aren’t necessarily shown in the order they were defined in. Dictionaries don’t care about the order of their contents like lists do.
Accessing values in a dict

You access a value in a dictionary in a similar way to how you access an item in a list:

```python
>>> brian['name']
'Brian'
>>> brian['age']
23
```

Updating a dict

The way you update a dict is similar to how you update a list too.

```python
>>> brian['age'] = 24
>>> brian
{'age': 24, 'name': 'Brian', 'sex': 'M'}
```

Tip: If you access or update a non-existent key the computer will say there is a `KeyError`.

Nesting

Data structures can include any type of object including other data structures.

Here is a list of dictionaries:

```python
>>> people = [
    {'name': 'Naomi', 'age': 32, 'sex': 'F', 'status': 'Single'},
    {'name': 'Jane', 'age': 29, 'sex': 'F', 'status': 'Married'},
    {'name': 'Brian', 'age': 23, 'sex': 'M', 'status': 'Single'}
]
```

Nested data structures are extremely common.

Think how this could be used to store information about all of the students in a class.

The `for` loop

Often, when we have a list of objects, we want to go through them one by one and do something with each of them.

Use `for` to iterate over each item in a given list.

```python
from turtle import Turtle, exitonclick

alex = Turtle()

for a_colour in ['yellow', 'red', 'purple', 'blue']:
    alex.color(a_colour)
    alex.forward(50)
    alex.left(90)

exitonclick()
```
Here by iterating through a list of `str` objects we change the colour of our turtle alex.

**Refactoring square**

We can refactor `square` by combining `range` with a `for` loop.

```python
def square(side):
    for i in range(4):
        turtle.forward(side)
        turtle.left(90)
```

Drawing a square is reduced to repeating the same action four times.

Thanks to the `for` loop our definition of a square in code:

- is shorter and more readable.
- communicates an insight into the geometry of a square.

**Exercises**

**Refactor `shapes.py`**

Refactor all the shapes in `shapes.py` and make good use of loops where you can.

**Hexagon**

Write code that draws this:
**Honeycomb**

Write code that draws this:

![Honeycomb Diagram]

**Any Shape**

Write code that can draw any shape like this:
Tip: The sum of the external angles of any shape is always 360 degrees.

Practical: Paper Scissors Rock

Steps:
1. The user inputs either ‘paper’, ‘scissors’ or ‘rock’.
2. The computer randomly chooses one too.
3. The outcome is printed, according to the rules of the game:
   • If the user chooses ‘paper’ and the computer chooses ‘rock’, then print ‘rock wins’
   • If the user chooses ‘scissors’ and the computer chooses ‘paper’ then print ‘scissors wins’
   • ... and so on ...
4. Exit

You will need to use some randomness:

```python
>>> import random
>>> random.choice(['a', 'b', 'c'])
```

Looping turtles

Using the following as template draw this:
Put the following in a file called `turtle_queue.py` and finish off the program.

```python
import turtle

number_of_turtles = 4

turtles = []
for _ in range(number_of_turtles):
    turtles.append(turtle.Turtle())

# position point of origin at bottom left of window
turtle.setworldcoordinates(0, 0, 600, 600)

for i, turtle_ in enumerate(turtles):
    turtle_.up()

# Evenly space out the turtles
for i, turtle_ in enumerate(turtles):
    ypos = 600 / number_of_turtles * i
    turtle_.setpos(0, ypos)

for i, turtle_ in enumerate(turtles):
    turtle_.down()
```

8.5. Exercises
# Your turn! Enter your code here #

###

---

Chapter 8. Data Structures
Conclusions

Programming

The constructs we have learned (loops, conditions, data structures) mean that we can be very expressive as programmers.

Combined with abstractions we can combine new programs together.

Building on our previously defined concept of a house we now use repetition to define a row of houses.

```python
def row_of_houses(number, size):
    for i in range(number):
        house(size)
        turtle.forward(size)
```

This is how complex and useful programs are built.

Abstractions

We have gone from understanding this:

```
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
```

to programming like this:

```python
def square(side):
    for i in range(4):
        turtle.forward(side)
        turtle.left(90)
```

Why is the second version better than the first?

Computers are complex. Even the smallest operation hides layers of incredible complexity. Programming is not only about getting a computer to do things. It is about writing code that is useful to humans.
Good programming is harnessing complexity by writing code that rhymes with our intuitions. Good code is code that we can use with a minimal amount of context and that allows us to be productive straight away.

By calling:

```
>>> square(100)
```

The above code called `square` can be understood even by a non programmer. Intuition helps because the code is defined at the appropriate level of abstraction over the complex details for understanding to take place.

The two major advantages are:

- detail and complexity is hidden.
- the definition of the function object called `square` is shorter clearer and truer to its mathematical (conceptual) definition.

This course illustrates that creative programming is about constructing useful abstractions. It is also about exercising your intuition to make you more productive.

**Design**

We have gone from step by step instructions to defining blocks of code in such a way as to define higher level concepts. Defining reusable components and the ability to repeat them is immensely powerful.

Think of everything you can make from Lego bricks. Minecraft is a world build with cubes. In the real world think of all the components and repetition you typically find in a skyscraper.

This is where programming starts to become creative. You can define the universe of things that is of interest to you.

**Exercises**

**A Text editor**

Think about the objects that you’d have to use to represent editing text.

**Your Project**

Programmers model other domains. Think of an area in which you are an expert and how you might code it.

What objects, functions and variables would need to be defined?
You interact with python using the terminal program.
You can find it by looking in the applications > utilities folder.

A shortcut:
1. Press `Cmd + space` (the two keys together)
2. A search prompt pops up.
3. Type `terminal` and press enter.

**Terminal 101**

Typically you interact with your operating system using a mouse with certain actions: point, click, drag. Using these you can launch programs and move files.

The terminal program offers the same interaction but using typed commands:

When `terminal` launches you get a prompt:

```
> 
```

The prompt gives you your current location followed by a `>`. Here I am in the directory `COMPLETE THIS` which itself is in the directory `Users`.

This will list all the files and folders in your current directory.

Here are all the commands you need for this course:

- `cd` - change directory
- `ls` - list the directory’s contents
- `cp` - copy a file or a directory
- `mv` - move a file or a directory
- `mkdir` - make a directory
- `rm` - delete a file or directory
- `unzip` - unzip a zipped (compressed) file
**Tip:** There is nothing here that you aren’t familiar doing with the mouse. If necessary use your mouse to orientate yourself.

## The Python Interpreter

Python is installed on Mac OSX machines by default

```
PATH>python
Python 3.4.2 (v3.4.2:8711a0951384, Sep 21 2014, 21:16:45) [MSC v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

**Tip:** At first it is normal to confuse the terminal and the python interpreter. Python instructions don’t run in the terminal shell and shell commands don’t work in the interpreter.

The interpreter has `>>>` as its prompt

The terminal shell has the file path eg `PATH >`
Appendix B: Debugging

Exceptions occur when the interpreter can’t carry out a given instruction. The type of error (Exceptions are objects) communicates what is wrong.

It is important to remember that most of programming is error driven. Don’t think of errors negatively, rather they are problem solving opportunities.

Debugging is working out what went wrong and fixing it.

Learn to be guided by Errors, and use debugging tools to master programming.

Here we explore some common errors and then we introduce pdb the python debugger.

Errors

Errors always tell you what when wrong but not always why.

Read errors, first using intuition then by debugging and research. Google is a very valuable resource when faced with an error, odds are you’re not the first person to come up against the problem.

Tip: You need to learn how to find information. Always read Errors and use your intuition, then Google. If that hasn’t helped only then ask an expert.

With time, you will know the solution to an error instantly.

AttributeError

An AttributeError means the interpreter can’t find the name you have asked for on the object.

```
>> import turtle
>>> turtle.shp('waef')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'shp'
```

Here the programmer has misspelt shape.
SyntaxError

Learning a language involves making many syntax (grammatical) errors.

A function defined badly:

```python
>>> def print_hi:
    File "<stdin>", line 1
    def print_hi
                  ^
SyntaxError: invalid syntax
```

Parentheses () are required after the name and the ending colon :.

```python
>>> def print_hi():
    print('hi')
```

No error, print_hi is properly defined.

NameError

```python
>>> var + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'var' is not defined
```

Here we try and use a variable var before we have defined it, we need to always ensure we define a variable before we use it:

```python
>>> var = 1
>>> var + 1
2
```

pdb

pdb is the python debugger. You can freeze execution at a particular point in time, step through it, examining objects as you go.

To execute code with pdb:

```bash
python -m pdb my.py
```

You can also pause the execution at any time by placing this line into your code:

```python
import pdb; pdb.set_trace()
```

When you run your code normally (python my.py) the interpreter will break at the that line of code.

Type h to get a list of all the commands. The important ones for now are:

Move along the execution timeline:

- l print lines of code surrounding cursor
- n execute next line
• s step into a line. Typically used for entering functions.
• c continue till the end of the program (or next break point).

Inspect the current location:
• w print frames on the stack at current position
• u go up a frame in stack
• d go down a frame in the stack

To exit: * q exit the debugger. Will terminate program execution.

**Tip:** On any error or exception enter a `import pdb; pdb.set_trace()` on the line preceding your program terminating. Run the program, then inspect what went wrong.

---

### example

We will use pythontutor hand in hand with pdb to exercise visualising program execution.

Put this code into a file named `my.py`:

```python
x = 1
y = 2
success = 'works'
failure = 'broken'

def inc(p):
    incremented = p + 1
    return incremented

def print_result(result):
    if result:
        print(success)
    else:
        print(failure)

inc_x = inc(x)
print_result(inc_x == y)
```

Execute with:

```
python -m pdb my.py
```

`pdb` starts program and pauses at first line:

```
> my.py(1)<module>()
--> x = 1
(Pdb)
```

Type l and press enter - this results in:

```
(Pdb) l
1  --> x = 1
2   y = 2
3  success = 'works'
4  failure = 'broken'
```
Step through each line of code by typing \texttt{n} and pressing enter.

Ensure you explore the two frames when you enter the \texttt{f} functions’ frame.

\textbf{Tip:} We have used a python code visualiser in a similar way.
Appendix C: Classes

We are now going to bring what we have learnt about object oriented programming together as we define our own object type using classes.

Classes are how programmers define objects that make new objects. Think of them as object templates or factories.

Tip: The Koans are structured as classes with each koan as a method.

Defining & usage

Much like we defined functions lets define a class.

A python.py file contains:

```python
class Python:
    """ A class that represents a snake """
    def __init__(self, name, sex, age, length):
        self.name = name
        self.sex = sex
        self.age = age
        self.length = length

    def move(self):
        print("{} moves".format(self.name))

    def eat(self):
        """ a snake gets longer when it eats """
        self.length = self.length + 1

    def starve(self):
        """ a snake shorter when it starves """
        # is there a bug here?
        self.length = self.length - 1
```

class object

Lets introspect the new type of object:
>>> from python import Python
>>> type(Python)
<class 'type'>
>>> dir(Python)
[ ... many methods ... ]

**instances**

A class is like an object instance factory. Here our class makes snakes.
Implicitly it runs the `__init__` function as defined on the class.
Creating:

```python
>>> john = Python('John', 'M', 15, 4)
>>> jane = Python('Jane', 'F', 4, 6)
```

Introspecting:

```python
>>> type(john)
<class 'python.Python'>
>>> dir(john)
[ ... many methods ... ]
```

Note we get `move`, `starve`, and `eat` which we defined, but we also get many methods others.

**Tip:** The other methods are those found when executing `dir(object)`

**methods**

```python
>>> Python.move
<function Python.move at 0x10f9b6840>
>>> john.move
<bound method Python.move of <python.Python object at 0x10fb04898>>
```

A function and a method are very similar. A function can stand alone, a method however is ‘bound’ to an object. When defined methods always take `self` as their first argument. It is thereby implicit when called.

**snakes**

The `__str__` special method is called on an object when we pass it to the `print` function.
We decide that the semantics of printing a python is to show a visual representation of a snake using characters.
Added to definition in `python.py`:

```python
class Python():
    def __init__(self, name, sex, age, length):
        self.name = name

[...]```
```python
def __str__(self):
    body = '=' * self.length
    return "/{}/".format(body)
```

results:
```
>>> from python import Python
>>> john = Python('John', 5)
>>> print(john)
~===%>
```

## special methods

### Exercises

#### attack

Decide on the semantics of a python attacking another object.
Implement your decision by defining a new method.

#### __add__

Lets define another special method to exploit the nice syntax python gives us.
Decide on the semantics of ‘adding’ pythons together.
Implement by defining your __add__ method on the Python class.
Appendix D: Resources

There are many resources freely available on the web for further learning. One of the strengths of Python relative to other languages is the diversity of applications it has. Do find one area of interest and make it your own.

Documentation

https://docs.python.org/3/

The important ones are:

- Tutorial
- Library Reference - Practical
- Language Reference - Academic, authoritative.

Tutorials

- start with these:
  - http://interactivepython.org/runestone/static/thinkcspsy/toc.html
  - http://learnpythonthehardway.org/ Not hard, hands on
- Popular:
  - http://inventwithpython.com/
  - http://www.diveintopython3.net/

Online Courses

Python today is widely taught in University. Here are some excellent resources.

- Udacity - CS101
• MIT Intro to Computer Science and Programming (MITOpenCourseware)
• Introduction to Interative Programming in Python (Coursera)
• Learn to Program: The Fundamentals (University of Toronto, Coursera)

Applied

The best way to consolidate learning programming is to find a domain that is of interest and learn more about programming it.

Here are some resources that will inspire.

Natural Language Processing

Natural Language processing refers to tools used to parse and add semantics to human language. Talking with computers.

• http://www.nltk.org/
• http://www.nltk.org/book/
• Python 3 Text Processing with NLTK 3 Cookbook

Web development

• https://docs.djangoproject.com/en/ Django Tutorial
• http://www.obeythetestinggoat.com/ To become a pro!

Games

• http://www.pygame.org/wiki/tutorials

3D Graphics

• Blender
  – http://www.blender.org/api/blender_python_api_2_72_1/
• Maya
  – http://zurbriggen.com/maya-python

Electronics

Raspberry Pi is currently the center of focus for fun with home electronics.

• Make: Sensors: A Hands-On Primer for Monitoring the Real World with Arduino and Raspberry Pi
• Make: More Electronics: Journey Deep Into the World of Logic Chips, Amplifiers, Sensors, and Randomicity
• Raspberry Pi Home Automation with Arduino
• Raspberry Pi for Secret Agents
• Raspberry Pi Cookbook for Python Programmers
• O'Reilly RPi
• Getting Started with Sensors: Measure the World with Electronics, Arduino, and Raspberry Pi

**Geospatial (GIS)**

• http://geospatialpython.com/
• http://qgis.org/en/site/
• http://docs.qgis.org/2.0/en/docs/pyqgis_developer_cookbook/

**Biology**

• http://biopython.org/DIST/docs/tutorial/Tutorial.html

**Maths**

• http://www.sagemath.org/doc/thematic_tutorials/
• http://www.sagemath.org/doc/tutorial/

**Forensics**

• Violent Python (book)
• Grey Hat Python (book)

**Mobile**

• http://kivy.org/docs/tutorials/pong.html

**Data Science**

• Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython (book)
• http://pandas.pydata.org/pandas-docs/dev/tutorials.html

**Machine Learning**

• Scikit-Learn: Machine Learning in Python
• Practical Data Science Cookbook (book)
• Building Probabilistic Models with Python (Book)
Course References

Some materials that inspired this course.

- Open Tech School

- How to Design Programs
  - [http://htdp.org/](http://htdp.org/)

- Structure and Interpretation of Computer Programs
Variable

A variable can be thought of as a labelled box. You store a value in the box then you can refer to them later. When you assign a value to a variable you are changing what is stored in it.

For example: $x = 5$ $x$ is the variable.

Assignment

In coding, assignment is giving a value to a variable.

For example: $x = 5$ 5 is assigned to the variable $x$

Interpreter

An interpreter takes a program written in a language that is human readable. This is because in order for a computer to be able to execute your code it needs to be in a very different format, google machine code if you’re interested what it looks like. Machine code is known as a low level language whereas Python is a high level language. This means that the interpreter takes care of lots of things for you such as managing memory allocation.

For example Python and translates it into instructions that a computer can execute. An interpreter does this line by line. This means if you have a 10 line program and a bug on the 5th line the interpreter will execute lines 1 to 4 before an error is raised.

Namespace

The namespace is a collection of all the available names that could be associated with an object. You can think of a namespace as like a register. A register is a list of all possible names a child in a class could have and each name is assigned to one child only and uniquely identifies them.

Boolean

A boolean is a type. It has only 2 possible values true or false.
Conditional Flow Control

Flow control is the order in which the individual instructions that make up a program are executed. Conditional flow control is when conditional execution (if statements) is used to change the flow of the execution.

Syntax

Syntax defines how to correctly structure your code. For example if you wrote

```python
>>> if 5 > 4
...
   print "5 is more than 4"
```

You would get the following error:

```python
>>> if 5 > 4
...
   print "5 is more than 4"
File "<stdin>", line 1
  if 5 > 4
^  
SyntaxError: invalid syntax
```

This is because the code is incorrectly structured. All if statements must end with a colon.

Function

A function is a block of instructions assigned to a name. For example:

```python
>>> def add_5_subtract_2(number):
...     return number + 5 - 2
...
>>> add_5_subtract_2(7)
10
```

Functions are useful as it means that you can reuse code. Every time you want to add 5 and subtract 2 from a number you can just call the function. This is particularly useful if you are writing large complex functions as you don’t have to type the code out each time. Also it also makes it easier to find problems in your code.

Calling a Function

Code inside a function does not run until the function is called. For example:

```python
>>> def say_hello(name):
...     return "Hello " + name
...
>>> say_hello("Susie")
Hello Susie
```

We have defined a function that gives instructions on how to say hello. However we don’t actually carry out the set of instructions until we call the function using:
Parameters

A parameter is additional information that is given to the function when it is called. In this example the parameter is `number`.

```python
>>> def add_5_subtract_2(number):
...     return number + 5 - 2
```

If a function has no parameters you will always get the same behaviour every time you call it. A function is much more powerful if it has parameters. In the case of the `add_5_subtract_2` function you will get a different response depending on the value of the parameter.

Bugs

An error in code that causes the program to break or not behave as expected.

Debugging

This is the process of finding errors in code and fixing them. Sometimes this is easy sometimes it is very difficult and time consuming. Debugging your code with someone else can often be very helpful.

Parentheses

() These are parentheses. They are common syntax in many programming languages. Parentheses can also be referred to brackets or braces.

Terminal

There are two main ways to interact with a computer. The most common way is using a GUI. A GUI is a Graphical User Interface and is focused around icons and mouse clicks. The terminal is another way to interact with a computer. It is text based and involves the user typing commands and the computer then displays a text response. For example (for macs) open the terminal:

1. Press `Cmd(apple) + Spacebar` (the two keys together). A search box pops up.
2. Type `terminal` and press enter.
3. Type `ls`

This will display a list of all the files and directories in your home directory.

Execute

Execution is when a computer runs a series of code instructions.
Terminate

A program has terminated when it has executed all instructions and stopped running.

Types

Types are used to classify the data in a program. Some examples of types are integer (whole numbers e.g. 5, 103), boolean (true or false), 'string' (characters surrounded by quotes e.g. “hello”, “ID1453”). Types are useful as it allows us to understand what we can do with data of this type for example it makes sense to divide one integer by another but dividing one string by another doesn’t make sense.

Types are also useful to understand how different bits of data within your program can interact with each other. For example you can add two integers together and get an answer however it is not possible to add a boolean to an integer as the two types are not compatible.

Objects

Objects in the real world have properties that describe them and actions they can carry out. For example a dog is an object. A dog can be described in terms of properties e.g. breed, colour, name etc. A dog also has actions that it can carry out e.g. fetch, lie down, bark etc.

We also have objects in programming. These objects also have properties to describe them and actions they can carry out. For example we may have an object that represents a video. The video will have properties for example a title and how long it is. There will also be actions on the video such as play and pause. In an object in programming the properties are variables and the actions are functions.

An object can also be referred to as an instance of a class.

Class

Classes and objects are very closely related. Classes are templates for creating objects. It is likely that you will want multiple copies of the same type of object. For example It is unlikely that you would only have one video you would want to have multiple video objects all representing different videos. This is when you use a class. A class describes what an object should look like. It determines what properties the object should have and what actions it can carry out. This class is then used to create an object. The object will have values for each of the properties. This is what make an object an instance of a class, the class indicates what properties an object should have and each object will have values for each one of these properties.

Operators

An operator is a character or set of characters that represents an action. Some examples of operators are +, &, -, .

Operators perform an action on operands. For example in the case of the sum 2 + 3

2 and 3 are operands and + is the operator.