Intro to Programming in Python

April 1, 2019

```
In [2]: %matplotlib inline
```

1 Intro to Programming in Python

This tutorial provides an introduction to programming in Python, along with a few introductory examples on how Python is generally used for research. We will cover:

- Data types: integers, floats, strings, booleans
- Data structures: lists, sets, dictionaries and tuples
- Loops
- Conditional statements
- Writing functions
- Reading and writing data
- Additional resources

1.1 Variables, data types and operators

You create a new variable by simply declaring it.

```
In [3]: a="Hello World!" #a string variable. Strings need to be placed in
                         #single or double quotation marks.
        b=2 #an integer variable
        c=2/3 #a float variable
        d=(b==24) #a boolean variable
```

To print to console:

```
In [4]: print(a)
```

```
Hello World!
```

```
In [5]: b
```

```
Out[5]: 2
```

```
In [6]: print(c+b)
```

```
2.6666666666666665
```

```
In [7]: d
```

```
Out[7]: False
```

You should always know what type your variables are, since some operations can only be done on certain types of variables. To check variable types:

```
In [8]: print("a is", type(a), ",b is", type(b),  ",c is", type(b),  ",d is", type(d) )
```

```
a is <class 'str'> ,b is <class 'int'> ,c is <class 'int'> ,d is <class 'bool'>
```

1.1.1 Operators

Mathematical, comparison and boolean operations and their order or evaluation:
1. exponent: ** 2. multiplication, division, modulo *, /, % 3. addition, subtraction +, -
4. comparison operators <, <=, >=, >, ==, != 5. comparison operators: is, is not, in, not in 6. boolean NOT, AND, OR: not, and, or

Use () to change the default order. This is just maths.

```
In [9]: 2**b+20/b<=15
```

```
Out[9]: True
```

```
In [10]: 2**(b+20)/b<=15
```

```
Out[10]: False
```

```
In [11]: d==False
```

```
Out[11]: True
```

```
In [12]: d is not True
```

```
Out[12]: True
```

```
In [13]: d is not True and b==3
```

```
Out[13]: False
```

This is just English!

```
In [14]: d is not True or b==3
```

```
Out[14]: True
```

```
In [15]: result=(b+c)-(d*2)
         result
```

```
Out[15]: 2.6666666666666665
```

You can use some of these operators on strings as well.

```
In [16]: "Hello" in a

Out[16]: True
```

But try:

```
In [17]: x="2"
         y="3"
         result=y+x
         print(result)

32
```

For strings, '+' performs concatenation.

```
In [18]: type(result)

Out[18]: str
```

Now try this:

```
In [19]: x="2"
         y=3
         result=x+y

         print(result)


         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-19-58c4b7fd0d3f> in <module>()
           1 x="2"
           2 y=3
     ----> 3 result=x+y
           4
           5 print(result)


         TypeError: must be str, not int
```

Exercise":
What does the error say? How do we fix it?
Insert a new cell below. Declare x and y as numeric variables, the first one equal to 2 and the second one equal to 3, and sum them up.

1.1.2 Converting variables from one type to another

Sometimes, your data has variables that you would like to use as numbers coded as string, just as we have x and y above. Or some of the variables are coded as strings, while others are numbers, although you believe they should all be numbers. If you try to add them however, you get an error saying that you can't add strings and numbers. Assuming all the values of the dataset variables look like numbers, you can convert them into integers or floats. Or the other way around. Now try this:

```
In [20]: x="2"
         y="3"
         result=int(x)+float(y)
         print(result)
```

```
5.0
```

```
In [21]: type(result)
```

```
Out[21]: float
```

And back to string:

```
In [22]: type(str(result))
```

```
Out[22]: str
```

1.1.3 Exercise:

1. Create a new variable called 'birth_year' that contains your year of birth.
2. Using your birth year, calculate your age and assign it to a new variable called 'age'.

3. Print a sentence of the form "I am *age* years old." to the console.
4. Create a new string variable called 'sentence' that contains this statement.

Write your code in the box below. Let's see who finishes first!

1.2 Data Structures

Note that the variable we've been working with so far contain a single value. However, what we normally refer to as "variables" in data analysis are variables from datasets, which contain more than one value. In python, these types of data structures can be lists, sets, dictionaries and tuples.

1.2.1 Lists

Lists are stored between square brackets, and the elements are separated by commas. Here is a list of ages:

```
In [23]: ages=[34, 20, 19, 21, 20, 33, 22, 23, 26, 21, 22, 30, 19, 28]
         ages
```

```
Out[23]: [34, 20, 19, 21, 20, 33, 22, 23, 26, 21, 22, 30, 19, 28]

In [24]: ages[1:5]

Out[24]: [20, 19, 21, 20]
```

Where is the 34 in the list?

```
In [25]: ages[0]

Out[25]: 34

In [26]: len(ages) # this is the number of elements in the list

Out[26]: 14
```

Lists can be indexed and sliced:

```
In [27]: # Indexing - getting an element by position. Note that we start from 0 and we stop at
         first_element=ages[0] # this is the element at index 0
         last_element=ages[13] # this is the element at index 13
         print(first_element, "to", last_element)

34 to 28


In [28]: # Slicing - getting a subset of the elements in the list.
         first_3=ages[0:3] # the same thing as ages[:3]
         last_3=ages[-3:min(ages)] # the same thing as ages[10:14]
         print(first_3, "and", last_3)

[34, 20, 19] and [30, 19, 28]


In [29]: min(ages)

Out[29]: 19

In [30]: ages[10:14]

Out[30]: [22, 30, 19, 28]
```

1.2.2 Other common list operations

```
In [31]: # Check if values in list:
         40 not in ages # true if value is not in the list

Out[31]: True

In [32]: # sorting the list by values:
         ages.sort()
         ages
```

```
Out[32]: [19, 19, 20, 20, 21, 21, 22, 22, 23, 26, 28, 30, 33, 34]

In [33]: # adding to the list
         ages.append(2)
         ages

Out[33]: [19, 19, 20, 20, 21, 21, 22, 22, 23, 26, 28, 30, 33, 34, 2]

In [34]: # concatenating two lists:
         l1=["a", "b", "c"]
         l2=[1, 2, 3]
         l3=l1+l2
         l3

Out[34]: ['a', 'b', 'c', 1, 2, 3]

In [35]: # removing an element from the list by value
         ages.remove(2)
         ages

Out[35]: [19, 19, 20, 20, 21, 21, 22, 22, 23, 26, 28, 30, 33, 34]

In [36]: # finding the index (position) of the first place where the value occurs in the list
         ages.index(21)

Out[36]: 4

In [37]: # remove an element from the list by index
         del ages[0:5]
         ages

Out[37]: [21, 22, 22, 23, 26, 28, 30, 33, 34]
```

1.2.3 Sets

A set contains an unordered collection of unique and immutable objects. If you want to get all unique values in a list, a quick way it to transform the list into a set:

```
In [38]: set_ages=set(ages)
         set_ages

Out[38]: {21, 22, 23, 26, 28, 30, 33, 34}

In [39]: unique_ages=list(set_ages)
         unique_ages

Out[39]: [33, 34, 21, 22, 23, 26, 28, 30]
```

1.2.4 Dictionaries

In a dictionary, an entry consists of a word and the word's definition. The word is the key to finding out what a word means, and what the word means is considered the value for that key. In Python, dictionaries have keys and values. Keys are used to find values. Here is a dictionary of people and their ages:

```
In [40]: mydict = {"John": 21,
                   "Jake": 20,
                   "Jack": 23,
                  }
         mydict

Out[40]: {'John': 21, 'Jake': 20, 'Jack': 23}

In [41]: mydict.keys()

Out[41]: dict_keys(['John', 'Jake', 'Jack'])

In [42]: mydict.values()

Out[42]: dict_values([21, 20, 23])

In [43]: mydict["John"]

Out[43]: 21
```

Dictionaries will be very useful when we start working with web data, such as social media data.

1.3 Indentation

Python **requires** blocks to be structured through indentation. Not just as a matter of style, but as a rule. Statements with the same distance to the left belong to the same block of code. To nest blocks, you need to indent them further to the right. The number of white spaces doesn't matter, what matters is that you are consistently using the same number for blocks that are at the same level. Usually, we start at the very left edge, and each level in goes a further 1 tab (or 4 white spaces) to the right. If the code does not follow this rule about the relative indentation of blocks, then you will get an **IndentationError**.

However, the indentation level is ignored when you use explicit (or implicit) continuation lines. You can split a list or dictionary across multiple lines, and the indentation doesn't matter.

You will see a few examples in the sections below.

1.4 Loops

Most of our work involves some type of iteration over observations in a dataset. Iteration is very easy and intuitive in Python, and there are many ways to loop through data in order to access and manipulate it.

```
In [44]: # for loops
         for i in range(5):
          print("I can count to "+str(i))
```

```
I can count to 0
I can count to 1
I can count to 2
I can count to 3
I can count to 4
```

```
In [45]: a=[20,11,22,3]
```

```
In [46]: # while loops
         counter = 0
         while counter < 5:
             print("I can count to", counter)
             counter=counter+1 #Shortcut is counter += 1
```

```
I can count to 0
I can count to 1
I can count to 2
I can count to 3
I can count to 4
```

```
In [47]: #List comprehension
         k=[key for key in mydict.keys()]
         k
```

```
Out[47]: ['John', 'Jake', 'Jack']
```

1.5 Conditional statements

Data management, processing and analysis involve taking a series of decisions. We use conditional statements (most often in the form of if statements) to take these decitions.

```
In [48]: # if statement
         for i in range(5):
             if i % 2 == 0:
                 print("I can count even numbers to "+str(i))
```

```
I can count even numbers to 0
I can count even numbers to 2
I can count even numbers to 4
```

```
In [49]: # if-else statement
         for i in range(5):
```

```
            if i  % 2 == 0:
                print("I can count even numbers to "+str(i))
            else:
                print("I can count odd numbers to "+str(i))
```

```
I can count even numbers to 0
I can count odd numbers to 1
I can count even numbers to 2
I can count odd numbers to 3
I can count even numbers to 4
```

```
In [50]: # if-elif-else statements
         for i in range(5):
             if i<=1:
                 print("I can count to "+str(i))
             elif 2<=i<=3:
                 print("I can also count to "+str(i))
             else:
                 print("But I can't count to "+str(i))
```

```
I can count to 0
I can count to 1
I can also count to 2
I can also count to 3
But I can't count to 4
```

1.6 Writing functions

You often have to perform the same type of task many times, on different data. To avoid writing the same code over and over, you can write functions that can be called every time you want to perform the specific task.

```
In [51]: def power_of(a, b):
             return a**b
         print(power_of(2,3))
```

```
8
```

```
In [52]: print(power_of(5,5))
```

```
3125
```

1.7 Reading and writing files

You can use the read, write, readlines and writelines functions from base Python to read and write files. We have the examples.csv file that you saved from GitHub. Let's say you are interested

in what regions there are in this data. Let's start by creating a set called regions, which we will populate with the values available in the data.

```
In [53]: regions=set() # create an empty set
         with open("mydataset.csv", "r") as myfile:  # open the file for reading
             data = myfile.readlines()
             for line in data:
                 region=line.split(",")[1]
                 regions.add(region)
         print(regions)
```

```
{'South', 'West', 'North', 'region', 'North-East', 'East'}
```

Now we can open a new file, and write the regions to it:

```
In [54]: with open("regions.txt", "w") as myfile:  # open the file for writing
             for region in regions:
                 myfile.write(region + '\n' )
```

You can also append to a file in mode "a" and open it both for reading and writing in mode "r+".

```
In [55]: with open("regions.txt", "a") as myfile:  # open the file for writing
             for region in regions:
                 myfile.write(region + '\n' )
```

1.8 Additional resources

All very hands-on, excellent for beginners, both in Python and in programming in general.

The Python Tutorial
Learn Python the Hard Way
Dive Into Python 3