Python

Introduction:

Everyone knows and loves Python. It is certainly one of the most versatile languages to exist in recent times. Python is, of course, meant to be an easy-to-learn, and easily-to-understand language. Although it is one of the slowest languages out there, it is still ideal for automation of complex maths and algorithms because of its wonderful libraries and large community. It is also ideal for scripting, as it allows for platform-independent API calls for accessing system resources and kernel calls.

Invoking the Interpreter:

We can invoke the Python interpreter using python -c command [arg] …, which will execute the statements in a command, analogous to the shell’s -c option. Since Python statements often contain spaces of other characters that are interpreted by the shell, it is advised that you surround the command in quotes. We can run Python modules as scripts using python -m module [arg] … which executes the source file for module as if you had spelled out its full name on the CLI. When a script is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing -i before the script.

Argument Passing:

When known to the interpreter, the script name and additonal arguments thereafter are turned into a list of string and assigned to the argv variable, which is available in the sys module. We can access this list by importing the sys module (import sys). When the script name is given as ‘-’ (meaning standard input), sys.argv[0] is set to ‘-’. When the *-c command* is used, sys.argv[0] is set to -c. When -m is used, sys.argv[0] is set to the full name of the located module. Options after the -c or -m module are not consumed by the Python interpreter’s option processing, but left in sys.argv for the command or module to handle.

Interactive Mode:

When commands are read from a TTY, the interpreter is said to be in interactive mode. In this mode, it prompts for the next command with the primary prompt, usually three greater-than signs (>>>). For continuation lines it prompts with the secondary prompt (…). Continuation lines would be prompted for multi-line statements, such as if statements or for loops.

Comments:

Single-line comments in Python begin with the hash character (#). Python seems to have a weird obsession with requiring 2 spaces after a statement before beginning an in-line comment.

Arithmetic:

Python supports all of the arithmetic operators that you’re used to (+, -, \*, /, %). Additionally, it includes // for floor divisions and \*\* for powers. In interactive mode, the last printed expression is assigned to the variable \_. This allows you to reuse the output of a previous operation within the next operation. This is a read-only variable, and attempting to write to it would just create an independent local variable of the same name that would mask the built-in variable. Python even has built-in support for complex numbers, using the ‘j’ or ‘J’ suffix to indicate the imaginary part e.g., 3+5j.

Strings:

Strings may either be expressed using single or double quotes. The backslash can escape either of these. In order to remove the special meaning of escaped characters, we can prepend ‘r’ before the actual string literal e.g. print(r’C:\some\name’), which would print C:\some\name, rather than interpreting the \n as a newline. Similar to Kotlin, multi-line strings can be created using three double quotes at the beginning and end of the string e.g.:

print(“””\  
Usage: thingy [OPTIONS]  
 -h Display this usage message

-H hostname Hostname to connect to

“””)  
  
Note the backslash character, which prevents the default end of line from being included in the string.

Strings can be concatonated with the ‘+’ operator, and repeated with the ‘\*’ operator e.g. print(3 \* ‘un’ + ‘ium’) will print unununium. Similar to C or many other languages, two string literals which are placed beside each other are automatically concatenated: ‘Py’ ‘thon’ prints Python. This only works with string literals, of course, and not between variables.

Strings can be indexed, as you might expect. Something unique about Python is that we can provide negative indices, akin to R, where indexing will begin from the final element. Since -0 would not make much sense, the final index is accessed using -1.

We can slice strings (i.e., create substrings) using a special range operator (:). For example, if we have a variable called word, which contains the string literal ‘Python’, then we can print the first two characters like so: print(word[0:2]). Alternatively, omitting the index before or after the range operator will implicitly refer to indices 0 and -1 respectively. The same output as the prior example could be achieved like so: print(word[:2]). We can also append two substrings, of course: print(word[:2] + word[2:]) prints ‘Python’. Something a little strange is that Python will throw an exception if we attempt to access an index that is out of bounds within a normal string e.g., word[42] results in an IndexError, however, string slices are handled gracefully e.g., word[42:] prints an empty string.

Python strings are immutable, meaning we cannot change them. A new string must be constructed.

Lists:

Lists function similar to JavaScript in Python. To declare an empty list, we simply assign an empty index operator to a variable e.g., mylist = []. Note that list is a keyword in Python, so it is best not to name your list ‘list’. Lists can also be indexed and sliced, exactly like strings. All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list: mylist[:].

Unlike strings, lists are mutable. You can change elements via the index operator, and append to the list using append(). The len() function (which also applies to strings) returns the number of elements in the list. Nesting lists is easily accomplished:

>>> a = [ ‘a’, ‘b’, ‘c’ ]

>>> n = [ 1, 2, 3 ]

>>> x = [ a, n ]

>>> x

[[‘a’, ‘b’, ‘c’], [1, 2, 3]]

>>> x[0]

[‘a’, ‘b’, ‘c’]

>>> x[0][1]

‘b’

Also note in this example that we print the contents of x by simply entering it on its own line, similar to R.

Fibonacci Sequence:

The classic fibonacci sequence example can be programmed very easily in Python:

a, b = 0, 1

while a < 10:

print(a)

a, b = b, a+b

This may be quite confusing for those of you who are not familiar with multiple assignment, but perhaps as soon as I mentioned that this is using multiple assignment the example became much more clear. If you’re not familiar with multiple assignment, it is the process of assigning multiple variables their values within the same line of code. The statement a, b = 0, 1 assigns 0 to a and 1 to b. Similarly, the statement a, b = b, a+b, assigns the value of b to a and the value of a+b to b. Also, in case you weren’t familiar, Python is one of the few languages to not use braces for scope. Instead, it uses whitespace to deduce which statements belong within a scope. According to the Python specification, tabstops should be multiples of 4. Each additional indent dictates a new scope.

In this example, the print function implicitly determines that it should append a newline character to the variable a. In order to change the string that Python appends, we can supply an optional argument to the print function like so: print(a, end=’,’). This will make the fibonacci sequence print out as a comma separated list.

If Statements:

If statements follow the structure if, elif, else, akin to shell scripting. There are actually no switch statements in Python. Instead, we just use multiple elif statements to catch each branch condition.

For Loops:

Technically, there is no such thing as a standard for loop in Python. Every for loop is really just a for-each loop. If we have no collection to iterate over, we instead create a temporary range object using Python’s built-in range() function and iterate over that. For example, if we need to loop 10 times, we can do something like:

for idx in range(0, 10):

…

Alternatively, we could’ve just used range(10). Note that in the function override with two parameters, the initial index is inclusive, but the second index is exlusive. In the function override with only one parameter, the index is exclusive and implicitly begins at 0. We can also provide an optional third parameter to range(), which defines the step size:

mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for i in range(0, len(mylist), 2):

print(mylist[i], end=’,’)

Output: 1, 3, 5, 7, 9

To go backwards is a little more confusing:

mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for i in range(-2, -len(mylist)-2, -2):

print(mylist[i], end=’,’)

Recall that the index -1 is the last element in the list (10 in this case). Since we want to start at 9, we actually need to begin indexing at -2. Also recall that the second parameter for range() is exclusive. -len(mylist) would be the element 2 (index 1), so we need to subtract 1 to get to index 0, but then we need to subtract one again since we want the 0th element to be included in the range. Hopefully the step being -2 makes enough sense.

If the statement within our for loop is simple, we can reduce this code by using a slightly different syntax:

mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

[print(i) for i in mylist]

We can also do cool things like use the map() function which will create a shallow copy of our list and then apply a lambda expression on each element in the copy:

transformed = list(map(lambda x: x \* 2, mylist))

Note in this example that we cast the map back into a list using the list() function. We’ll talk about casting in a bit, but generally, you just wrap the statement within the type you want to cast to e.g., int(4.5 / 86.3).

Finally, let us discuss the enumerate() function. enumerate() is essentially Python’s answer to iterators. Enumeration is a bit different than a standard for loop. It returns a <K, V> pair with the index followed by the element in the collection:

for number in enumerate(mylist):

print(number, end=’,’)

Output: (0, 1), (1, 2), (2, 3) …  
  
If you require access to the index within the scope of your loop, simply supply the for loop with two arguments:

for idx, number in enumerate(mylist):

print(idx, number, end=’,’)  
  
Output: 0 1, 1 2, 2 3 …

break, continue, and pass:

The break and continue keywords carry on their conventional meanings from most other languages. The break keyword exits early out of the first enclosing loop that it finds, if any. The continue keyword returns early from the current iteration and begins the following iteration of the first enclosing loop that it finds, if any. Pass is a unique keyword for Python, however. In C, we can represent a no-op (no operation) with two semicolons (;;). This is effectively what pass does in Python. It simply means “don’t do anything”. In some sense it can also act a little bit like the todo! macro in Rust. If you just want to declare a function or class but not actually define them just yet, you can add a pass statement and come back to it later.

The match statement:

The match statement in Python is a lot like switch statements in other languages, except more powerful. Rather than only being able to evaluate the equality of a primitive type, it can also evaluate whether a statement or complex type matches. Here is an example of matching on a tuple containing the values x and y:

point = (0, 4)

match point:

case (0, 0):

print("Origin")

case (0, y):

print(f"Y={y}")

case (x, 0):

print(f"X={x}")

case (x, y):

print(f"X={x}, Y={y}")

case \_:

raise ValueError("Not a point")

This example is actually not as simple as you may initially presume. In the first case, we match against the tuple (0, 0), so that if the variable point has a value of 0 for x and for y, it will be considered a match (simple enough). For the next two patterns, however, they first check if the literal 0 matches, with the specified coordinate, and if they do, the variable in the case statement will get bound to the corresponding value of point. For example, if point is (0, 4), then the second case statement will apply, since point contains the literal 0 for x, which matches that case statement, and then y will be assigned the value 4 since it essentially acts as a wildcard. Thus this will print out Y=4. As I’m sure most intermediate programmers will understand, the underscore as the last statement represents the default case. In this example, we throw an error since a valid tuple would have to match with one of the 4 cases prior.

Perhaps even more interesting is Python’s ability to match objects based upon 0 or more attributes/properties. Here is the same example as previously demonstrated, but this time using an instance of a Point object rather than a tuple:

class Point:

x: int

y: int

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def where\_is(point):

match point:

case Point(x=0, y=0):

print("Origin")

case Point(x=0, y=y):

print(f"Y={y}")

case Point(x=x, y=0):

print(f"X={x}")

case Point():

print("Somewhere else")

case \_:

print("Not a point")

point = Point(0, 4)

where\_is(point)

This example will print out the same results as last time. This code may or may not make sense to you depending upon your experience. If it doesn’t make sense, understand that the \_\_init\_\_() function is a special predefined function in Python that allows us to construct a new instance of the class Point. It implicitly accepts a reference to self even though we never pass it into the constructor as an argument. Another thing to note here is that we declare the attributes x and y as type int. Remember though, Python is not typesafe and these values can easily be changed at runtime. They are mostly just a hint to both the programmer and the interpreter, but by no means do they prevent the programmer from reinterpreting those variables as a completely different type.