PYTHON FUNDAMENTALS

TROY P. KLING

Contents

1. Syntax

Python uses whitespace to delimit control flow blocks, like if statements and functions. Instead of punctuation (e.g. curly braces) or keywords (e.g. start and end), it makes use of indentation to show the structure of the program. Unlike programming languages like Java and C, where whitespace is mostly ignored, the functionality of a Python program depends on correct whitespace usage. This makes Python programs easily-recognizable by their consistent format.

Python also uses dynamic typing, which means that one does not have to explicitly state the data type of a newly-declared variable. It is syntactically accurate to write `x=3.14159` rather than something like `float x=3.14159`, and `s=‘‘Python!’’` rather than `string s=‘‘Python!’’`

2. Operations

The first step towards understanding a new programming language is getting a firm grasp on the syntax and operators. Thankfully, Python's syntax is simple and intuitive. The following table lists the most common binary (and unary) operations in Python. Most of them are straightforward, but a few are a bit unusual.

	Operation	**Syntax**	**Result**
Arithmetic	Addition	5 + 2	7
	Subtraction	5 - 2	3
	Multiplication	5 * 2	10
	Division	5 / 2	2.5
	Integer division	5 // 2	2
	Remainder	5 % 2	1
	Exponentiation	5 ** 2	25
Comparison	Equal to	5 == 4	False
	Not equal to	"Me" != "Me"	False
	Greater than	10 > 1.5	True
	Less than	$-1 < -2$	False
	Greater than or equal to	6 >= 6	True
	Less than or equal to	"a" <= "b"	True
Logical	Logical AND	True and False	False
	Logical OR	True or False	True
	Logical NOT	not True	False
Strings	Concatenation	"foo" + "bar"	"foobar"
	Repetition	"ho" * 3	"hohoho"
	Index	x="aBcDe"; x[0]	"a"
	Slice	x="aBcDe"; x[1:3]	"Bc"
	Membership	x="aBcDe"; "C" in x	False
Lists	Concatenation	[1,2,3] + [4,5,6]	[1,2,3,4,5,6]
	Repetition	[1,2,3] * 2	[1,2,3,1,2,3]
	Index	x=[5,6,7,8]; x[0]	5
	Slice	x=[5,6,7,8]; x[1:3]	[6,7]
	Membership	x=[5,6,7,8]; 6 in x	True

3. Lists

Lists are Python's version of arrays. They have no predetermined size, so they can be updated dynamically without having to worry about memory allocation. It is also worth noting that lists are zero-based in Python. Below is some example code showing basic list functionality.

```
>>> list1 = [1.414, 2.718, 3.141]
>>> list2 = [1, 2, 3, 2]
>>> len(list1)
3
>>> len(list2)
4
>>> list1.extend(list2)
>>> list1
[1.414, 2.718, 3.141, 1, 2, 3, 2]
>>> list1.append(2)
>>> list1
[1.414, 2.718, 3.141, 1, 2, 3, 2, 2]
>>> len(list1)
8
>>> min(list1)
1
>>> max(list1)
3.141
>>> list1.count(2)
3
>>> list1.index(2)
4
>>> list1.remove(2)
>>> list1
[1.414, 2.718, 3.141, 1, 3, 2, 2]
>>> list1.insert(4, 2)
>>> list1
[1.414, 2.718, 3.141, 1, 2, 3, 2, 2]
>>> list1.sort()
>>> list1
[1, 1.414, 2, 2, 2, 2.718, 3, 3.141]
>>> list1.reverse()
>>> list1
[3.141, 3, 2.718, 2, 2, 2, 1.414, 1]
```

The following table explains the purpose of each of the functions used above.

Function	**Purpose**
len(list)	Returns the length of the list.
max(list)	Returns the maximum element in the list.
min(list)	Returns the minimum element in the list.
list.append(x)	Adds a single element to the end of the list.
list.extend(list2)	Adds all elements of another list to the end of the list.
list.insert(i, x)	Inserts element x into position i in the list.
list.remove(x)	Removes the first occurrence of x in the list.
list.index(x)	Returns the index of the first occurrence of x in the list.
list.count(x)	Returns the number of times x occurs in the list.
list.sort()	Sorts the list.
list.reverse()	Reverses the list.

4. Strings

In Python, strings are essentially lists of characters. Because of this, some of their functionality is quite similar to lists. However, there are many built-in operations that can be performed on strings that cannot be performed on lists. Consider the following example code.

```
>>> str1 = ‘‘University of’’
>>> str2 = ‘‘North Carolina’’
>>> str3 = ‘‘Wilmington’’
>>> str4 = str1 + ‘‘ ’’ + str2 + ‘‘ ’’ + str3
>>> print(str4)
University of North Carolina Wilmington
>>> len(str4)
39
>>> str1.lower()
‘‘university of’’
>>> str3.upper()
‘‘WILMINGTON’’
>>> str2.index(‘‘a’’)
7
>>> str4.split(‘‘ ’’)
[‘‘University’’, ‘‘of’’, ‘‘North’’, ‘‘Carolina’’, ‘‘Wilmington’’]
>>> ‘‘     space     ’’.strip()
‘‘space’’
```

The following table explains the purpose of each of the functions used above.

Function	**Purpose**
len(str)	Returns the length of the string.
max(str)	Returns the maximum element (ASCII code) in the list.
min(str)	Returns the minimum element (ASCII code) in the list.
str.index(x)	Returns the index of the first occurrence of x in the string.
str.lower()	Converts all characters in a string to lowercase.
str.upper()	Converts all characters in a string to uppercase.
str.split(delimiter)	Splits a string along the given delimiter, returning a list.
str.strip()	Removes all left and right-padding whitespace from a string.

5. Functions

As mentioned in the section on syntax, functions in Python are not delimited by curly braces or other punctuation. Instead, they are defined by the way they are indented. Below is a boilerplate Python function.

```
def name(param1, param2=False):
    # Body of the function.
    return val1, val2
```

Note that functions can take multiple parameters and, interestingly, return multiple values, regardless of data type. To call this function, one would write the following.

```
val1, val2 = name(param1, param2=True)
```

In this example, the only required argument is `param1`. If a value for `param2` is not supplied, the default value of `False` (from the function definition) will be used. Arguments that do not have default values are called positional; otherwise, they are called keyword arguments.

Functions make use of pass by reference, rather than pass by value. This means that whenever a function modifies a variable, that change is reflected in the original scope as well, regardless of whether or not the variable is returned by the function.

6. Control Flow

In addition to function calls, there are the usual control flow structures available in Python - if statements, for loops, etc. If statements allow a program to branch based on a given condition. The three keywords for an if statement are: `if`, `elif`, `else`.

```
x = input(‘‘Enter a number: ’’)
if x < 0:
    print(‘‘Negative.’’)
elif x == 0:
    print(‘‘Zero.’’)
else:
    print(‘‘Positive.’’)
```

Loops in Python are somewhat more unusual. They often make use of the `range()` function, which produces a range of values, usually from 0 to the specified value.

```
list = []
for i in range(10):
    list.append(2*i)
```

The above code produces the list `[0,2,4,6,8,10,12,14,16,18]`. Note that using `range(10)` makes the index variable go from 0 to 9, not 0 to 10. Therefore, 18 is the largest element in the resulting list. We can sum up values in a list by executing the following code.

```
sum = 0
for i in range(len(list)):
    sum += list[i]
```

There are two things to note here. First, using `range(len(list))` is quite common – if the length of a list is variable, using `range(len(list))` will ensure it iterates through all the elements. Second, the use of the `+=` operator is new. `sum += list[i]` is shorthard for `sum = sum + list[i]` and is very common in Python. All arithmetic operators on the first page can be modified in this way.

One can also iterate through a list in Python by directly referencing the contents of the list, rather than using an index variable to do so. Consider the following code.

```
names = [‘‘Kevin’’, ‘‘Morgan’’, ‘‘Garrett’’, ‘‘Catherine’’,
         ‘‘Jon’’, ‘‘Diana’’, ‘‘Ben’’, ‘‘Katie’’]
names.sort()
for name in names:
    print(name)
```

This code sorts the eight names and prints them out in alphabetical order. Note that no index variable is used in the loop; instead, the names themselves are explicitly referenced.

The last major topic related to iteration to be covered is list comprehension. This is a shorthand technique for building lists which bypasses the need to write an entire loop. Consider the problem of computing all factors (aside from 1 and itself) of a given integer. It would be simple enough to write a loop for this, as shown for the number 48 below.

```
x = 48
factors = []
for y in range(2, x):
    if x%y == 0:
        factors.append(y)
```

This works for all values of `x`, but for `x=48` in particular, the list `[2,3,4,6,8,12,16,24]` is produced. Clearly, this is all the non-trivial factors of 48. The interesting question is whether or not there is a shorter way to write this in Python, and the answer (yes!) relies on using list comprehension.

```
x = 48
factors = [y for y in range(2,x) if x%y == 0]
```

This single line of code produces exactly the same output as the loop above. Converting simple loops into a single list comprehension is often straightforward and elegant.

Challenge: How would you find all of the prime numbers less than 100 using a single line of code? (*Hint:* The list comprehension above will be helpful).

Two final keywords worthy of being mentioned are `break` and `continue`. When placed in a loop, a `break` statement will force the loop to preemptively terminate. The `continue` statement, on the other hand, will simply skip the current iteration of the loop and continue on.

7. Conclusion

This is all you need to get started in Python! There are many external packages available, which can be taken advantage of by using the `import` statement. We will be using numpy, matplotlib, skimage, and some others.