Jacob's Python Primer

Python programs are typically named with the extension `.py`. The code contained in these files can be executed using the command `python`. A python file that is directly executed is typically referred to as a "script." Other python files are referred to as "libraries" or "modules." These files are utilized by loading (a.k.a. "importing") into a script file. A script example:

File `hello_world.py`:

```
print("Hello, world!")
```

Command line:

```
$ python hello_world.py
Hello, world!"
$
```

Comments

Any line or part of a line in a `.py` file that starts with a `#` will be ignored by the python function. This allows you to annotate or "comment" your code.

Variables and datatypes

Variables are values that you have given a name to. Variable names can be any string of alphanumeric characters (plus the "_" character) as long as that string of characters does not start with a number. Valid variable names: `a`, `var1`, `my_favorite_variable`. Invalid variable names: `2aa`, `"variable"`, `!num`.

NOTE: You should never use variable names that are the same as the names of built-in python functions! Python will allow you to do this, but it will result in broken built-in functions! You can find the list of all Python built in functions here.

Datatypes are coding objects that can be used to store data in various forms and are the main things you will be assigning to variables. The main data types are as follows:

```
NAME        BUILT-IN      EXAMPLE
Integer     int()         1, 70001
Float       float()       0.54456, 1.23e-5, 1.0
String      str()         'abcdefg', "My string!"
List        list()        [1, 2, 3, 'a', 'b', 'c']
Tuple       tuple()       (1.0, 'abc', 1e5)
Dict        dict()        {'a': 42, 'b': "value"}
Set         set()         {1, 2, "element", 314.0e-2}
Bool        bool()         True, False
```

Some of these data types take on only a single value (int and float) while the remainder can contain multiple values (referred to as 'iterables'). Python does not require you to specify the datatype of a variable, instead it is inferred from the syntax of your code (based on the bracket, quotations, or other punctuation used), however, the built-in functions listed above all you to force an object to take on the particular data type. You can always check the datatype of a variable by using the `type()` built in function, like you see in the example below.

File `variables.py`:

```
a = 5
b = 7
print(a, b)
print(a + b)

print(type(b))

b = "7"
print(b)
print(type(b))
```

Command line:

```
$ python variables.py
5 7
12
<class 'int'>
7
<class 'str'>
$
```

Here you can see we created two integer variables a and b and printed their sum. We then reassigned b to a string with the same character.

Variables are stored in "memory" this is (literally) a physical location (an "address") in your computer's RAM. If two variables or code objects have the exact same address in memory, then they have the *exact same* contents. The built-in function id() will print out this "identifier" or "address" for the object.

File `identity.py`:

```
# Has the same id
a = [1, 2, 3]
c = a
print(id(a), id(c))

# Has a different id
b = 42
print(id(b))
b = '42'
print(id(b))
```

Command line:

```
$ python identity.py
140396474295552 140396474295552
140396474879568
140396473486704
$
```

As you can see the two lists in the first example have the location in memory (its “id”) despite being different variable names. In the second case the same variable has a different location in memory when it was reassigned from an integer datatype to a string datatype.

At any point in your code you can access what variable names are assigned to some code object by using the built in functions `locals()` and `globals()`. The two correspond to different “name spaces.”

Indentation/whitespace

Whitespace is how Python identifies different “code blocks” (lines of code that are executed together). Any amount of whitespace at the start of a line will indicate it is a different code block, but the amount of white space (number of spaces) must be the same. It is standard practice to use 4 space characters (same as a <tab>) to delineate separate code blocks. This is necessary for nesting of loops, if-statements, and function definitions.

Functions

Functions are an essential way of simplifying and modularizing code. A chunk of code can be put into a function which allows it to be called at other points in the code, multiple times, without having to reproduce all the lines in the chunk of code.

To declare a function you start with the def statement, then the function name, and lastly the parameters the function takes as input are included in parenthesis. Each function parameter is either positional (come first and don’t have a variable name), named (also called a “keyword argument” or “kwarg” and have to come after positional).

File `function.py`:

```
def check_equal(a, b, precision = 0.001):
    is_equal = abs(a - b) < precision
    return is_equal

print(check_equal(0.1, 0.005))
print(check_equal(0.0015, 0.001))
```

Command line:

```
$ python function.py
False
True
$
```

Here you can see we don't have to specify a value for the kwarg `precision=` because it has a "default" value. Unless specified the function will use the default. Functions also are what define the "name space"---local variables are those defined *inside* a function and global variables are those defined *outside* a function.

NOTE: Avoid using mutable values for your kwargs defaults. Defaults are defined at the time of definition, not at the time of calling. Therefore, if you have multiple instances of a function and you change the default value of a kwarg, it will propagate to the other function calls.

Imports

Importing is a process by which you can pull in large portions of code (defined functions, classes, data literals, etc.) from other files, to be used in your `.py` file. To do so, there is a command `import` you can call within your scripts. For conda installed packages, the files containing the module's code reside in the conda environment directory. Say we installed `numpy` in our environment, for example. We can use the contents of the package by importing it as follows.

File `numpy_import.py`:

```
# import command can use 'as' to provide an abbreviation for the package
import numpy as np

rand_integers = np.random.randint(0, 10, size=20)
print("Twenty random ints from 0 and 10:", rand_integers)
```

Command line:

```
$ python numpy_import.py
Twenty random ints from 0 and 10: [8 0 5 7 9 3 7 9 1 2 7 1 3 6 2 8 2 9 0 1]
$
```

You can also create your own modules/libraries to be imported into your scripts. The simplest way to do this is to create your library file in the same directory as the script that will be importing it. The import name is simply the library filename, sans `.py` extension. In the below example we put the same function `check_equal()` we wrote in the previous section into a new file `my_lib.py`. We then import that function in our `my_imports.py` script.

File `my_lib.py`:

```
if __name__ == '__main__':
    raise RuntimeError("This module is intended only for import.")

def check_equal(a, b, precision = 0.001):
    is_equal = abs(a - b) < precision
    return is_equal
```

File `my_imports.py`:

```
import my_lib as ml

print(ml.check_equal(100, 100.5, precision=1.0))
print(ml.check_equal(100, 100.5, precision=0.1))
```

Command line:

```
$ python my_lib.py
Traceback (most recent call last):
  File "my_lib.py", line 2, in <module>
    raise RuntimeError("This module is intended only for import.")
RuntimeError: This module is intended only for import.

$ python my_imports.py
True
False
$
```

This approach is restrictive because it requires the library file that's being loaded to be located in the same directory as your script. There are more sophisticated ways of creating your own library, such that they can be conda installed. We will cover this later on in the course.

Iterable Variables

Here we will discuss four of the iterable datatypes: lists, tuples, sets and strings. As the name implies, all of these can be iterated (or "looped") over. This means you can write a simple routine to step through every element of the datatype. They are defined by the bracketing used to enclose them: list (square brackets, [ ]), tuple (parenthesis, ( )), set (curly braces, { }), and strings (quotations, " " or ' '). All of these are of indefinite length. However they have many differing properties.

- **Order:** Lists, tuples, and strings are ordered (which means they can be indexed) whereas sets are not. As a result, sets cannot be subscripted or "sliced" (subscript is zero-indexed).
- **Mutability:** Lists and sets can be altered after their initial creation (mutable). Items can be added, removed, reordered, or changed. Tuples and strings on the other hand are *immutable*, meaning that they cannot be changed once created. If you want a tuple with a different set of elements, you need to create a whole new one.

- **Elements:** The elements of lists, sets, and tuples can be essentially any python object. Strings on the other hand can only contain ascii characters.
- **Uniqueness:** Lists, tuples, and strings can contain any number of repeated elements. The elements of a set on the other-hand consist of only a unique collection.

Here we demonstrate some of the various properties of these four datatypes.

File `list_tup_set_str.py`:

```
my_list = [1, 2.0, "three", 4]
my_tup = (1, 2.0, "three", 4)
my_set = {1, 2.0, "three", 4}
my_str = "1, 2.0, three, 4"

# Subscripting ordered datatypes
print(my_list[0])      # index
print(my_tup[0:2])     # slice
print(my_str[-5:])     # negative index and slice

# Adding elements
my_list.append(5.0)
print("my_list with new element:", my_list)

# Uniqueness
my_set.add(2.0)
print("2.0 was already in my_set:", my_set)
```

Command line:

```
$ python list_tup_set_str.py
1
(1, 2.0)
three, 4
my_list with new element: [1, 2.0, 'three', 4, 5.0]
my_set didn't change: {'three', 1, 2.0, 4}
$
```

Arrays are yet one more iterable datatype, though they are not a basic type in Python. They are essentially lists, but require that all the elements of the list be of the same datatype (specifically a numeric data type). The array can be accessed from either the `array` module (`from array import array`) or from the `numpy` module (`from numpy import array`).

Loops

Loops are one of the most essential tools in procedural programming. The two main types of loops are `for` and `while` loops. Essentially they are a way of repeatedly executing a code block some specified number of times. The difference is that the for-loop requires you to specify the number of times of execution with a `range` function, whereas a while-loop will execute that block indefinitely, until a stop condition is met.

File `for_while.py`:

```
print("The range function:", range(1, 5))

# for runs the code block N times, defined by the range(start, stop)
total = 0
for i in range(1, 5)
    print(total)
    total += i

print("Total sum using for-loop:", total)


# while tests whether i is less than 5 each time it runs the code block
total = 0
i = 1
while i < 5:
    total += i
    i += 1

print("Total sum using while-loop:", total)
```

Command line:

```
$ python for_while.py
The range function: range(1, 5)
0
1
3
6
Total sum using for-loop: 10
Total sum using while-loop: 10
$
```

Both can be used to iterate through any of the datatypes discussed in the last section. For example, instead of using the range() function to define the number of loops, the for-loop can take a list. It will then execute the code block a number of times equal to the length of the list.

File `for_iterable.py`:

```
my_list = [1, 2, 3, 4]

prod = 1
for elem in my_list:
    print(prod)
    e_squared = elem ** 2
    prod *= e_squared

print("Product of the squares of all elements in my_list:", prod)
```

Command line:

```
$ python for_iterable.py
1
1
4
36
Product of the squares of all elements in my_list: 576
$
```

Here we see that not only did the length of `my_list` determine how many times the for-loop's code block would run, but we could also use the elements of the list in the code block.

If-statements

If-statements are a way of running mutually exclusive code blocks, depending on the result of a conditional check. The if statement condition must evaluate to a boolean value (`True` or `False`) and the code block will only run if the condition evaluates to `True`. if-statements can be used inside for-loop code blocks to make conditional actions on each element in an iterable.

File `if_statements.py`:

```
a = 40

if a == 42:
    print(a)
else:
    print("Variable a is not the answer to everything.")


# if-statement in a for loop to keep only positive values

my_list = [-1, 13, 2.3, -5, -0.01, 520]
positives = list()

for e in my_list:
    if e >= 0.0:
        positives.append(e)
    else:
        continue

print(positives)
```

Command line:

```
$ python if_statements.py
Variable a is not the answer to everything.
[13, 2.3, 520]
$
```

String parsing

One of the many reasons strings are a useful datatype is because they are the literal format that you evaluate when reading the contents of a text file, line-by-line. For that reason, it is helpful to be able to split up the contents of each line in a data file, consistent with how it is formatted ("parsing"). The most common way of storing various fields of information in a line from a data file is to delimit the fields by commas ("comma separated values" or CSV). Therefore, we can use some string methods to split a string at the right spots.

File `string_parse.py`:

```
# .split() breaks a string at specified character into a list of strings

my_string = "Boulder,CO,80304,106392"
my_split_string = my_string.split(",")
print(my_split_string)


# .strip() will remove leading or trailing whitespace

white_str = "   Boulder,CO,80304,106392\n"
strip_split_str = white_str.strip().split(",")
print(strip_split_str)
```

Command line:

```
$ python string_parse.py
['Boulder', 'CO', '80304', '106392']
['Boulder', 'CO', '80304', '106392']
$
```

The `.strip()` method is particularly helpful to remove the newline character ("\n") at the end of all lines in a text file, as shown in the example. Lastly, note that the elements in these final lists are still strings. We can then use the built-in type-casting functions to express these strings as their numeric equivalents (e.g. `int('80304')` will return the integer `80304`).

Reading files

To read a file in Python you need to open it, loop over all the lines in it, and then close the file. You can extract the data you need as you loop over each line. This procedure is facilitated by creating a "file object" with the `open()` command. The file object it creates is an iterable, so you can loop over the elements (lines) of it just like we looped over a list in one of the previous examples. Each line is read in as a single string. The open() function allows you to read or write to a file. To read a file you need to specify the 'r' option. To write to a file you need to specify the 'w' option.

In text files there are multiple types of whitespace characters that structure the file, but are not obviously visible when you open the file in an editor. However, when python reads the file it will parse these just like any other ASCII character (see "String Parsing"). For example a <tab>

character is expressed as '\t'. The character that separate one line from the next is called a "newline" character, which can be '\n', '\r', or '\r\n', depending on how the file was created and on what operating system.

When you print a string containing whitespace characters, those characters will be reformatted.

```
$ python -c "print('a\nb\tc\nd\n\ne')"
a
b       c
d

e
$
```

There are two ways of using the `open()` function---either it is paired with the `close()` function or it is placed within a with/as code block. The following example uses both methods to print the contents of a file.

File `file_reading.py`:

```
file_name = "counties_data.csv"

# Reading a file using open/close
file = open(file_name, 'r')

for line in file:
    print(line, end='')

file.close()

# Reading file using with/as
print("Now to use the with/as method.")

with open(file_name, 'r') as f:

    for l in f:
        print(l, end='')
```

Command line:

```
$ python file_reading.py
County,State,Population,Area_sq_miles
Boulder,CO,106392,740.0
Broomfield,CO,70465,33.55
Gilpin,CO,6243,150.0
Grand,CO,15734,1870.0
Jefferson,CO,582881,774.0
Larimer,CO,356899,2634.0
Weld,CO,324492,4017.0
Now to use the with/as method.
County,State,Population,Area_sq_miles
Boulder,CO,106392,740.0
```

```
Broomfield,CO,70465,33.55
Gilpin,CO,6243,150.0
Grand,CO,15734,1870.0
Jefferson,CO,582881,774.0
Larimer,CO,356899,2634.0
Weld,CO,324492,4017.0
$
```

The second method (with/as) will automatically close the file once the code block has completed running. However, all the file parsing code must be placed within the code block.

Read from STDIN

STDIN ("standard input") can be thought of as similar to a file. It is the stream of characters that is fed from the command line. When coding in Python, STDIN can be accessed with the `sys` library and can be treated the same as a file object. Below you can see an example

File `read_stdin.py`:

```
import sys

for i in sys.stdin:
    print(i, end='')
```

Command line:

```
$ cat counties_data.csv | python read_stdin.py
County,State,Population,Area_sq_miles
Boulder,CO,106392,740.0
Broomfield,CO,70465,33.55
Gilpin,CO,6243,150.0
Grand,CO,15734,1870.0
Jefferson,CO,582881,774.0
Larimer,CO,356899,2634.0
Weld,CO,324492,4017.0
$
```

Command line variables

The `sys` library can be used to access command line arguments. In Python, these are converted to a list and thus can be indexed. This method is an easy way of passing input parameters to your python script.

File cmd_line_args.py:

```
import sys

args = sys.argv

print("args object type:", type(args))
print("length of args:", len(args))
print(args)
```

Command line:

```
$ python cmd_line_args.py 1 two 3 four
args object type: <class 'list'>
length of args: 5
['cmd_line_args.py', '1', 'two', '3', 'four']
$
```

Here we see that `sys.argv` is a list object and it has separated all the command line arguments passed to the `python` call. Note the delimiter for the command line is whitespace, and the script name is the first element in that list. Each element is cast as a string.