Pyth Documentation

Release 1.0

Isaacg1

Nov 26, 2017

Contents

Pyth is an extremely concise language used for golfing. Try it here at: https://pyth.herokuapp.com. This is a tutorial/documentation/language-specification for it.

Contents:

CHAPTER 1

1. Getting Started

Pyth is a golfing language based on Python (hence the name) created by Programming Puzzles and Code Golf (PPCG) user Isaacg. Unlike most golfing languages, Pyth is fully procedural. Since Pyth is quite young, its features are constantly changing, and this document might be out of date for a few functions (Just check the source). Pyth is licensed under the MIT license.

1.1 1.1. How to Start Programming in Pyth

You have a few options when it comes to running Pyth. You could install it on your machine by cloning *the repository <https://github.com/isaacg1/pyth>* then adding this alias to your .bashrc:

```
alias pyth="python3 <path to pyth>/pyth.py"
```

(Windows users, you'll have to use the PATH and call `pyth.py`)

But the method we will be using in the tutorial, and the suggested one, is to use the online interpreter at http://pyth.herokuapp.com. This provides a programming environment with places in which to put code and input (and a handy function cheat-sheet!). The code is executed on the server and sent back to the webpage. The examples won't be too different if you decide to install the interpreter yourself. Installing the interpreter, however, will become necessary, when we start conducting "unsafe" operations which allow for arbitrary execution of code.

1.2 1.2. Hello World!

A customary start to programming tutorials is the "Hello World Program" which consists of printing out the text "Hello World!". Since Pyth *is* a golfing language, let's golf it! In the process, we will demonstrate some key features of Pyth. So without further ado, here is our first program:

```
"Hello World!
```

Type this into the code textbox, leave the input box empty, click the run button. The results (in the output box) should look something like this:

```
Hello World!
```

Well, that went pretty much as expected, but if you have any experience with other programming languages, you'll notice a few interesting things about the program.

1. Printing is implicit (Just name a value or identifier and it will be printed).
2. Quotes are automatically closed at the end of the program.

These features are obviously beneficially for reducing the length of your programs. Another thing you should know early on if you decide to go experimenting on your own is that programs in Pyth are typically only one line long. Statement separation is typically achieved through semicolons: `;`.

CHAPTER 2

2. More Details About Pyth

Now that you have the programming environment set up, let's learn some more about the language.

2.1 2.1. Pyth Uses Prefix Notation

Try entering `2+2`. It'll be 4 right?:

```
2
Traceback (most recent call last):
  File "safe_pyth.py", line 300, in <module>
  File "<string>", line 5, in <module>
TypeError: plus() missing 1 required positional argument: 'b'
```

Oh noes! An error! What went wrong? Well, Pyth doesn't use Infix notation (the operator goes between the operands) like most languages do. Instead, Pyth uses Prefix (aka Polish) notation, which means the operator goes *before* the operands. This has the benefit of not requiring parenthesis, making programs shorter and freeing up operators for other uses. Let's try that math problem again:

```
+2 2

Output:

4
```

Now it is working as expected! Notice the space between the `2`'s so the parser doesn't interpret them as a `22`.

2.2 2.2. Pyth Has Many, Many Operators

The addition operator we just saw doesn't even begin to scratch the surface of Pyth's rich variety of operators. As well as addition, Pyth has all the customary arithmetic operators - `-` for subtraction, `*` for multiplication, `/` for division, `%` for modulo, and `^` for exponentiation.

Integers are defined as you would expect and Floats with the decimal point. Ex:

```
0
1
18374
2.5
```

However, negative numbers do not use a unary version of the subtraction symbol since all operators have a fixed arity (more on arity later). Instead, negative numbers use the unary reversal operator: _:

```
Invalid: -25

Valid: _25
```

Pyth also has many predefined variables:

```
Z=0
T=10
k=""
d=" "
b="\n"
```

All operators, variables, and control flow keywords, are always one character long to reduce the size of the program.

2.3 2.3. All Operators in Pyth Have a Fixed Arity

The arity of an operator or function (according to Wikipedia) is *the number of arguments or operands the function or operation accepts.* Most programming languages allow functions to accept different numbers of arguments, but doing so requires parenthesis. Pyth instead has a fixed number of operands per operator, allowing it to do away with parenthesis or any other delimiter.

2.4 2.4. Operators Mean Different Things in Different Contexts

To further increase the number of functions available with only one character operators, operators operate differently depending on the values passed to it. For example, we saw the + operator adds numbers, but if + gets two sequences (e.g. strings, lists) as operands, it concatenates them:

```
+2 T -> 12 (remember that T=10)

+"Hello ""World!" -> Hello World!
```

Letting + mean both concatenation and addition is pretty common, but Pyth takes this to another level, with most operators having 2 or more meanings.

2.5 2.5. The Pyth Code is Compiled to Python Code Then Run

Pyth is technically a JIT compiled language since the Pyth code is converted to Python through a series of rules defined in the file `data.py` and then run with the variables and functions defined in `macros.py`. You can select the debug button in the online interpreter or pass the `-d` flag to the local one to see the compiled Python code. Here is what comes out as debug from `^T6` which evaluates to a million:

```
==================================================
^T6
==================================================
Pprint("\n",Ppow(T,6))
==================================================
1000000
```

As you can see, the exponentiation call is translated to a call to the function `Ppow` and is implicitly printed through a custom print function called `Pprint`. The debug option can be very helpful with seeing what at first glance looks like a bunch of random characters does.

CHAPTER 3

3. Some Simple Programs

The best way to learn Pyth is to dive head-first into it by writing some programs. We will be writing simple algorithmic exercises.

3.1 3.1. Factorial

An easy problem that will get you into thinking the correct way is computing factorials and all the different ways to do it. We will trying simple loops, the recursive approach, and the `reduce` function. Let's try it the obvious, straightforward way first. But how do we get input in Pyth?

3.1.1 3.1.1. Input in Pyth

There are many ways to get input in Pyth. The obvious one that handles all cases is the `w` function which acts exactly like Python's `raw_input`. In fact, it compiles to that. The preferred way over this is the variable `z` which is preinitialised to the input and is easier to use multiple times unlike `w`. But both it and `w` are string representations of the input and need to be evaluated with the `v` function. Hence, the most common input method is to use the `Q` variable which is preinitialised to the *evaluated* input so it can be used outright. Try using it:

```
input: 5

==================================================
*2Q
==================================================
Q=copy(literal_eval(input()))
Pprint("\n",times(2,Q))
==================================================
10
```

See? Easy.

Note: The parser checks if `Q` or `z` are used in the program and adds a line setting them to the input if they are used, as you can see from the debug output above.

3.1.2 3.1.2. For Loops In Pyth

Pyth has many of the control flow options available by Python, including `for` loops, which is what we will be using here. They are done by the `F` command and takes the name of a variable, the list to loop on, and then an infinite amount of code only bounded by a parenthesis or semicolon. Let's try looping from zero through ten:

```
================================================
FNrZTN
================================================
for N in Prange(0,T):
 Pprint("\n",N)
================================================
0
1
2
3
4
5
6
7
8
9
```

Notice that:

1. The range function is `r`.
2. It is, like Python, not inclusive of the end value (We'll have to fix that).
3. Printing is implicit so we just had to name `N` and it was surrounded by a `Pprint` call.
4. `Z` is preinitialised to `0`

Oops! We forgot that like Python, range does not include the end value. We can do +1T, but Pyth's head function, `h`, also functions as an increment. Let's try that again:

```
================================================
FNrZhTN
================================================
for N in Prange(Z,head(T)):
 Pprint("\n",N)
================================================
0
1
2
3
4
5
6
7
8
9
10
```

But we can make it shorter still. The `U` function, or "unary-range" is like Python's range with only one argument. It'll save one character by removing the need for the `Z`:

```
================================================
FNUhTN
================================================
```

```
for N in urange(head(T)):
 Pprint("\n",N)
==================================================
0
1
2
3
4
5
6
7
8
9
10
```

And even shorter. The `V` keyword is the "unary-range-loop" which does the looping through the one argument range. It uses N as the loop variable:

```
==================================================
VhTN
==================================================
for N in urange(head(T)):
 Pprint("\n",N)
==================================================
0
1
2
3
4
5
6
7
8
9
10
```

Notice the debug output for the last two were exactly the same. In fact, during preprocessing, the parser expands all occurrences of `V` to `FNU`.

Now we should be able to write an iterative factorial.

3.1.3 3.1.3. The Iterative Factorial

First, let's loop from one to the input. It should be easy, but we can't use `V` since we want our range to start from `1`, not `0`. Remember also to increment the input:

```
input: 5

==================================================
FNr1hQN
==================================================
Q=copy(literal_eval(input()))
for N in Prange(1,head(Q)):
 Pprint("\n",N)
==================================================
1
2
```

```
3
4
5
```

Now, we have to have our variable that holds the answer. Pyth has an assignment operator which works pretty much as you'd expect:

```
==================================================
=N5N
==================================================
N=copy(5)
Pprint("\n",N)
==================================================
5
```

But if you only need one variable, it's better to use `K` or `J` which don't need an equals sign to be assigned to on their first use:

```
==================================================
K5K
==================================================
K=5
Pprint("\n",K)
==================================================
5
```

Applying that to our factorial:

```
==================================================
K1FNr1hQ=K*KN
==================================================
Q=copy(literal_eval(input()))
K=1
for N in Prange(1,head(Q)):
 K=copy(times(K,N))
==================================================
```

Now we just have to print our answer which should be easy since it is implicit:

```
input: 5

==================================================
K1FNr1hQ=K*KNK
==================================================
Q=copy(literal_eval(input()))
K=1
for N in Prange(1,head(Q)):
 K=copy(times(K,N))
 Pprint("\n",K)
==================================================
1
2
6
24
120
```

Yikes! We forgot that a `for` loop's influence is infinite so the printing happens every time the loop runs. We can use a parenthesis since we only have to end one control flow, but it is better practice to use a semicolon:

```
input: 5

==================================================
K1FNr1hQ=K*KN;K
==================================================
Q=copy(literal_eval(input()))
K=1
for N in Prange(1,head(Q)):
 K=copy(times(K,N))
Pprint("\n",K)
==================================================
120
```

It works!

One final change we can make to shorten the program is to use Pyth's augmented assignment syntactic sugar. Just like Python has +=, -= and so forth, Pyth has the same constructs, except in reverse, such as =+, =-, etc. However, Pyth's augmented assignment can be used with any function, not just binary arithmetic operators. For instance, `=hK` has the same effect as `K++`. For this code, we will use `=*`:

```
input: 5

=================== 14 chars ====================
K1FNr1hQ=*KN;K
==================================================
assign('Q',eval(input()))
assign("K",1)
for N in num_to_range(Prange(1,head(Q))):
 assign('K',times(K,N))
 imp_print(K)
==================================================
120
```

3.1.4 3.1.4. User Defined Functions in Pyth

The most general way of defining functions in Pyth is with the `D` keyword. `D` works similarly to `def` in Python. To define a triple function called `h` that takes the input variable `Z`, you could write the following:

```
=================== 9 chars =====================
DhZK*3ZRK
==================================================
@memoized
def head(Z):
 K=times(3,Z)
 return K
==================================================
```

Note that `R` is the equivalent of `return`. Also, since arities in Pyth are unchangable, to define a new 1-variable function, an existing 1-variable function name must be used.

Pyth has a shorthand for function definition. They work similarly to lambdas in Python, in that there is an implicit return statement. The one var lambda uses the `L` keyword, uses the variable `b`, and defines a function named `y`.The two var lambda uses `M`, the variables `G` and `H`, and defines `g`.Here is a demonstration of a triple function:

```
==================================================
L*3b
```

```
================================================
@memoized
def subsets(b):
 return times(3,b)
================================================
```

And here's me calling it:

```
================================================
L*3by12
================================================
@memoized
def subsets(b):
 return times(3,b)
Pprint("\n",subsets(12))
================================================
36
```

Note: All functions are automatically memoized in Pyth.

3.1.5 3.1.5. The Recursive Factorial

The recursive factorial is a common solution. It works by taking the factorial of the number lower than it, recursively, until you get to zero, which returns 1. Let's first define our factorial function's base case of zero:

```
=================== 5 chars ====================
L?bT1
================================================
@memoized
def subsets(b):
 return (T if b else 1)
================================================
```

Here I'm using `T` as a placeholder for the recursive case.

Also, notice that the ternary operator `?abc` evaluates to `if a then b else c`.

Now let's complete the factorial function:

```
=================== 9 chars ====================
L?b*bytb1
================================================
@memoized
def subsets(b):
 return (times(b,subsets(tail(b))) if b else 1)
================================================
```

This uses `t`, the decrement function, to recursively call the function on the input minus 1.

Pretty simple. Now we have to take input and run the function:

```
input: 5

=================== 11 chars ===================
L?b*bytb1yQ
================================================
    assign('Q',literal_eval(input()))
```

```
    @memoized
    def subsets(b):
     return (times(b,subsets(tail(b))) if b else 1)
    imp_print(subsets(Q))
==================================================
120
```

Another factorial example...

3.1.6 3.1.6. Factorials With Reduce

The best way to do it, the way most people would do it, would be to use the reduce function. The `u` operator works exactly like Python's reduce, except for an implicit lambda so you can just write code without a lambda declaration. All a factorial is, is a reduction by the product operator on the range from 1 through n. This makes it very easy. The reduce function takes a statement of code, the sequence to iterate on, and a base case:

```
input: 5

==================================================
u*GHr1hQ1
==================================================
Q=copy(literal_eval(input()))
Pprint("\n",reduce(lambda G, H:times(G,H),Prange(1,head(Q)),1))
==================================================
120
```

As with each function in Pyth which uses an implicit lambda, reduce (`u`) has its own built in variables. In this case, the variables in question are `G`, the accumulator variable, and `H`, the sequence variable.

However, we can do better than this. If we use the list from `0` to `Q-1` instead, but multiply by one more than the sequence variable in the reduce, we can shorten the code. `UQ`, unary range of `Q` will produce the appropriate list, but `u` has a handy default where a number as the second variable will be treated identically to the unary range of that number. Thus, our code becomes:

```
input: 5

=================== 7 chars ====================
u*GhHQ1
==================================================
assign('Q',eval(input()))
imp_print(reduce(lambda G, H:times(G,head(H)),Q,1))
==================================================
120
```

Final way to calculate the factorial:

3.1.7 3.1.7. Factorial With Built-in

Pyth has a lot of specialty functions. So many, in fact, that there are too many to write them all with single character names. To remedy this, we use the `.` syntax. `.` followed by another character does something entirely different. `.!` in particular is the factorial function:

```
input: 5
```

```
=================== 3 chars ====================
.!Q
================================================
assign('Q',eval(input()))
imp_print(factorial(Q))
================================================
120
```

3.2 3.2. The First n Fibonacci numbers

The Fibonacci sequence is another subject of many programming problems. We will solve the simplest, finding the first n of them. The Fibonacci sequence is a recursive sequence where each number is the sum of the last two. It starts with 0 and 1. We'll just set the seed values and then loop through how many ever times needed, applying the rule. We will need a temp variable to store the previous value in the exchange:

```
input 10:

=================== 15 chars ====================
J1VQJKZ=ZJ=J+ZK
================================================
    assign('Q',literal_eval(input()))
    assign("J",1)
    for N in num_to_range(Q):
     imp_print(J)
     assign("K",Z)
     assign('Z',J)
     assign('J',plus(Z,K))
================================================
1
1
2
3
5
8
13
21
34
55
```

Notice that we used `Z` as one of the variables. `Z` is preinitialized to `0`, which was appropriate to use here. All of Pyth's variables have some sort of special property.

That was pretty easy, but this can be shortened (or should be) with the double-assignment operator, `A`. This has an arity of 1 and takes a tuple of two values. This shortens the assignment, but in this case we have to re-assign `H` and `G` since A implicitly uses them and their defaults are `{}` and an alphabetical string respectively. We use the `(` tuple creation operator to make the tuple:

```
input: 10

=================== 14 chars ====================
A(Z1)VQHA(H+HG
================================================
    assign('Q',literal_eval(input()))
    assign('[G,H]',Ptuple(Z,1))
    for N in num_to_range(Q):
```

```
    imp_print(H)
    assign('[G,H]',Ptuple(H,plus(H,G)))
==================================================
1
1
2
3
5
8
13
21
34
55
```

We will examine some more exercises in the next chapter.

CHAPTER 4

4. Some Simple Programs - Part II

4.1 Collatz Sequence

Another relatively simple programming problem for us to golf down is to generate Collatz sequences. The Collatz sequence consists of repeatedly applying the following procedure: If the current number is even, divide it by two. If the current number is odd, triple it and add one. Repeat until 1 is reached. Suppose we would like to print out the entire sequence, from the input to 1. Here is a very straightforward implementation, which illustrates many of Pyth's statements:

```
input: 3

==================== 23 chars ====================
WtQI%Q2=Qh*Q3).?=Q/Q2)Q
==================================================
assign('Q',Punsafe_eval(input()))
while tail(Q):
 if mod(Q,2):
  assign('Q',head(times(Q,3)))
 else:
  assign('Q',div(Q,2))
 imp_print(Q)
==================================================
10
5
16
8
4
2
1
```

Here, we use a while loop, `W`, with an if statement, `I`, and an else statement, `.?` in the body. The if and else adjust the value of the changing variable, `Q`, then the value is printed with the final `Q`. The loop condition is checked using `tQ`, which, since `t` is the decrement function, is nonzero and thus truthy whenever `Q!=1` as desired. `%Q2`, corresponding to `Q%2` in most languages, is truthy whenever `Q` is odd.

As an improvement, we can use `?`, Pyth's `if ... then ... else ...` operator. Like C, but unlike Python, the conditional goes in front, then the truthy branch, then the falsy branch. This allows us to get rid of the statement overhead and the repetition of `=Q`:

```
=================== 17 chars ====================
WtQ=Q?%Q2h*Q3/Q2Q
==================================================
assign('Q',eval(input()))
while tail(Q):
 assign('Q',(head(times(Q,3)) if mod(Q,2) else div(Q,2)))
 imp_print(Q)
==================================================
10
5
16
8
4
2
1
```

That saved 5 characters, or 23%. However, we can still do better. Pyth's assignment and lookup functions, `X` and `@` respectively, have a feature which is very handy in exactly this situation. Assignment and lookup wrap around when called on sequences, which means that if we lookup the `Qth` location of a sequence consisting of the two possible Collatz sequence successors, the proper one will be selected. Fortunately, the perfect function exists, `,`, which constructs a 2-tuple of its 2 arguments. Putting these together gives us:

```
=================== 16 chars ====================
WtQ=Q@,/Q2h*3QQQ
==================================================
Q=copy(literal_eval(input()))
while tail(Q):
 Q=copy(lookup((div(Q,2),head(times(3,Q))),Q))
 Pprint("\n",Q)
==================================================
10
5
16
8
4
2
1
```

That's as short as it can be, as far as I know.

CHAPTER 5

5. Learning More - Documentation and Errors

5.1 5.1. Documentation

Now that you've got a basic grasp of Pyth, the best way to learn more about the language is via the documentation. Pyth's documentation is located on Github, and is updated continually as the language evolves.

While the documentation is a good first step towards understanding how a function works, the only way to get a full understanding is by using it. Write programs using that function, and it'll become clear.

5.2 5.2. Errors

There are two levels of errors you might get: errors at the Pyth level, and errors at the Python level.

5.3 5.2.1 Pyth Errors

Currently, there are only 3 types of errors implemented in Pyth: token not implemented errors, unsafe input errors, and wrong type errors. I'll go through these one by one.

Token not implemented errors occur when you try to use an unimplemented token. They look like this:

```
==================== 6 chars =====================
5.@1 1
==================================================
Traceback (most recent call last):
  File "pyth.py", line 454, in <module>
    py_code_line = general_parse(code)
  File "pyth.py", line 38, in general_parse
    parsed, code = parse(code)
  File "pyth.py", line 105, in parse
    raise PythParseError(active_char, rest_code)
extra_parse.PythParseError: .@ is not implemented, 4 from the end.
```

These are most commonly caused by using non-ASCII characters in Pyth code outside of strings, and by ending floating point numeric literals with a `.`, which is confused with tokens of the form `._`, as seen above.

Unsafe input errors occur when you try to use `$` (the python code injection character) in the online compiler / executor or when the `-s` flag (safe mode) is enabled. Using `$` is a security hole, and is therefore disabled online.

Wrong type errors are the most common type of error. Most functions are defined via type overloading, where the function does something entirely different depending on what types are given as input. However, not all combinations of types have an associated functionality. For instance:

```
==================== 9 chars =====================
@"abc"1.5
==================================================
Pprint("\n",lookup("abc",1.5))
==================================================
Traceback (most recent call last):
  File "pyth.py", line 478, in <module>
  File "<string>", line 4, in <module>
  File "/app/macros.py", line 84, in lookup
macros.BadTypeCombinationError:
Error occured in function: @
Arg 1: 'abc', type str.
Arg 2: 1.5, type float.
```

The relevant part to look at is the last four lines. The fourth to last line indicates that an error as caused due to a bad type combination. The third to last line indicates that the error occurred in the `@` function, and the rest of the lines indicate that the error occurred because `@` was called with a string as its first argument and a float as its second argument, for which it has no defined functionality.

5.4 5.2.2 Python Errors

Occasionally you will get an error such as:

```
==================== 4 chars =====================
@""1
==================================================
Pprint("\n",lookup("",1))
==================================================
Traceback (most recent call last):
  File "pyth.py", line 478, in <module>
  File "<string>", line 4, in <module>
  File "/app/macros.py", line 73, in lookup
ZeroDivisionError: integer division or modulo by zero
```

that doesn't fall into any of the previous categories. At this point, the best solution is to simply try different variants on the program in an attempt to understand how it ticks. If that doesn't work, the only remaining recourse is to look at the code itself.

As indicated by the traceback, looking at line 73 of `macro.py` we see that `@` attempts to perform a lookup in the string at the index given by the second argument modulus the length of the string. Since the length of the string is zero, Python's modulus operator will fail and throw a `ZeroDivisionError`.

CHAPTER 6

6. Adding to Pyth

Pyth is an enthusiastically open source project. If you've caught a bug, thought up a cool new idea, or want to add to the project in any way, you can do so via Pyth's Github page. If you want to leave a suggestion, file a bug a report or simply get help with the language, you can open an issue by clicking on "New issue" here. If you've got some code you'd like add, you can do so via a pull request. For information on how to do so, look here.

However you end up using Pyth, I hope you enjoy it.

CHAPTER 7

7. The Language Specification - Variables

The next few chapters will be formal specifications about the language. Each chapter will have listings in alphabetical order. Since many function have completely unrelated uses, each will be listed by its primary use. This chapter is about the preinitialised variables in Pyth.

7.1 7.1. “G” - The Alphabet

This variable is preinitialised to the lowercase letters in the alphabet (i.e. “abcdefghijklmnopqrstuvwxyz”).

Ex:

```
==================================================
@G5
==================================================
Pprint("\n",lookup(G,5))
==================================================
f
```

7.2 7.2. “H” - Empty Dictionary

This variable is set to an empty Python dictionary. Pyth also has a dictionary constructor, `.d`.

Ex:

```
==================================================
XH"a"5
==================================================
Pprint("\n",assign_at(H,"a",5))
==================================================
{'a': 5}
```

7.3 7.3. “J” - Auto-Assignment With Copy

This, like `K` gets auto assigned the first time it is used. However, this is not directly assigned but assigned to a copy.

Ex:

```
==================================================
J5*3J
==================================================
J=copy(5)
Pprint("\n",times(3,J))
==================================================
15
```

7.4 7.4. “K” - Auto-Assignment

The first time time this variable is mentioned, it assigns itself to the next expression. Unlike `J`, this is not assigned to a copy but instead directly. The difference is relevant for mutable data types.

Ex:

```
==================================================
K7+TK
==================================================
K=7
Pprint("\n",plus(T,K))
==================================================
17
```

7.5 7.5. “N” - Double Quote

This is pre-set to a string containing only a double quote. This useful since its one character shorter than `\"`.

Ex:

```
==================================================
+++"Jane said "N"Hello!"N
==================================================
Pprint("\n",plus(plus(plus("Jane said ",N),"Hello!"),N))
==================================================
Jane said "Hello!"
```

7.6 7.6. “Q” - Evaluated Input

This variable auto-initializes to the evaluated input. The parser checks whether `Q` is in the code, and if it is, adds a line to the top setting `Q` equal to the evaluated input. This is the primary form of input in most programs.

Ex:

```
input: 10

==================================================
yQ
==================================================
Q=copy(literal_eval(input()))
Pprint("\n",subsets(Q))
==================================================
20
```

7.7 7.7. “T” - Ten

Pretty self-explanatory. It starts off equalling ten. Ten is a very useful value.

Ex:

```
==================================================
^T6
==================================================
Pprint("\n",Ppow(T,6))
==================================================
1000000
```

7.8 7.8. “Y” - Empty List

Just an empty list that comes in handy when appending throughout a loop.

Ex:

```
==================================================
lY
==================================================
Pprint("\n",Plen(Y))
==================================================
0
```

7.9 7.9. “Z” - Zero

This starts of as another very useful value, 0.

Ex:

```
==================================================
*Z5
==================================================
Pprint("\n",times(Z,5))
==================================================
0
```

7.10 7.10. “b” - Line Break

This is set to a newline character.

Ex:

```
==================================================
jbUT
==================================================
Pprint("\n",join(b,urange(T)))
==================================================
0
1
2
3
4
5
6
7
8
9
```

7.11 7.11. “d” - Space

This is set to a string containing a single space.

Ex:

```
==================================================
jdUT
==================================================
Pprint("\n",join(d,urange(T)))
==================================================
0 1 2 3 4 5 6 7 8 9
```

7.12 7.12. “k” - Empty String

Pre-initialised to an empty string. Useful for joining.

Ex:

```
==================================================
jkUT
==================================================
Pprint("\n",join(k,urange(T)))
==================================================
0123456789
```

7.13 7.13. “z” - Raw Input

This is set to the input, like `Q`, but not evaluated. This is useful for string input.

Ex:

```
input: Hello

==================================================
*z5
==================================================
z=copy(input())
Pprint("\n",times(z,5))
==================================================
HelloHelloHelloHelloHello
```

CHAPTER 8

8. The Language Specification - Control Flow

This section of the language specifications deals with control flow. It contains the keywords and the operators that affect which parts of the programs are run.

8.1 8.1. "#" - Exception Loop

Arity: Unbounded

This is the only form of error handling available in Pyth. It runs an infinite while loop until an error is reached, then breaks out of the loop.

Ex:

```
==================== 11 chars ====================
#/100T=T-T1
==================================================
while True:
 try:
  imp_print(div(100,T))
  assign('T',minus(T,1))
 except Exception:
  break
==================================================
10
11
12
14
16
20
25
33
50
100
```

8.2 8.2. ")" - Close Parenthesis

This ends one function or statement. Control flow like `if` or `for` all open up an unbounded arity and this closes one of them. Also useful for tuple and list constructors.

Ex:

```
=================== 28 chars ====================
I>5T"Hello")"Bye"[0 1 2 3 4)
==================================================
if gt(5,T):
 Pprint("\n","Hello")
Pprint("\n","Bye")
Pprint("\n",Plist(0,(1),(2),(3),(4)))
==================================================
Bye
[0, 1, 2, 3, 4]
```

8.3 8.3. ";" - End Statement

This is effectively an infinite amount of close parenthesis. This closes how many ever arities are needed to start completely afresh.

Ex:

```
=================== 16 chars ====================
V5I>5T[1 2;"Bye"
==================================================
for N in urange(5):
 if gt(5,T):
  Pprint("\n",Plist(1,(2)))
Pprint("\n","Bye")
==================================================
Bye
```

8.4 8.4. "B" - Break

This translates into the break keyword in Python. It is used to break out of both for and while loops (and also the infinite error loop). Pyth does not have a continue statement. Break automatically puts a close parenthesis after itself.

Ex:

```
==================================================
#ZI>ZTB~Z1
==================================================
while True:
 try:
  Pprint("\n",Z)
  if gt(Z,T):
   break
  Z+=1

 except:
```

```
  break
================================================
0
1
2
3
4
5
6
7
8
9
10
11
```

8.5 8.5. ”.?” - The Else Statement

Arity: Unbounded

This is the else part of the if-else construct. It is pretty self explanatory and works like it would in any programing language. This can also be used as part of a for-else or while-else construct. The If still needs a close parenthesis after it.

Ex:

```
================================================
I>5T"It's greater").?"It's less than"
================================================
if gt(5,T):
 Pprint("\n","It's greater")
else:
 Pprint("\n","It's less than")
================================================
It's less than
```

8.6 8.6. “F” - The For Loop

Arity: Variable, Sequence, Unbounded

This is the ubiquitous for loop. It works like it does in Python, iterating through a sequence.

Ex:

```
================================================
FNU5N
================================================
for N in urange(5):
 Pprint("\n",N)
================================================
0
1
2
3
4
```

8.7 8.7. “I” - The If Statement

Arity: Boolean, Unbounded

This is the If statement from Python. If the first argument is truthy, it executes the code, else it does nothing.

Ex:

```
==================================================
I>5T"The Universe Has Exploded"
==================================================
if gt(5,T):
 Pprint("\n","The Universe Has Exploded")
==================================================
```

8.8 8.8. “V” - Unary-Range-Loop

Arity: Integer, Unbounded

It is the shortest way to do a for loop. It is equivalent to the characters `FNU`. This makes it execute the following code a number of times equal to the input, with `N` being the loop variable. If a sequence is given as input, it is converted to an integer via its length.

Ex:

```
==================================================
VT*NN
==================================================
for N in urange(T):
 Pprint("\n",times(N,N))
==================================================
0
1
4
9
16
25
36
49
64
81
```

8.9 8.9. “W” - While Loop

Arity: Boolean, Unbounded

This the while loop construct from Python. It executes the following code until the condition becomes False.

Ex:

```
==================================================
W<lYT~Y]5;Y
==================================================
while lt(Plen(Y),T):
 Y+=[5]
```

```
Pprint("\n",Y)
================================================
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
```

8.10 8.10. & - Logical And

Arity: 2

This the logical `and` operator. It returns the first falsy value of its inputs, or the last value if all are truthy. It is shortcircuiting, just like Python's `and`.

Ex:

```
================================================
&Z1&1T
================================================
Pprint("\n",(Z and 1))
Pprint("\n",(1 and T))
================================================
0
10
```

8.11 8.11. "|" - Logical Or

Arity: 2

This is the logical `or` operator. It returns the first truthy value of the input, or the last value if all are falsy. It is shortcircuiting, just like Python's `or`.

Ex:

```
================================================
|Z1|ZZ
================================================
Pprint("\n",(Z or 1))
Pprint("\n",(Z or Z))
================================================
1
0
```

8.12 8.12. "?" - Logical If Else

Arity: 3

This is Pyth's ternary. Like most languages, but unlike Python, the conditional is the first input. The second input is executed and returned if the conditional is truthy, and the third input is executed and returned if the conditional is falsy. It is shortcircuiting, just like Python's `if else`.

Ex:

```
==================== 8 chars ====================
?T1 3?Z1 3
=================================================
Pprint("\n",(1 if T else 3))
Pprint("\n",(1 if Z else 3))
=================================================
1
3
```

CHAPTER 9

9. The Language Specification - Arithmetic

This is the first glimpse in the specification you are getting of multi-use functions. They will be organized as subsections under each section. Each subsection will be labelled by the type of arguments it receives. Here are the possible types of arguments:

```
Integer: An integer
Number: An integer or floating point decimal
String: A string of characters
List: A list of arbitrary elements
Set: A Python Set
Sequence: A list, tuple, or string
Collection: A list, tuple, string or set
Any: Anything
```

9.1 9.1. “+” - Addition

Arity: 2

9.1.1 9.1.1. Num a, Num b: Addition

This simply returns the sum of the two numbers.

Ex:

```
================================================
+2 2
================================================
Pprint("\n",plus(2,(2)))
================================================
4
```

9.1.2 9.1.2. Seq a, Any b: Concatenation

If the `b` is also a sequence of the same type as `a`, it returns the concatenation of the two sequences.

Ex:

```
=================================================
+"Hello"" World!
=================================================
Pprint("\n",plus("Hello"," World!"))
=================================================
Hello World!
```

But if `b` is not a sequence of the same type as `a`, a one-element sequence of the same type as `a` containing only `b` will be concatenated to a.

Ex:

```
=================================================
+[1 2 3)4
=================================================
Pprint("\n",plus(Plist(1,(2),(3)),4))
=================================================
[1, 2, 3, 4]
```

9.2 9.2. "*" - Multiplication

Arity: 2

9.2.1 9.2.1. Num a, Num b: Multiplication

This returns the product of the two numbers.

Ex:

```
=================================================
*2T
=================================================
Pprint("\n",times(2,T))
=================================================
20
```

9.2.2 9.2.2. Seq a, Int b: Repetition

This function repeats sequence `a`, `b` times. This is the same as repetition in Python.

Ex:

```
=================================================
*"abc"5
=================================================
Pprint("\n",times("abc",5))
=================================================
abcabcabcabcabc
```

9.2.3 9.2.3. Seq a, Seq b: Cartesian Product

Calculates the Cartesian Product of the two sequences. They have to be of the same type. This means that it generates all the possible ways that you can select one value from both sequences.

Ex:

```
==================================================
*"abc" "123"
==================================================
Pprint("\n",times("abc",("123")))
==================================================
[('a', '1'), ('a', '2'), ('a', '3'), ('b', '1'), ('b', '2'), ('b', '3'), ('c', '1'), (
↪'c', '2'), ('c', '3')]
```

9.3 9.3. “-” - Subtraction

Arity: 2

9.3.1 9.3.1. Num a, Num b: Subtraction

Computes the difference of `a` from `b`.

Ex:

```
==================================================
-T4
==================================================
Pprint("\n",minus(T,4))
==================================================
6
```

9.3.2 9.3.2. Col a, Col b: Setwise Difference

Computes the setwise difference of `a` from `b`. This means it returns a collection with the elements in `a` that are not in `b`, using the type of `a`. It preserves the order of `a`.

Ex:

```
=================== 10 chars ===================
-[1 2 3)[2
==================================================
Pprint("\n",minus(Plist(1,(2),(3)),Plist(2)))
==================================================
[1, 3]
```

9.4 9.4. “/” - Division

Arity: 2

9.4.1 9.4.1. Num a, Num b: Division

Returns `a` divided by `b`. Uses integer division which means it truncates the fractional part of the answer.

Ex:

```
================================================
/T4
================================================
Pprint("\n",div(T,4))
================================================
2
```

9.4.2 9.4.2. Seq a, any b: Count Occurrences

Returns the number of times element b appeared in sequence a.

Ex:

```
================================================
/[1 2 3 2 5)2
================================================
Pprint("\n",div(Plist(1,(2),(3),(2),(5)),2))
================================================
2
```

9.5 9.5. "%" - Modulus

Arity: 2

9.5.1 9.5.1. Num a, Num b: Modulus

Returns the remainder when `a` is integer divided by `b`.

Ex:

```
================================================
%T3
================================================
Pprint("\n",mod(T,3))
================================================
1
```

9.5.2 9.5.2. String a, Any b: String Formatting

This applies Python's string formatting that normally occurs with `%`. Requires `%s` or any of the other within the string,, just like in Python.

Ex:

```
==================================================
%"a: %d"2
==================================================
Pprint("\n",mod("a: %d",2))
==================================================
a: 2
```

9.5.3 9.5.3. Int a, Seq b: Extended Slicing

Pyth's slicing operator does not support extended slicing, so this operator has the effect of doing `b[::a]`. This means that it will pick every `a` elements of `b`.

Ex:

```
==================================================
%2"Hello
==================================================
Pprint("\n",mod(2,"Hello"))
==================================================
Hlo
```

9.6 9.6. "^" - Exponentiation

Arity: 2

9.6.1 9.6.1. Num a, Num b: Exponentiation

This raises the `a` to the power of `b`. Like Python, it allows rational exponents.

Ex:

```
==================================================
^4 2
==================================================
Pprint("\n",Ppow(4,(2)))
==================================================
16
```

9.6.2 9.6.2. Seq a, Int b: Cartesian Product With Repeats

Finds the Cartesian Product of `b` copies of sequence `a`. This means that it finds all possible sequences with length `b` that contain only elements from sequence `a`.

Ex:

```
==================================================
^"abc"3
==================================================
Pprint("\n",Ppow("abc",3))
==================================================
['aaa', 'aab', 'aac', 'aba', 'abb', 'abc', 'aca', 'acb', 'acc', 'baa', 'bab', 'bac',
↪'bba', 'bbb', 'bbc', 'bca', 'bcb', 'bcc', 'caa', 'cab', 'cac', 'cba', 'cbb', 'cbc',
↪'cca', 'ccb', 'ccc']
```

9.7 9.7. "_" - Unary Negation

Arity: 1

9.7.1 9.7.1. Num a: Negation

Returns the additive inverse of `a` or `-a`. There are no negative number literals in Pyth, this is how you define negatives in Pyth.

Ex:

```
==================================================
_25
==================================================
Pprint("\n",neg(25))
==================================================
-25
```

9.7.2 9.7.2. Seq a: Reversal

Returns `a` in reversed order. This is equivalent to the alien smiley face, `[::-1]` in Python.

Ex:

```
==================================================
_"abc"
==================================================
Pprint("\n",neg("abc"))
==================================================
cba
```

9.7.3 9.7.3. Dict a: Invert

Returns `a` with its keys and values swapped.

Ex:

```
==================== 7 chars ====================
XH1\a_H
==================================================
Pprint("\n",assign_at(H,1,"a"))
Pprint("\n",neg(H))
==================================================
{1: 'a'}
{'a': 1}
```

9.8 9.8. "P" - Prime Factorization

Arity: 1

9.8.1 9.8.1. Int a: Prime Factorization

Returns the prime factorization of `a`. Returns it as a list and multiplicities are just repeated.

Ex:

```
================================================
P12
================================================
Pprint("\n",primes_upper(12))
================================================
[2, 2, 3]
```

9.8.2 9.8.2. Seq a: All But Last

Returns all but the last element of `a`. This is equivalent to the Python `[:-1]`

Ex:

```
================================================
P"abc"
================================================
Pprint("\n",primes_upper("abc"))
================================================
ab
```

CHAPTER 10

10. The Language Specification - Comparisons

This section contains a list of comparison operators. They all take two arguments and return a boolean value depending on the relationship between the two arguments.

10.1 10.1. "<" - Is Less Than

Arity: 2

10.1.1 10.1.1. Any a, Same Type b: Less Than

If a and b are of the same type, checks if `a` is less than `b` by whatever way that type is compared (e.g. Numbers by size, strings lexicographically).

Ex:

```
================================================
<5T<T5
================================================
Pprint("\n",lt(5,T))
Pprint("\n",lt(T,5))
================================================
True
False
```

10.1.2 10.1.2. Seq a, Int b: Slice Till

Takes a slice of sequence `a` until index `b`. This is equivalent to the Python statement `a[:b]`. The slice is not inclusive of the element at index `b`

Ex:

```
<"abcdefgh"5
==================================================
Pprint("\n",lt("abcdefgh",5))
==================================================
abcde
```

10.1.3 10.1.3. Set a, Set b: Is Subset

Checks if set `a` is a subset of set `b`. This means that it checks if set `b` contains all the elements of set `a`.

Ex:

```
==================================================
<{[1 2){[1 2 3)
==================================================
Pprint("\n",lt(Pset(Plist(1,(2))),Pset(Plist(1,(2),(3)))))
==================================================
True
```

10.2 10.2. “q” - Is Equal To

Arity: 2

This checks if the two values are equal to each other. This is the same as the Python ==.

Ex:

```
==================================================
qT5qTT
==================================================
Pprint("\n",equal(T,5))
Pprint("\n",equal(T,T))
==================================================
False
True
```

10.3 10.3. “>” - Is Greater Than

Arity: 2

10.3.1 10.3.1 Any a, Same Type b: Greater Than

This checks if `a` is greater than `b`. Uses the same type of comparisons as <

Ex:

```
==================================================
>5T>T5
==================================================
Pprint("\n",gt(5,T))
Pprint("\n",gt(T,5))
```

```
==================================================
False
True
```

10.3.2 10.3.2 Seq a, Int b: Slice From

This takes a slice of sequence `a` from index `b` onwards till the end. This is equivalent to the Python `a[b:]`. The slice is inclusive of the element at index `b`.

Ex:

```
==================================================
>"abcdefgh"5
==================================================
Pprint("\n",gt("abcdefgh",5))
==================================================
fgh
```

10.3.3 10.3.3. Set a, Set b: Is Superset

Checks is set `a` is a superset of set `b`. This means that it checks if set `a` contains all the elements of set `b`. This does not return True if the two sets are equal.

Ex:

```
==================================================
>{[1 2 3){[1 2)
==================================================
Pprint("\n",gt(Pset(Plist(1,(2),(3))),Pset(Plist(1,(2)))))
==================================================
True
```

10.4 10.4. “n” - Not Equal To

Arity: 2

Checks if the two elements are not equal to each other. This is equivalent to Python’s ”!=”.

Ex:

```
==================================================
nT5nTT
==================================================
Pprint("\n",ne(T,5))
Pprint("\n",ne(T,T))
==================================================
True
False
```

10.5 10.5. “g” - Is Greater Than or Equal To

Arity: 2

10.5.1 10.5.1. Any a, Same Type b: Greater Than or Equal To

Checks if `a` is greater than or equal to `b`.

Ex:

```
================================================
gT5gTTg5T
================================================
Pprint("\n",gte(T,5))
Pprint("\n",gte(T,T))
Pprint("\n",gte(5,T))
================================================
True
True
False
```

10.6 10.5.2. Seq a, Int b: Slice From, 1-indexed

This takes a slice of `a`, from the element `b-1`. It is equivalent to `a[b-1:]`.

Ex:

```
=================== 3 chars ====================
gG5
================================================
Pprint("\n",gte(G,5))
================================================
efghijklmnopqrstuvwxyz
```

10.6.1 10.5.3. Set a, Set b: Superset or Equal

Checks if set `a` is a superset of set `b` or equal to set `b`.

Ex:

```
================================================
g{[1 2 3){[2 3)g{[1 2 3){[1 2 3)
================================================
Pprint("\n",gte(Pset(Plist(1,(2),(3))),Pset(Plist(2,(3)))))
Pprint("\n",gte(Pset(Plist(1,(2),(3))),Pset(Plist(1,(2),(3)))))
================================================
True
True
```

10.7 10.6. "}" - Contains

Arity: 2

Checks if the second argument, a collection, contains the first argument. Is equivalent to the Python `in` operator.

Ex:

```
==================================================
}\a"abc"
==================================================
Pprint("\n",("a" in "abc"))
==================================================
True
```

CHAPTER 11

11. The Language Specification - Sequences

This chapter of the specification deals with sequences. These are strings, arrays, tuples, etc. This chapter contains both sequence constructors and functions and operators that work on them.

11.1 11.1. ” - String Literal

Arity: Unbounded

Starts a string literal. Stops parsing until it reaches a matching quote or EOF.

Ex:

```
================================================
"abc""def
================================================
Pprint("\n","abc")
Pprint("\n","def")
================================================
abc
def
```

11.2 11.2. “[” - List Constructor

Arity: Unbounded

This starts a list definition. Elements are space-separated since the comma is the couple-constructor. Is ended like any other unbounded arity, with a `)`. Lists are mutable.

Ex:

```
==================== 9 chars ====================
[1 2 3)[4
```

```
==================================================
Pprint("\n",Plist(1,(2),(3)))
Pprint("\n",Plist(4))
==================================================
[1, 2, 3]
[4]
```

11.3 11.3. “(” - Tuple Contructor

Arity: Unbounded

This starts a tuple definition. It works in the same way as the list constructor. Unlike lists, tuples are immutable.

Ex:

```
==================================================
(1 2 3)
==================================================
Pprint("\n",Ptuple(1,(2),(3)))
==================================================
(1, 2, 3)
```

11.4 11.4. “{” - Set Constructor

Arity: 1

This is the Python set constructor `set()`. It takes a sequence and makes a set out of it. An important consequence for golfing is that all duplicates are removed when a set is created. On numbers it makes a one element set containing the number.

Ex:

```
==================================================
{[1 2 3){T
==================================================
Pprint("\n",Pset(Plist(1,(2),(3))))
Pprint("\n",Pset(T))
==================================================
{1, 2, 3}
{10}
```

11.5 11.5. “\” - String Escape

Arity: One Character

Creates a one character string containing the next character in the program.

Ex:

```
==================================================
\a
==================================================
```

```
Pprint("\n","a")
================================================
a
```

11.6 11.6. “]” - One Element List

Arity: 1

Makes a list containing only one element.

Ex:

```
================================================
]5
================================================
Pprint("\n",[5])
================================================
[5]
```

11.7 11.7. ”,” - Couple Constructor

Arity: 2

A couple is a tuple containing only two elements. This creates a couple containing the arguments passed to it.

Ex:

```
================================================
,5T
================================================
Pprint("\n",(5,T))
================================================
(5, 10)
```

11.8 11.8. “a” - Append

Arity: 2

Appends the second argument to the first argument, a list, by mutating the list.

Ex:

```
================================================
aY5Y
================================================
Y.append(5)
Pprint("\n",Y)
================================================
[5]
```

11.9 11.9. “c” - Chop

*Arity: 2**

11.9.1 11.9.1. Seq a, Int b: Chop

Splits sequence `a` every `b` elements.

Ex:

```
==================================================
cG2
==================================================
Pprint("\n",chop(G,2))
==================================================
['ab', 'cd', 'ef', 'gh', 'ij', 'kl', 'mn', 'op', 'qr', 'st', 'uv', 'wx', 'yz']
```

11.9.2 11.9.2. Int a, Seq b: Chop Into Nths

Splits sequence `b` into `a` pieces, distributed equally.

Ex:

```
=================== 4 chars ====================
c3UT
==================================================
Pprint("\n",chop(3,urange(T)))
==================================================
[[0, 1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

11.9.3 11.9.3. Str a, Str b: String Split

Splits string `a` by occurrences of string `b`. Uses `string.split()`. If `b` is `None` (default value) it splits by whitespace.

Ex:

```
==================================================
c"kjclkshfglasjfljdlakjaflkajflkajfalkjgaf"\a
==================================================
Pprint("\n",chop("kjclkshfglasjfljdlakjaflkajflkajfalkjgaf","a"))
==================================================
['kjclkshfgl', 'sjfljdl', 'kj', 'flk', 'jflk', 'jf', 'lkjg', 'f']
```

11.9.4 11.9.4. Num a, Num b: Float Division

Computes true division on the arguments. Does not truncate the result.

Ex:

```
==================================================
cT4
==================================================
Pprint("\n",chop(T,4))
==================================================
2.5
```

11.10 11.10. “e” - End

Arity: 1

11.10.1 11.10.1. Seq a: End

Returns the last element of sequence `a`.

Ex:

```
==================================================
e"abc"
==================================================
Pprint("\n",end("abc"))
==================================================
c
```

11.10.2 11.10.2. Num a: Modulus By Ten

Returns `a%10` which is the remainder when `a` is divided by `10`.

Ex:

```
==================================================
e25
==================================================
Pprint("\n",end(25))
==================================================
5
```

11.11 11.11. “h” - Head

Arity: 1

11.11.1 11.11.1. Seq a: Head

Returns the first element of sequence `a`.

Ex:

```
==================================================
h"abc"
==================================================
Pprint("\n",head("abc"))
==================================================
a
```

11.11.2 11.11.2. Num a: Increment

Returns `a+1`.

Ex:

```
==================================================
h5
==================================================
Pprint("\n",head(5))
==================================================
6
```

11.12 11.12. "j" - Join

Arity: 2

11.12.1 11.12.1. Str a, Seq b: Join

This works the same as the Python `string.join()`. It takes sequence b and concatenates all of its elements, separated by string a. However, unlike Python's `.join` method, it coerces the elements of the sequence to strings instead of throwing an error.

Ex:

```
==================================================
jdUT
==================================================
Pprint("\n",join(d,urange(T)))
==================================================
0 1 2 3 4 5 6 7 8 9
```

11.12.2 11.12.2. Int a, Int b: Base Conversion

This takes the integer a, and converts it into base b. It however, outputs the result in a list of digits.

Ex:

```
==================================================
jT2
==================================================
Pprint("\n",join(T,2))
==================================================
[1, 0, 1, 0]
```

11.13 11.13. “l” - Length

Arity: 1

11.13.1 11.13.1. Seq a: Length

Returns the length of sequence a. Uses Python `len()`.

Ex:

```
==================================================
lG
==================================================
Pprint("\n",Plen(G))
==================================================
26
```

11.13.2 11.13.2. Num a: Log Base 2

Calculates the logarithm in base two of a.

Ex:

```
==================================================
lT
==================================================
Pprint("\n",Plen(T))
==================================================
3.3219280948873626
```

11.14 11.14. “r” - Range

Arity: 2

11.14.1 11.14.1. Int a, Int b: Range

Returns a list containing the integers over the range `[a, b)`. Like Python, it is inclusive of `a` but not of `b`

Ex:

```
==================================================
r5T
==================================================
Pprint("\n",Prange(5,T))
==================================================
[5, 6, 7, 8, 9]
```

If the first argument is larger than the second, it returns a list containing the range `[a, b)`, in descending fashion. This is the same as Python’s `range(a, b, -1)`.

Ex:

```
=================== 3 chars ====================
rT3
================================================
Pprint("\n",Prange(T,3))
================================================
[10, 9, 8, 7, 6, 5, 4]
```

11.14.2 11.14.2. Str a, Int b: String Processing

Pyth's Range function contains a lot of string processing functions. It processes the input string `a` in various ways depending on the option provided by integer `b`.

11.14.2.1. Option 0: Lowercase

Returns the string with all letters lower-cased. Uses `str.lower()` from Python.

Ex:

```
================================================
r"HEEEllllooooooBYE"Z
================================================
Pprint("\n",Prange("HEEEllllooooooBYE",Z))
================================================
heeellllooooooobye
```

11.14.2.2. Option 1: Uppercase

Returns the string with all letters upper-cased. Uses `str.upper()` from Python.

Ex:

```
================================================
r"HEEEllllooooooBYE"1
================================================
Pprint("\n",Prange("HEEEllllooooooBYE",1))
================================================
HEEELLLLOOOOOOOBYE
```

11.14.2.3. Option 2: Swapcase

Returns the string with all the cases switched (i.e. all upper-cased become lower-cased and vice versa). Uses `str.swapcase()` from Python.

Ex:

```
================================================
r"HEEEllllooooooBYE"2
================================================
Pprint("\n",Prange("HEEEllllooooooBYE",2))
================================================
heeeLLLLOOOOOOObye
```

11.14.2.4. Option 3: Title Case

Splits the string up by all occurrences of non-alphabetical characters and capitalizes the first letter of all tokens. Internally it is `str.title()`.

Ex:

```
================================================
r"the philosopher's stone"3
================================================
Pprint("\n",Prange("the philosopher's stone",3))
================================================
The Philosopher'S Stone
```

11.14.2.5. Option 4: Capitalize:

Returns the string so that the first letter of the string is capitalized and all others are lower-cased. Uses `str.capitalize()`.

Ex:

```
================================================
r"the Philosopher's stone"4
================================================
Pprint("\n",Prange("the Philosopher's stone",4))
================================================
The philosopher's stone
```

11.14.2.6. Option 5: Capwords

This is almost the same as Option # 3 in that it tokenizes and capitalizes the first letter of each token in the string. However, it does not capitalize by all non-alphabetical but only by spaces. Uses the `capwords()` function from the `string` module.

Ex:

```
================================================
r"the philosopher's stone"5
================================================
Pprint("\n",Prange("the philosopher's stone",5))
================================================
The Philosopher's Stone
```

11.14.2.7. Option 6: Strip

This removes all whitespace from the beginning and end of the string. Note that it leaves all whitespace in the middle of the string untouched. Uses `str.strip()`.

Ex:

```
================================================
+r"          the philosopher's stone       "5G
================================================
Pprint("\n",plus(Prange("          the philosopher's stone       ",5),G))
```

```
================================================
The Philosopher's Stoneabcdefghijklmnopqrstuvwxyz
```

11.14.2.8. Option 7: Evaluate Tokens

This tokenizes the string by whitespace, then evaluates each token into a Python object.

Ex:

```
================================================
sr"1 2 3 4 5 6 7 8 9 10"7
================================================
Pprint("\n",Psum(Prange("1 2 3 4 5 6 7 8 9 10",7)))
================================================
55
```

11.15 11.15. “s” - Sum

Arity: 1

11.15.1 11.15.1. Seq a: Sum

This sums the numbers in the sequence. The base case for an empty list is `0`.

Ex:

```
================================================
sUT
================================================
Pprint("\n",Psum(urange(T)))
================================================
45
```

11.15.2 11.15.2. Str a: Int

This converts string `a` into an integer. Uses Python’s `int()` built-in.

Ex:

```
================================================
s"123
================================================
Pprint("\n",Psum("123"))
================================================
123
```

11.16 11.16. “t” - Tail

Arity: 1

11.16.1 11.16.1. Seq a: Tail

Returns all but the first element of the sequence. Equivalent to the slice `a[1:]` except that returns its input unchanged when given an empty sequence.

Ex:

```
==================================================
tG
==================================================
Pprint("\n",tail(G))
==================================================
bcdefghijklmnopqrstuvwxyz
```

11.16.2 11.16.2. Num a: Decrement

Returns `a-1`.

Ex:

```
==================================================
tT
==================================================
Pprint("\n",tail(T))
==================================================
9
```

11.17 11.17. “x” - Index

Arity: 2

11.17.1 11.17.1. Seq a, Element b: Index Of

Returns the position of the first occurrence of element b within sequence a. Returns `-1` if it is not found.

Ex:

```
==================================================
xG\f
==================================================
Pprint("\n",index(G,"f"))
==================================================
5
```

11.17.2 11.18.1. Int a, Int b: XOR

Computes the bitwise XOR of `a` and `b`. Same as Python `a^b`.

Ex:

```
==================================================
xT2
==================================================
Pprint("\n",index(T,2))
==================================================
8
```

11.18 11.18. “y” - Powerset

Arity: 1

11.18.1 11.18.1. Seq a: Powerset

Returns the powerset of sequence a. The powerset is the set of all possible sets using the elements of sequence a. However, The result is returned as a list of lists, each list in sorted order.

Ex:

```
==================================================
yU3
==================================================
Pprint("\n",subsets(urange(3)))
==================================================
[[], [0], [1], [2], [0, 1], [0, 2], [1, 2], [0, 1, 2]]
```

11.18.2 11.18.2. Num a: Double

Returns `a*2`.

Ex:

```
==================================================
yT
==================================================
Pprint("\n",subsets(T))
==================================================
20
```

11.19 11.19. “S” - Sorted

Arity: 1

Returns the input, except sorted. The same as python `sorted()`.

Ex:

```
==================================================
S"asjdasljls"
==================================================
Pprint("\n",Psorted("asjdasljls"))
```

```
==================================================
aadjjllsss
```

11.20 11.20. “U” - Unary Range

Arity: 1

11.20.1 11.20.1. Int a: Unary Range

Returns all the integers in the range `[0, a)`.It is the same as `r` with first parameter being `0`.

Ex:

```
==================================================
UT
==================================================
Pprint("\n",urange(T))
==================================================
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

11.20.2 11.20.2. Seq a: Len Unary Range

Does the same as normal unary range, except it uses `len(a)` as the parameter.

Ex:

```
==================================================
UG
==================================================
Pprint("\n",urange(G))
==================================================
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
↪ 24, 25]
```

11.21 11.21. “X” - Update Mutable

Arity: 3

11.21.1 11.21.1. Mutable Seq a, Index b, Value c: Update Mutable

This updates the mutable sequence (list or dict) by assigning the value at index b to value c. This is equivalent to the Python `a[b]=c`. This both updates the mutable and returns the updated.

Ex:

```
==================================================
XUT5Z
==================================================
Pprint("\n",assign_at(urange(T),5,Z))
```

```
==================================================
[0, 1, 2, 3, 4, 0, 6, 7, 8, 9]
```

11.21.2 11.21.2. Immutable Seq a, Index b, Value c: Replace Element

This works similarly to its effect on mutable sequences, except that a new sequence with the indexed element replaced is returned. This effect occurs in tuples and strings.

Ex:

```
=================== 5 chars ====================
XG9\0
==================================================
Pprint("\n",assign_at(G,9,"0"))
==================================================
abcdefghi0klmnopqrstuvwxyz
```

11.21.3 11.21.3. Index a, Mutable b, Value c: Augmented Update Mutable

This updates the mutable `b` by adding `c` to the element found at index `a`.

Ex:

```
=================== 12 chars ====================
J[1 2 3)X1J5
==================================================
J=copy(Plist(1,(2),(3)))
Pprint("\n",assign_at(1,J,5))
==================================================
[1, 7, 3]
```

11.21.4 11.21.4. Seq a, Seq b, Seq c: Translate

This takes `a`, looks up each of its elements in `b`, and replaces them with the element at the same location in `c`. If `c` is omitted, `b` is used in reverse instead.

Ex:

```
=================== 15 chars ====================
X"Hello""el""tu
==================================================
Pprint("\n",assign_at("Hello","el","tu"))
==================================================
Htuuo
```

CHAPTER 12

Indices and tables

- genindex
- modindex
- search