Intro to Python

Benjamin Short

What is Python?

From Wikipedia:

“Python is an interpreted high-level general-purpose programming language. Python's design philosophy emphasizes code readability with its notable use of significant indentation. Its language constructs as well as its object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects”

What does that mean?

- Interpreted - This means the code you write is what is being executed. The code is not being transformed into an intermediate file through “compiling”, it is “interpreted” as-is
- High-Level - Essentially, it’s closer to English than Binary. Very close to English.
- General-purpose - As the tin says, Python was developed to be used for a wide variety of problems rather than solving a specific problem
- Emphasize Code Readability - When you inherit code, or even want to use someone else’s code for something cool, you’d like to know what it’s doing; Python tries to persuade the developer to follow this
- Significant Indentation - Python’s closure delimiter is not a curly brace, but the tab character
- Object-Oriented - Everything in Python is an Object, meaning there is no such thing as the “primitive” data types, at least not in the common sense of thinking

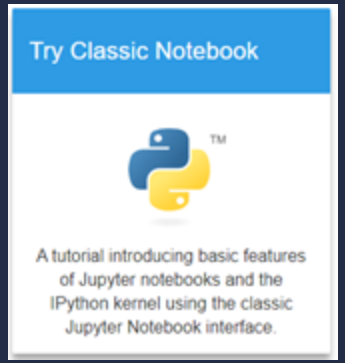
Why Python?

- Easy access for beginners
- Large library of existing packages
- Interpreted language status supports REPL-style usage
- Good at data processing
- Dynamically Typed

Getting Started

For those at home who desire to do so, here's how you can launch a local Jupyter instance in your web browser to follow along.

Step 1: Go to https://jupyter.org/try
Step 2: Choose 'Try Classic Notebook'
Step 3: Wait for the notebook to load
Step 4: Start coding!



A few quick commands:
DD - delete current cell
B - Create new cell Below
A - Create new cell Above

Humble Beginnings

Traditionally, the first program someone writes in a language they are learning is called “Hello, World”. It’s called this because the only objective is to print the string “Hello, World”. In Python, this is a single-line program:

```
print("Hello, World")
```

When using the REPL, it’s technically even easier.

```
>>> "Hello, World"
'Hello, World'
>>>
```

This is because within the REPL, it will automatically print the value of the last instruction. This doesn’t translate perfectly into Jupyter, so I personally don’t advise relying on this too often. The main issue is that it restricts what you can print to a single thing. See:

```
a = "Hello"
b = "World"
a
b
```

```
'World'
```

```
print(a)
print(b)
```

```
Hello
World
```

More Printing - Observations

You may have noticed that one print statement was surrounded by quotes, and one wasn't. You also may have wondered why that would be. The simple answer is to tell you what type of variable you're manipulating. Examine:

```
a = 1
b = "1"
print(a)
print(b)
```

```
1
1
```

```
a
```

```
1
```

```
b
```

```
'1'
```

```
b.lower
```

```
<function str.lower>
```

You can clearly see in the two later images what the type of your data is, while the first is ambiguous. You might even have more complicated types which can't be easily expressed as a string

You might wonder why I advise against using "last instruction printing" if it helps you determine types then. Well, there is a better way: The "type" function.

```
print(type(a))
print(type(b))
```

```
<class 'int'>
<class 'str'>
```

```
print(type(b.lower))
```

```
<class 'builtin_function_or_method'>
```

Data Types

There are a number of data types to deal with in Python. Commonly they will boil down to a handful, but it's good to have at least passing exposure to each. The ones I will cover are:

- Number - Numbers for math
- String - Text, typically ASCII readable
- Bytes - Raw byte data
- Dict - Key:Value lookup table, or a map
- List - Ordered set of elements
- Set - Set of elements without duplicates
- Bool - True / False
- None - Nothing
- Tuple - Like a list, but immutable

Numbers

Technically, Python will split these into int and float, but unlike some languages, a variable switches between them easily.

```
a = 1
print(type(a))
a /= 2
print(type(a))
```

```
<class 'int'>
<class 'float'>
```

These can be used for all the standard math operations

```
a = 3
b = 4
print(a+b)
print(a*b)
print(a/b)
```

```
7
12
0.75
```

Bool

Standard True / False dichotomy, not a whole lot special about this other than the fact that True / False must be capitalized

```
print(True)
print(False)
```

```
True
False
```

String

A string (or str) is basically any text that you want to use or manipulate. It's commonly composed of printable ASCII characters, but this is not a hard requirement.

Additionally, strings can be formatted in several different ways. Some of the formatting is more dependent on personal taste.

```
hello = 'Hello'
world = 'World'
all_together = f"{hello}, {world}!"
print(all_together)
```

```
Hello, World!
```

```
hello = 'Hello'
world = 'World'
all_together = "{}, {}!".format(hello, world)
print(all_together)
```

```
Hello, World!
```

```
a = "abc\x08def"
print(a)
a
```

```
abdef

'abc\x08def'
```

```
normal = "abc\ndef"
literal = r"abc\ndef"
sep = "---------"
print(normal)
print(sep)
print(literal)
```

```
abc
def
---------
abc\ndef
```

Bytes

The bytes object is as close to raw data as you can get, since they are composed of literal bytes. These can be used for a number of reasons, and while not incredibly common, they do occasionally have a use case and make an appearance.

```
a = b"abc\x08def"
print(a)
a
```

```
b'abc\x08def'

b'abc\x08def'
```

List

Kind of like an array, but you can't skip indices. As such, you also cannot set a list's length, you just add things to it or operate on it. Lists also do not have to contain everything of the same data type.

```
a = [1, 2, False, "hello"]
print(a)
print(a[1])
```

```
[1, 2, False, 'hello']
2
```

Tuple

Similar to a list, but unchangeable. Meaning, once values are placed into the tuple, they stay that way. New values can be appended, but existing values cannot change.

```
t1 = (1, 2)
t2 = (3, 4)
t1 += t2
print(t1)
try:
    t1[0] = 5
except:
    print("Told you so")
```

```
(1, 2, 3, 4)
Told you so
```

Dictionary

What Python calls a dictionary is also known as a map. Conceptually, there is a value that needs to be stored, but this value is associated with another value; this could be the address associated with a name, or a translation of one word into another language, etc. The associated value can even be another dictionary to nest values.

```
person = { 'first_name': 'Ben', 'last_name': 'Short', 'profession': 'Software Engineer' }
pid = '12345'
my_dict = { pid: person }
print(my_dict.get(pid))
```

```
{'first_name': 'Ben', 'last_name': 'Short', 'profession': 'Software Engineer'}
```

Set

Notationally similar to a dictionary, but conceptually a little closer to a list. Sets allow you to naively add elements and be sure that at the end, you'll have no duplicates. Functionality can get finicky with special types (i.e. not numbers and strings), but can be finagled.

```
my_set = {1, 2, 3, 2, 3, 4, 1, 3 }
print(my_set)
```

```
{1, 2, 3, 4}
```

Another main difference from lists is the appending procedure; lists use the .append() function, where sets use .add()

NoneType

Basically, null for other languages. Signifies there's nothing there, or an error occurred, etc.

```
d = {}
print(d.get('stuff'))
print(type(d.get('stuff')))
```

```
None
<class 'NoneType'>
```

Since I have the empty space to do this, wherever possible, do not use index notation to access a dictionary. If nothing is there you will raise an exception, and if you aren't handling it explicitly your program will crash. Technically if you don't handle the None it will crash too but we don't make those kinds of programming oversights, right? =)

```
d = {}
print(d.get('stuff'))
print(d['stuff'])
```

```
None

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-21-6bd3930ef988> in <module>
      1 d = {}
      2 print(d.get('stuff'))
----> 3 print(d['stuff'])

KeyError: 'stuff'
```

Looping

Sometimes you just gotta do things many times. Like saying “Good Morning” 3 times, right in a row. Of course, we could just do it three times…

Or, we could use a loop!

```
print("Good Morning")
print("Good Morning")
print("Good Morning")
```

Loops allow us to do things repeatedly, and offer us a choice for conditions; a set number of times, or until a certain condition is evaluated false.

The ‘for’ loop:

```
for i in range(0, 3):
        print("Good Morning")
```

```
Good Morning
Good Morning
Good Morning
```

A custom greeting:

```
names = ['John', 'Jane', 'Sam']
for name in names:
        print("Good Morning, {}!".format(name))
```

```
Good Morning, John!
Good Morning, Jane!
Good Morning, Sam!
```

The ‘while’ loop:

```
times_to_repeat = 3
times_repeated = 0
while times_repeated < times_to_repeat:
        print("Good Morning")
        times_repeated += 1
```

```
Good Morning
Good Morning
Good Morning
```

Control Flow

So now we know some basics, and we know how to repeat ourselves; but a program that always does the same thing every time doesn't seem the most helpful.

This is where 'if' statements come in; 'if' this, do that.

From one of our 'for' loop examples, let's say Sam and I don't get along, and I don't want to greet them. Using an 'if' statement, I can selectively not say 'Good Morning'.

```
names = ['John', 'Jane', 'Sam', 'Greg', 'Sarah']
for name in names:
        if name is not 'Sam':
            print("Good Morning, {}!".format(name))
```

```
Good Morning, John!
Good Morning, Jane!
Good Morning, Greg!
Good Morning, Sarah!
```

The 'if' structure also allows for an 'else' block. So if I still wanted to be cordial to Sam, but not as energetic, I could have something like this:

```
names = ['John', 'Jane', 'Sam', 'Greg', 'Sarah']
for name in names:
        if name is not 'Sam':
            print("Good Morning, {}!".format(name))
        else:
            print("Hello, {}.".format(name))
```

```
Good Morning, John!
Good Morning, Jane!
Hello, Sam.
Good Morning, Greg!
Good Morning, Sarah!
```

The ternary operator

Within the ‘if...else’ structure is what’s known as the ‘ternary operator’. It’s called this due to having a three-component setup:

1. The initial value
2. The conditional to confirm the initial value
3. The backup value

This is useful when you only have two possible values, and a fairly simple way of differentiating which of the two values you want to use. For example, if you were evaluating whether a certain expression was greater than another expression.

You could write it something like this:

```
a = 5
b = 4
if a*b > 17:
    g = True
else
    g = False
```

Or, we could use the ternary operator to save some space and look nice:

```
a = 5
b = 4
g = True if a*b > 17 else False
```

For this example, ‘True’ is the initial value, ‘a*b > 17’ is the condition, and ‘False’ is the backup value

Functions

- Functions allow you to logically split your code into other segments
- Then, you can name those segments, and re-use the code
- Properly-constructed functions heavily reduce code duplication
- Allow exporting as a library for use elsewhere
- Allows event-driven programming
- Easier to understand complex code

Functions are declared with the ‘def’ keyword, and called using parenthesis

```
def print_hello():
    print("Hello, World!")

print_hello()
```

```
Hello, World!
```

Functions can also take arguments, so that output is customized or dynamic

```
def print_hello(name):
    print("Hello, {}!".format(name))

print_hello("Jerry")
```

```
Hello, Jerry!
```

Functions - Scope

Variables in functions have their own scope; having the same name will result in different values. This can be good or bad depending on what you want to do.

Here, a variable declared outside the function does not affect the variable inside

```
def myFunc():
    x = 7
    print("x (func) is {}".format(x))

x = 3
print("x (global) is {}".format(x))
myFunc()
print("x (global) is {}".format(x))
```

```
x (global) is 3
x (func) is 7
x (global) is 3
```

Additionally, the variable inside does not affect the variable outside

Functions - Scope

A lot of the time you may want to reference a variable out of scope. This can be done as well.

```
def myFunc():
    print("x (inside) is {}".format(x))

x = 3
print("x (outside) is {}".format(x))
myFunc()
```

```
x (outside) is 3
x (inside) is 3
```

The ‘x’ variable has global scope, and can thus be accessed from anywhere

Functions - Scope

To allow assignment operations, use the ‘global’ keyword when initializing the variable

```
def myFunc():
    global x
    x = 7
    print("x (func) is {}".format(x))

x = 3
print("x (global) is {}".format(x))
myFunc()
print("x (global) is {}".format(x))
```

```
x (global) is 3
x (func) is 7
x (global) is 7
```

Functions - Arguments & Returns

Functions can also take arguments and return results. This allows programs to be more dynamic; just like with control flow, a program is fairly boring if it always does the exact same thing.

By supplying different values, your program can receive different results and react accordingly

```
def square(x):
    return x * x

sq_3 = square(3)
print("3 squared is {}".format(sq_3))
```

```
3 squared is 9
```

By storing the return values, you can make future decisions based on those values

```
def square(x):
    return x * x

sq_4 = square(4)
if sq_4 > 10:
    print("It's greater than 10")
else:
    print("It's not greater than 10")
```

```
It's greater than 10
```

Lambdas

Functions that don’t require names. Typically smaller, useful for quick functions you only plan to use once, or functions that are very customized.

```
def runMyFunc(myFunc):
    myFunc()

def func1():
    print("I'm func1")

def func2():
    print("I'm func2")

runMyFunc(func1)
runMyFunc(func2)
runMyFunc(lambda : print("I'm a lambda"))
```

```
I'm func1
I'm func2
I'm a lambda
```

Note, this isn’t a very standard use-case, but illustrates the point

Lambdas - Contd

Lambdas can also be given parameters

```
def runMyFunc(f, a, b):
    return f(a, b)

ans = runMyFunc(lambda a, b : a + b, 3, 6)
print("The answer is {}".format(ans))
```

```
The answer is 9
```

Lambdas can also be assigned to variables

```
add = lambda a, b: a+b
ans = add(2, 4)
print("The answer is {}".format(ans))
```

```
The answer is 6
```

If lambdas can be assigned to a variable name, and they can take arguments, then aren't lambdas just functions?

The TL;DR? Yes, basically. Which is why they have their special use-case

```
def amIAJoke():
    print("Yes")

thisIsAJokeRight = lambda : print("Yes")

amIAJoke()
thisIsAJokeRight()
```

```
Yes
Yes
```

Lambdas are ill-suited to complex logic and calculations, but well-suited to simple yet custom and/or dynamic requirements

Libraries

Python has an extensive library collection. One you're likely very familiar with is pandas, for DataFrame usage. Others of note are os, requests, datetime, sqlite3, and more. Libraries are imported using the 'import' keyword

Some Examples:

```
>>> import os
>>> os.system('echo hello')
hello
0
>>>
```

```
from datetime import datetime
print(datetime(1994, 3, 8))
```

```
1994-03-08 00:00:00
```

```
import requests
resp = requests.get('https://www.google.com')
resp.status_code
```

```
200
```

```
import sqlite3
con = sqlite3.connect('ex.db')
con.execute('CREATE TABLE ex (data text)')
con.execute("INSERT INTO ex VALUES ('Example Data')")
con.commit()
cur = con.execute("SELECT data FROM ex")
res = cur.fetchall()
cur.close()
con.close()
for row in res:
    print(row[0])
```

```
Example Data
```

But...What if I don’t know what a package is capable of? Or forget the syntax?

Debugging Commands

Okay, not really “debug” commands per se, but that’s my typical use-case. When I forget what’s available, or want to check what should be available, this is my process.

Let’s say I couldn’t quite remember if the requests library puts the status code as status_code, or statusCode, or if it was a function so status_code(); without even going to the API, we can deduce this.

Step 1: Get the object you want to analyze. In this case, our GET return

```
ret = requests.get('https://www.google.com')
```

Step 2: Use the ‘dir’ function

```
print(dir(ret))
```

```
['__attrs__', '__bool__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__nonzero__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_content', '_content_consumed', '_next', 'apparent_encoding', 'close', 'connection', 'content', 'cookies', 'elapsed', 'encoding', 'headers', 'history', 'is_permanent_redirect', 'is_redirect', 'iter_content', 'iter_lines', 'json', 'links', 'next', 'ok', 'raise_for_status', 'raw', 'reason', 'request', 'status_code', 'text', 'url']
```

Step 3: Now that we know the naming format, what is this? (a variable, a function; further, is it a string, or a number, etc)
So now, run the ‘type’ command mentioned earlier

```
print(type(ret.status_code))
```

```
<class 'int'>
```

Step 4: Start using the data!

```
print("The request returned code {}".format(ret.status_code))
```

```
The request returned code 200
```

And that’s all there is to it. This might have to be nested a few times in the event each level is a new custom type, but you should be able to piece together any functionality. This even works when you forget what attributes a data type has, such as whether set uses ‘add’ or ‘append’, or if your list has a ‘pop’ function.

List/Dict Comprehensions

Ironically, comprehensions are one of the harder Python tools for people to comprehend =)

Standard method for iterating over a list is a ‘for...in’ loop

A list comprehension does this in a more concise manner, similar to the ternary operator for control flow.

The format for a list comprehension is: [a for a in b if condition]

Dict comprehensions also exist: {a:b for a,b in lst if condition}

Comprehensions can be very helpful depending on what you’re trying to do, simplifying code, or saving space

Our example data:

```
lst = [1, 2, 3, 4, 5, 6]
lst_d = [('a', 1), ('b', 2), ('c', 3), ('d', 4)]
```

'Standard' method:

```
new_list = []
for item in lst:
    if item > 2:
        new_list.append(item)
print(new_list)
```

```
[3, 4, 5, 6]
```

With a comprehension:

```
comp_list = [item for item in lst if item > 2]
print(comp_list)
```

```
[3, 4, 5, 6]
```

Similarly, if we were to make a dictionary the 'standard' way:

```
new_d = {}
for k,v in lst_d:
    if v > 2:
        new_d[k] = v
print(new_d)
```

```
{'c': 3, 'd': 4}
```

Or with our new comprehension skills:

```
comp_d = {k:v for k,v in lst_d if v > 2}
print(comp_d)
```

```
{'c': 3, 'd': 4}
```

Other topics - Stay Tuned

Generators

More string formatting

JSON

File IO

Closures

Variable-argument functions

- Function call variable assignment(foo(a=1, b=2, …))

User Input

Error handling

Classes

Questions?

- Comments?
- Concerns?
- Qualms?
- Queries?

Extra Project - Loan Calculator

To get user values, use input()

Function 1 - Calculate monthly payment given principal, interest, and term

Function 2 - Calculate time to pay in full given principal, interest, and payment amount

Function 3 - Calculate remaining balance given principal, interest, payment amount, and term