Python Syntax for Beginners

Understanding the basics of Python and its core functions

Disclaimer

The purpose of this tutorial is simply to provide an introduction to a variety of topics in Python. It is in no way a substitute for a full for-credit course on Python, and should students want to pursue Python on a more serious level they should look to the university courses on the subject

A Note on the history of Python

Python is a widely used high-level programming language for general-purpose programming, created in 1991. Python's design philosophy emphasizes ***code readability*** (notably using *whitespace* indentation to delimit code blocks rather than curly brackets or keywords), and a syntax that allows programmers to express concepts in fewer lines of code than might be used in languages such as C++ or Java. The core philosophy of Python is as follows:

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated
- Readability counts

(Source: The Zen of Python)

Why are we learning Python?

Other than it's code readability and free open source format, Pythons strongest asset to data scientist are its ***libraries***. Through a library called *NumPy*, Python support numeric analysis as naturally as Matlab does. Through *Pandas*, Python has the data frame manipulation abilities of the program R, and finally the *scikit-learn* project has allowed for complex machine learning algorithm usage in Python. As a result, Python is usually the strong tool in any data scientist's toolbox.

Whitespace

In Python, unlike R and SQL, there exists the property that **white space** (the spacing created by indenting pieces of code) matters. If you do not indent a line or create white space at the correct points, your code will not run properly. This is different than other popular languages like Java and C, but it is used in Python to increase readability of the code.

Variables

Just like in mathematics, we use variables to assign a value to a name. There are some rules for this.

1) Variable names must start with a letter or an underscore
2) Following the initial letter or underscore, the remainder of the variable must consist of letters, underscores, or numbers

3) Variable names are case sensitive

Readability is key with variable names, and makes programming much easier. Try to use relevant and descriptive names.

Numbers
There are 2 main types of numbers in Python: integers and floats. Integers are signed, which means positive or negative. On the other hand, floats are real numbers (decimals).

Some integers are: -6, 0, 44, 2346
Some floats are: 9.2, -1.3, 8.0

Syntax: Simply write the name of the variable and set it equal to a value

In order to **manipulate**, or change, the value of numbers, use these operations: +,-,*,/.

Example Problem:
Make 2 variables of value 18 and 22 then calculate their average

Answer:

```
var1 = 18
var2 = 22
sum = var1 + var2
average = sum/2
```

Strings
Strings are continuous sets of characters wrapped in double or single quotes

For example, “Hello World!” and ‘This is a string’

Syntax: You can define strings the same way you defined numbers, but they must be in a set of quotes or double quotes. Use double quotes for strings and single quotes for chars.

Let’s set str = ‘Hello Everyone!’

We use the command **print()** to print out the value of the string. Inside the parenthesis, we use the + operator to **concatenate** or combine two different strings and the , operator to concatenate the string with a number.

```
print(str) #prints entire string
    Hello Everyone!

print(str + “ How is your day going?”) #concatenates (combines)2 strings
```

Hello Everyone! How is your day going?

Strings start at element 0, so "hello" has a length of 5, but 'h' is at element 0, 'e' is at 1, etc. We can access specific characters by including the elements we want in **[]** brackets after the name of the string.

Alternatively, you can use a technique called **string formatting**. This is a trick to insert a value directly into a string without breaking it up. For an integer, use %d and for a string, use %s at location of interest.

Example:

```
age = 20
name = input("Enter your name")
print("Hello %s! Nice to meet you! Just a guess, but are you %d?" % (name, age))
```

Lists

Lists are Python's most versatile of python's data types, lists are defined as a set of elements that can be of any data type. One key component of lists is that they are **mutable,** meaning they can be updated or changed at any time.

Syntax: Lists can have any name you like, just like with other variables, but the items of your list must be stored in ***square brackets***, and separated by ***commas***

Let's have

```
list = [ 'my', 123, 'list', 456, 'is', 789.0, 'this' ]

print(list) #prints full list
        [ 'my', 123, 'list', 456, 'is', 789.0, 'this' ]
print(list[2]) #prints 2nd element of list
        list
list.insert(0, 'WOW')  #insert 'WOW' into the 0 element spot
        ['WOW',  'my', 123, 'list', 456, 'is', 789.0, 'this' ]
list.remove('WOW') #removes 'WOW'
        [ 'my', 123, 'list', 456, 'is', 789.0, 'this', 9, 8, 'b' ]
```

Example Problem: create a list with your first name, age, last name, and year you were born in then remove the year you were born and add eye color instead

Answer:

```
list = ['Josh', 19, 'Eimer', 1998]
list.remove(1998) #or list.pop()
```

list.append('brown')

Tuples

Tuples are very similar to lists, with the key difference being that tuples are **immutable,** meaning that the cannot be changed once they are created.

Again, items in a tuple can be of any data type:

Syntax: Identical syntax to lists with the exception that we use ***parenthesis*** in place of square brackets.

As in strings and lists, individual elements or slices are accessed by referencing the element number, starting at 0, with square brackets

While we can't directly update a tuple, we can create a new tuple with all of the desired elements in it.

```
tup1 = ('abcd', 'efgh')
tup2 = (12, 34)
tup3 = tup1 + tup2
print(tup3) #prints ('abcd', 'efgh', 12, 34)
del(tup) #deletes entire tuple
```

Dictionaries

Dictionaries are another way of storing data. They are mutable, and are key-value pairs. Keys are generally numeric or a name and point to values, which can be of any type of data.

Syntax: In place of square brackets or parenthesis, we use **curly braces**. Inside of the curly braces, we use first have the key followed by a colon followed by the data. Use commas to separate pairs of data.

Dictionaries have a different style from lists and tuples. They have a key instead of element numbers.

```
dict = {
        'Iron Man': 'Tony Stark',
        'Captain America': 'Steve Rogers',
        'Superman': 'Clark Kent'
        'Spiderman': 'Peter Parker'
        }
```

To add an additional element:

```
dict['Deadpool'] = 'Wade Wilson'
```

To delete an element:

del dict['Superman']

To access a single element of a dictionary, use the key:

print(dict.get('Iron Man'))

To print the entire dictionary:

print(dict)

Conditionals (if, for, while)

A **boolean statement** is a statement that returns "true" or "false". This value comes from comparing to value using these symbols: <, >, >=, <=, ==, !=. These mean, in order from left to right: less than, greater than, greater than or equal to, less than or equal to, equals (TESTS EQUALITY DOES NOT SET A VALUE TO A VARIABLE), and not equal to.

It is also important to note that floats are imprecise, meaning .1 + .2 + .3 == .3 + .2 + .1 won't always return true.

"if" statements

This is where we start to see python-specific formatting on the text. Being careful of whitespace, an if statement is formatted like this:

Some rules:

1) Must end first line with a colon (:)
2) Must indent all lines to be run after the conditional

The statement after the word "if" is called a **conditional**. It has a value of either **True** or **False**. If it is true, the code after the conditional will run. If it is not true, the program will skip to the next line of code that is not indented.

Example:

```
var1 = 1
var2 = 3
if var1 < var2:
    print(var1, " is less than ", var2) #This runs if the if statement provides True
print(var2, " is less than ", var1) #This runs after the if statement
```

If-Else

Used to execute one of two different chunks of code. If the boolean evaluates to True, the "if" part of the code is run, and if it evaluates to False, the "else" part of the code is run.

```
var1 = 1
var2 = 3
```

```
if var1 > var2:
        print(var1, " is the bigger number") #This runs if the if statement is True
else:
        print(var2, " is the bigger number") #This runs if the if statement is False
printf("End of code") #This always runs, but after the if-else statement is finished
```

One can also have **nested** (another statement inside of the statement) if statements and nested else statements.

Example:

```
var1 = 8
var2 = 6
var3 = 4
var4 = 2
if var1 > var2:
        if var1 > var3:
                if var1 > var4:
                        print(var1, " is the biggest value")
print("The end")
```

There is a much better way to do this! We can have multiple conditional statements if we use **and** or **or** statements. Note that using parentheses to separate statements makes reading and debugging code much easier.

For **and**: If both statements are true, the conditional will evaluate to true.
For **or**: If one of or both statements are true, the conditional will evaluate to true.

Example of "and":
Let's say we want to decide if one number is bigger than all of the others...

```
var1 = 10
var2 = 8
var3 = 6
var4 = 4
if (var1 > var2) and (var1 > var3) and (var1 > var4):
        print(var1, "is the biggest value")
print("The end")
```

Alternatively, we can use an **elif** statement.

Example of "elif":
Let's say we want to decide which number out of 3 is the biggest number.

```
var1 = 10
```

```
var2 = 8
var3 = 6
if (var1 > var2) and (var1 > var3):
        largest_val = var1
elif (var2 > var1) and (var2 > var3):
        largest_val = var2
else:
        largest_val = var3
print("The largest number is: ", largest_val)
```

"For" Loops

Generally speaking, code is run **sequentially**, or line by line in order. Sometimes, you may want to run a block of code numerous times. To do this, we use a loop. The first type of loop is called a "for" loop.

Example:

```
num = 0
for x in range(0, 5): #The range ( , ) tells the loop how many times to run
        x += 1
        print(x)
```

Example:

```
my_list = [1, 2, 3, 4, 5, "END"]
for elt in my_list:
        print(elt)
```

"While" Loops

Executes while a statement is true. This statement is reevaluated each time the loop goes to repeat.

Example:

```
val = 0
while val <= 10:
        val += 1
        print("My new val is %d." % (val))
```