Lecture 02 - Intro to Python - Notebook (pre-class version)

January 16, 2024

1 Lecture 2: Quick(ish) Introduction to Python

[ ]:
```
import platform
print(platform.python_version())
```

The goal of this lecture is to give you some of the basics. It's not possible for us to cover **everything** you'll need to know ahead of time. As graduate students, you are expected to be able to do some research and self-teaching on your own to build your coding skills, and of course you can *always* come ask me for help!

A nice reference for a lot of the computational skills we'll be covering (coding, unix command line, git) is the "Software Carpentry" set of lessons: https://software-carpentry.org/lessons/. You should seriously considering checking their tutorials for extra practice and more in-depth lessons!

This document is a "Jupyter Notebook". It's kind of like interactive mode, but also lets you intersperse text, html, etc, among it. (It's free and easy to set up.)

Python is a **whitespace-based language**. In C, C++, Java, and many other languages, you use braces to group code:

```
if (x == 1) {
    do_something();
}
```

and the spacing is just for readability. For example, the following code does the same thing:

```
if (x == 1) { do_something();
          }
```

In Python you use indenting, and colons (`:`) to have the same effect. You also do not use semicolons (`;`) to end commands.

```
if x == 1:
    do_something()
```

You have to be *really* careful to be consistent by either - always using tabs, or - always using spaces (and the same number)

[ ]:
```
x = 1
```

[ ]:
```
if x == 1:
    print("hello")
```

[ ]:
```
if x == 1:
            print("hello")
```

[ ]:
```
if x == 1:
    print("hello")
     print("world")
```

[ ]:
```
if x == 1:
    print("hello")
    print("world") # if you use a tab, Jupyter will fix it for you␣
 ↪automatically! Your code editor might not.
```

Python uses `if`, `for`, and `while` statements like many other languages.

[ ]:
```
x = 1
while x < 10:
    x = x + 2

    print(x)
```

[ ]:
```
for y in range(3, 6):  # 3, 4, 5
    print(y)
```

[ ]:
```
for letter in "apple":
    print(letter)
```

In a `for` loop, you can iterate over many different types of objects (lists, sets, tuples, dictionaries, strings, etc.)

`range(a,b)` is a way of looping over all of the integers between `a` (inclusive) and `b` (exclusive).

[ ]:
```
for z in "hello":
    print(z)
```

[ ]:
```
for z in [19, -100, "banana"]:
    print(z+1)
```

You may have noticed that Python is not a **statically-typed** language, which means you do not need to tell it whether a variable you are defining is an integer or a string or a list, etc. You just define it, and it figures it out.

But, there are still different types! You can always use the `type` function to check what type an object has.

[ ]:
```
L = [1, 2, 3]
type(L)
```

[ ]:
```
L = (1,2,3)
type(L)
```

[ ]:
```
L = {1,2,3}
type(L)
```

[ ]:
```
sum([1,2,3])
```

[ ]:
```
type(sum)
```

Now we're going to discuss a bunch of the fundamental types in Python.

1.1 Integers, Floating Point Numbers, and Complex Numbers

[ ]:
```
x = 7
type(x)
```

[ ]:
```
import math
math.pi
```

[ ]:
```
x = 7.0
type(x)
```

[ ]:
```
x = 3.0000000000000000001
print(x)
```

[ ]:
```
y = 0.1
print(y)
```

[ ]:
```
0.1 + 0.1 + 0.1 - 0.3
```

[ ]:
```
0.1 + 0.1 + 0.1 == 0.3
```

[ ]:
```
15 / 7
```

[ ]:
```
15 // 7
```

[ ]:
```
z = complex(3, 5)
t = complex(1,-1)
print(z)
print(type(z))
print(z*t)
```

1.2 Boolean

A boolean is just a `True` or `False` value. That's it!

[ ]:
```
b = True
type(b)
```

[ ]:
```
if b:
    print("hello")
```

[ ]:
```
if not b:
    print("hello")
```

[ ]:
```
t = 10 + 10 == 20 # Use "==" to test equality, and "=" to actually set␣
 ↪something equal
print(t)
type(t)
```

1.3 None

There is a weird object in Python called `None`. It's just a useful thing to have around, often as a default value until you assign something.

[ ]:
```
y = None
print(y)
if y is None:
    print("y has the value None")
y = 3
if y is None:
    print("y has the value None")
```

1.4 Strings

A string is just a sequence of characters.

[ ]:
```
type("banana")
'banana'
```

You can do a million things with strings.

[ ]:
```
s = "banana"
s.split("n")
```

Use `len` to find the length of a string and `+` to concatenate two strings together.

[ ]:
```
one = "hello"
two = "world"
three = one + " " + two
print(len(three))
print(three)
```

[ ]:
```
three.len()
```

1.5 Lists

A list is an **ordered sequence** of things.

[ ]:
```
L = [15, "banana", 7, False, [1, 2, 3]]
```

[ ]:
```
print(L)
```

Elements of lists are accessed with bracket notation, starting from 0.

[ ]:
```
L[0]
```

[ ]:
```
L[1]
```

[ ]:
```
L[2]
```

[ ]:
```
L[3]
```

[ ]:
```
L[4]
```

[ ]:
```
(L[4])[1]
```

Use `len(L)` to get the length of a list.

[ ]:
```
len(L)
```

[ ]:
```
len(L[4])
```

You can set elements of the list manually as well.

[ ]:
```
L
```

[ ]:
```
L[1] = "apple"
```

[ ]:
```
L
```

[ ]:
```
L[8] = "can't set this"
```

You can sort lists:

[ ]:
```
R = [15, -20, 0]
R.sort()
print(R)
```

Notice the `.` in the notation above. We'll talk about this more when we cover object-oriented programming, but what we're basically doing here is telling the list `R` to perform its `sort()` operation on itself.

You may wonder why we did `len(R)` instead of `R.len()`… it's kind of just a quirk. You get used to it.

A few more quick list functions:

[ ]:
```
R
```

[ ]:
```
R.append(17)
print(R)
```

[ ]:
```
R.extend([7, 8, 9])
print(R)
```

Lastly (for now) you can concatenate two lists together with the + sign.

[ ]:
```
[1,2,3] + [4,5,6]
```

[ ]:
```
print(R)
M = [100] + R
print(M)
```

1.6 Sets

A list was an ordered sequence of things. A set is an **unordered sequence** of things with no repeats (just like in math).

[ ]:
```
S = {1, 2, 3, 4}
print(S)
```

[ ]:
```
T = {3, 1, 4, 2}
print(T)
```

[ ]:
```
S == T
```

You can't access elements using the bracket notation because there is no first element, second element, etc. You should never assume that you know the order Python will internally store your list in!

[ ]:
```
S[2]
```

[ ]:
```
for element in S:
    print(element)
```

Here are some functions you can do with sets:

[ ]:
```
first = {1,5,6}
second = {2,4,5}
```

[ ]:
```
first.union(second)
```

[ ]:
```
second.union(first)
```

[ ]:
```
first.intersection(second)
```

[ ]:
```
first.difference(second) # all of the things IN first, and NOT IN second
```

[ ]:
```
# By the way, you write comments in python by just starting the line with the␣
↪pound key.
```

[ ]:
```
first
```

[ ]:
```
first.add(9)
print(first)
```

[ ]:
```
first.add(5)
print(first) # No duplicates!
```

[ ]:
```
first.remove(5)
```

[ ]:
```
print(first)
```

[ ]:
```
first.remove(5)
```

1.7 Tuples

It starts to get a little tricky here! A tuple is an **ordered sequence** of things.

Wait… isn't that what a list was?

[ ]:
```
T = (1,2,3,4)
print(T)
type(T)
```

[ ]:
```
L = [1,2,3,4]
L == T  # They are different types of objects, so they can't be equal.
```

The key is that a tuple is what we call **immutable**. Once it's defined, it *cannot* be changed, ever, at all.

[ ]:
```
print(T)
print(T[2])
```

[ ]:
```
T[2] = 17
```

It is still possible to do things like concatenate two tuples to make a new bigger tuple, but it's a **new** bigger tuple, and the original one is still unchanged.

[ ]:
```
T + (5,6)
```

[ ]:
```
T
```

[ ]:
```
T.append(5)
```

So, we define a new tuple with parentheses, but there's one catch: if your tuple has a single item, it needs a special bit of syntax.

[ ]:
```
x = (1)
print(x)
type(x)
```

[ ]:
```
x = (1,)
print(x)
type(x)
```

So, lists are **mutable**, tuples are **immutable**. Why do we need two different versions?

[ ]:
```
L = [1,2,3,4,5,6]
5 in L
```

Under-the-hood, when you store things in a set, Python is being super smart about how it stores it. When you add an element to a set you really do not want python to have to scan one-by-one through all the things in the set to make sure it's not already there. So, it uses a clever technique called *hashing*.

You don't need to know the details right now, but the broad idea is that Python takes each thing in the set and assigns a number to it called its *hash*, and then uses the hashes to make sure there are no duplicates.

[ ]:
```
hash(17)
```

[ ]:
```
hash("banana")
```

[ ]:
```
hash((1,2,3,4))
```

[ ]:
```
hash([1,2,3])
```

[ ]:
```
L = [1,2,3,4]
```

[ ]:
```
{[1, 2, 3, 4], [1, 2], [7,8]}
```

[ ]:
```
{(1, 2, 3, 4), (1, 2), (7, 8)}
```

The problem is that you **can't hash mutable things**. Once you get an object's hash, that needs to stay its hash forever. You could hash a list, then appending an element to the list would mean a new hash would have to be generated, and this would mess everything up.

Bottom line: Sometimes you need an immutable version of something, like to put it in a set.

[ ]:
```
{5, 17, [1,2,3]}
```

[ ]:
```
{5, 17, (1,2,3)}
```

[ ]:
```
{5, 17, {1,2,3}}
```

Sets are mutable too! Sets must contain immutable things, but they themselves are mutable.

Of course we knew this, because we can do `S.add()`. So what if you want sets in your sets? There is an immutable version of a set called a `frozenset`.

[ ]:
```
{ 5, 17, frozenset({1, 2, 3}) }
```

When should you use a tuple versus a list? - If it's going to go in a set (or, as we'll see in a second, in a dictionary), it has to be immutable. Thus, use a tuple. - If you need to be able to add and remove things, use a list. - If the size will always stay the same, you probably want a tuple. For example, if you're representing xy-coordiates, use tuples.

1.8 Dictionaries

You can think of a list as kind of like a mathematical function whose inputs are the the indices 0, 1, … and whose outputs are the elements of the list.

[ ]:
```
L = ["apple", "banana", "pear"]
```

[ ]:
```
#  0 -> apple,  1 -> banana,  2 -> pear
```

In a dictionary, the inputs don't have to be integers, they can be any (immutable) object.

[ ]:
```
# To define  17 -> apple,  banana -> pear,  (1, 2, 3) -> True
d = {17:"apple", "banana":"pear", (1,2,3):True}
print(d)
```

The inputs are called **keys** and the outputs are called **values**.

[ ]:
```
d[17]
```

[ ]:
```
d["banana"]
```

[ ]:
```
d[(1,2,3)]
```

[ ]:
```
d["pear"] = "hello"
d["pear"]
```

You can assign new values too

[ ]:
```
d[1] = "one"
print(d)
```

[ ]:
```
d[ (2, 3, 5, 7) ] = False
```

Dictionaries are *super* useful, but take some getting used to. The keys are hashed in the background, which makes looking up the value for a given key very fast.

[ ]:
```
for k in d.keys():
    print(k)
```

[ ]:
```
for v in d.values():
    print(v)
```

[ ]:
```
for pair in d.items():
    print(pair)
# (key, value)
```

1.9 Casting

You can tell Python to turn an object of one type into an object of another type. This is called **casting**.

[ ]:
```
L = [3, 7, 7, 12]
print(L)
```

[ ]:
```
T = tuple(L)
print(T)
print(L)
```

[ ]:
```
S = set(L)
print(S)
```

[ ]:
```
print(L)
list(set(L))
```

[ ]:
```
dict(L)
```

[ ]:
```
str(L)
```

[ ]:
```
int(L)
```

[ ]:
```
d = {1:"one", 2:"two", 3:"three"}
```

[ ]:
```
list(d)
```

1.10 Practice

Time for some practice!

https://projecteuler.net/

Problem 1: mod, looping, and comprehensions

Problem 2: negative indexing

Problem 5: all / any, and thinking mathematically

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

[ ]:
```
# mod - modulus
#  a % b -- the remainder you get when you divide a by b
```

```
# list comprehensions
# +=
```

[ ]:

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

[ ]:

2520 is the smallest number that can be divided by each of the numbers from 1 to 10 without any remainder.

What is the smallest positive number that is evenly divisible by all of the numbers from 1 to 20?

[ ]:
```
# all / any
```

[ ]: