Introduction to Python
Session 1

Robert Palmere

2021

Topics:

- Types of Variables in Python
- Iterables
- Flow Control
- Methods
- Arguments and Keyword Arguments

Recommendations:

- Take breaks
- Errors are opportunities for learning

Python is a powerful and high-level programming language. Python is "high-level" in that it abstracts away from the details that would otherwise be required by the developer to understand and implement (e.g. memory management). The power of Python is the ease at which code can be written, interpreted, debugged, and deployed. Although mid-level programming languages such as C/C++ offer faster speeds, they require a higher degree of book-keeping, library management, complex syntax, and handling of memory allocations which results in slower programming and increased time to deployment.

This document serves as a guide for its associated Jupyter Notebook.

Types of Variables in Python

To begin we can print text to the standard output device (screen) by calling the print function.

```
print(''Hello World!'')
```

This will print 'Hello World!' to the screen within the terminal once ran. We can also annotate our code using comments. A single line of comments can be made by placing the # symbol before the text. If we have a comment block, we can use multiple apostrophes.

```
# print(''Hello World!'')

'''
comment
block
'''
```

As we saw from the print function, functions are *called* using () after the name of the function. *Argument(s)* are then passed to the function to *return* or produce an output after computations within the function scope using the provided argument. In the first code snippet, we *called* the print function and passed a string literal 'Hello World!' as the argument.

We can also assign the string, and other types, to variables.

```
x = 'Hello World!'
print(type(x))
```

The above code will print the data type of the variable, x, which we assigned as the string literal 'Hello World!'. Notice that ' ' and " " are interchangeable.

A *string* is just the text data type in Python. There are also numeric types including *int*, *float*, and *complex*. Each of these data types is actually a *metaclass* as will be discussed in future sessions, and displayed in the following code:

```
print(x, type(x))
```

Iterables

The variable, x, can be *cast* as other types as well including the range type:

```
x = range(6)
print(x, type(x).__name__)
```

We have changed x to be the range data type which is a *generator* of integers from 0 - 5. As a generator, we can iterate though this set of integers using the start, step, and, stop *attributes* of the range type.

```
x.start # Start at 0 by default
x.step  # Step size of 1 by default
x.stop  # Stop at index 6 (which is the value 5)
```

The range type supports indexing / slicing as a way for accessing values. Since range is an *iterable* we can access each element by its position within the iterable (indexing), or a subset of the elements by their indices (slicing).

```
x[0]     # Index at position 0 returns 0
x[1]     # Index at position 1 returns 1
x[-1]    # Index at position -1 returns the last element
x[0:6:2] # Slicing to get the subset off even numbers with syntax [start : stop : step]
```

Strings are also iterable as a series of characters. As such, they can be indexed and sliced.

Now we will look at *container* types in Python. The core built-in container types of Python include list, tuple, sets, and dictionaries. Four qualities differentiate these containers:

1. Order
2. Mutability
3. Uniqueness of elements
4. Data types of elements

Lists are ordered and mutable. This means that we can use indexing / slicing to acquire specific elements of our list. They are mutable in that we can remove, insert, append, and alter values within the list. The elements of a list do not have to be unique and we can include most, if not all, data types within them. A list is *declared* using [] symbols or via the list() function as shown in the following example:

```
x = []
x = list()
```

Lists contain methods which we can access using the . operator in order to modify the list.

```
x = [1, 2.2, 'Hello World'] # list containing int, float, and str types.
x.pop(0)                    # access pop() method to remove element at index 0
del x[0]                    # del keyword to remove first element of x
x.insert(0, 1)              # prepend the value, 1, to the list
x.append(4+2j)              # append the complex value, 4+2i, to the list
```

Note that the pop() method returns the removed element while *del* keyword deletes the element without returning the removed element. The list() function may be useful in cases where we wish to cast elements of an iterable as a list.

```
x = list('Hello World')
x = list(range(6))
```

Tuples are similar to lists but are **not** mutable. This results in lower memory requirement for the declaration of a tuple than a list. Under the hood, lists require pointers to memory addresses as they are dynamic and over-allocate to efficiently append values to the list. Since tuples a stored in a single block of memory, generating a tuple is faster than a list which results in a faster program. Therefore, if we need a container similar to a list, but without needing to modify the contents of the container, it is often preferable to use a tuple over a list.

Tuples are declared using () or by use of the tuple() function.

```
x = ()
x = tuple()
```

Unlike lists and tuples, sets are not ordered, and therefore, specific elements cannot be accessed by their index or keys. Following the definition of a set from mathematics, sets require that each element be a unique value although these values can be of different data types. Sets are declared with or by using the set() function.

```
x = {1, 2, 3}
x = set('123')
```

There are also methods which sets have, pertaining to set operations, that lists and tuples do not.

```
x = set(range(6))
y = set(range(10))
x.union(y)        # unify elements between x and y
x.intersection(y) # return a set containing elements that both x and y share
x.difference(y)   # x is a subset of y so there is no difference
y.difference(x)   # y is a superset of x and returns a set of the difference
```

In a similar relationship shared between list and tuple types, sets have an immutable variant called *frozenset*.

Dictionaries are the final type of container we will describe. Although a dictionary is iterable, they are not sequeences which can be index by a range of integer values. Dictionaries are instead indexed by *keys* which can be any immutable type. The key is paired with a value to be returned at the specified key. Dictionaries are declared using or the dict() function. Each key-value pair is an element of the dictionary (comma separated) with key and value separated by :.

```
d = {1 : 'a', 2 : 'b', 3 : 'c'} # declare a dictionary
print(d[1])                     # prints 'a' since 1 is the key to return 'a'
d.keys()                        # returns available keys of the dictionary
d.values()                      # returns available values of the dictionary
d.items()                       # returns all key-value pairings
```

Control Flow

Now that we have an understanding of some of the data types and how containers work in Python, we can now move to *control flow*. Control flow is the ability for a developer to place statements and function calls to dictate when and how they are executed / evaluated in the program.

Before moving flow of control statments in Python, we will introduce another type, which is the boolean. Boolean values are binary meaning that they can either take on a value of True or False (1 or 0). If we cast an integer as a boolean using the bool() function, we will return either True or False depending on the value of the integer. Values greater than 0 return True while 0 gives False. In cases where the argument is not an integer, such as a string, the bool() function will return True if the value exists and has creater than 0 characters.

```
x = 1
bool(x) # returns True
x = 0
bool(x) # returns False
x = 'Hello World'
bool(x) # returns True
x = ''
bool(x) # returns False
```

Bool types are useful for conditional statements (if / else) and determining break points in for and while loops.

A *for* loop is a control flow statment which is used for iterating over a sequence (iterable). Using a for loop, we are able to execute code for however many iterations are available in the target iterable. For example,

```
x = list(range(6)) # generate a list of integers 0 - 5
for i in x:
print(i)       # prints each element of x on separate lines
```

In the above example, we see the syntax required by for loops (for <element>in <iterable>: <code to be executed>). Since there are 6 elements in the list, x, the print() function is called 6 times. At each iteration, the element is defined as the variable, i.

An common use case for a for loop is generating a list from another iterable.

```
myList = []          # declare and empty list
for i in x:
myList.append(i) # append each element, i, to myList
print(myList)        # print the resulting list
```

Notice that we defined the *scope* of the for loop using indentation. The code that is indented beneath the for loop is the code which is repeated by the for loop and therefore defines the scope of the for loop. To consolidate this code further we can generate the same list using *list comprehension* in Python. This is a way of generating a new list from an iterable without first declaring an empty list and then accessing the append method of the list.

```
myList = [i for i in x] # declare myList as a list with each element of x
print(myList)           # print the resulting list
```

The syntax of list comprehension is [<return value>for <element>in <iterable>].

For loops can also be *nested* which means that one for loop is in the scope of another for loop.

```
myList = []
for i in x:
for j in x:
myList.append((i, j))
print(myList)
```

In the above code, for every ith element, we iterate over all elements of x. Again, we can use list comprehension to shorten this nested for loop to a single line.

```
myList = [(i, j) for i in x for j in x]
```

Notice that the outer for loop comes before the nested for loop.

Adding to our control flow repertoire are conditional statements. Conditional statements provide a way for code to be executed depending on the state of other variables in your code. If / elif / else statments are the only built-in conditional statements in Python unlike C++ which also has the *switch* statement. However, a function can be easily written with the use of a dictionary to mimic switch-case behavior found in C++.

```
x = bool(input())    # ask user for input
if x == True:  # if x is True, proceed to the associated scope
print('True')
else:          # otherwise print 'False'
print('False')
```

Refer to the Jupyter notebook for more examples of relational operators (e.g. ==).

Unlike for loops, which move through each element and stop at the end of an iterable, while loops require a boolean. The code within the scope of the while loop will continue to be ran as long as the boolean remains constant. If the boolean changes from True to False or False to True, the while loop ends.

```
x = 0
b = True
while b == True:
x += 1
print(x, end='\r')
if x == 10_000:
b = False
```

In the above code, we declare x and b to be the integer 0 and bool True, respectively. Since b is now True, the code within the while loop is ran repeatedly adding 1 to x at each iteration. If b is True and x has reached 10,000, the while loop will end since b will be set to False before the next iteration can occur.

Functions / Methods

Functions or *methods*, in Python, are self contained portions of code which accomplish a specific task as defined by the programmer. We have already seen how a funnction works via type casting and the print() function. Arguments may be passed to the method, the method performs a set of actions on or with use of the argument(s) if an argument is passed, and the method may or may not return a value which can be assigned to a variable.

```
def add(x, y):    # Method declaration
return x + y  # Return statement

add(1, 2)         # Returns 3
```

We declare a function / method using the *def* keyword followed by the function name and variables which serve as place holders for arguments to be passed (i.e. def <function name>(<variables>): <code><return statement (optional)>). Although we will not review argument passing models here, it should be noted that Python uses a passes arguments to functions / methods by *object reference*.

Small functions such as the add() function defined above may be better suited as a *lambda* expression. These are single-lined, anonymous functions. They are useful for decluttering code and for use along with other, higher-order functions, that take lambda expressions as arguments. The syntax for a lambda: lambda <arguments>: <return value>

```
add = lambda a, b : a + b   # declare the lambda function
add(1, 1) # returns 2

(lambda a, b : a + b)(1, 1) # returns 2
```

Python also makes the distinction between arguments and keyword arguments (*args and **kwargs). Generic arguments and arguments which contain key-value pairings (dictionaries) can be separated by specifying '*args' and '**kwargs' as the two arguments in the function definition. This is useful for handling these argument types separately, say, if we wanted to perform actions on our *args using options handled by **kwargs. The important part is the use of the * and ** notation as 'args' and 'kwargs' are just conventional names given to these variables.