Python Syllabus

Basics

1. PYTHON SETUP
2. TUPLES Ch 2
3. TOKENS
4. DICTIONARIES Ch 4
5. FUNCTIONS Ch 1-11
6. DATA TYPES ch 3
7. MODULES
8. OPERATIONS
9. PYTHON DEBUGGER
10. BRANCHING
11. LOOPS
12. SORTING
13. SEARCHING
14. LISTS Ch 2

OOPS

1. CLASSES
2. OBJECTS
3. METHODS
4. OVERRIDING
5. INHERITANCE
6. DATA HIDING
7. EXCEPTION
8. HANDLING

Scripting

1. REGRESSION
2. EXPRESSION
3. STRINGS
4. FILE
5. COMMAND LINE
6. ARGS

Python : Beginning Phyton From Novice to Professional

13/11/2021

Operators: +, −, ×, /, //, %, **

// - This discards the Fractional part (puntos decimales)

```
>> 5/3
   1.666
>> 5//3
   1.0
```

% - remainder of the division

```
>> 10 % 3
   1.0
```

$$3 \overline{)10}\ 3 \quad \text{remainder } 1$$

** - exponential (power of)

```
>> 2 ** 3
   8
>> -3 ** 2
   -9
>> (-3) ** 2
   9
```

Hexadecimal Octal and Binary

```
>> 0x AF      - Hexadecimal
   175
>> 0o010      - Octal
   8
>> 0b 10110100 10 - Binary
   722
```

convert Decimals to

hex( )

oct( )

bin( )

Getting Input from the User: input() gives us a piece of text, or string.

Too convert it to a number we need the >>> int() command

```
>> input("x: ")
   x:34
>> input("y: ")
   y: 42
>> print(int(x) * int(y))
   1428
```

Some functions:

- power function:

  2**3 = pow(2,3) ; both do the same thing

Example:

```
>>> 10 + pow(2,3*5) /3.0
   10432.666
```

Other expressions are:

- abs(-10)
  10 — absolute value of a number
- round(2/3)
  1.0 — rounds floating-point number to the nearest integer.

Modules Math:

First write

```
>> import math
```

This will allow you to use other expressions like math.floor(), (round down); does same as int()

math.ceil(), rounds up ; math.sqrt()

If you are sure that you won't import more than one function with a given name (from different modules), you might not want to write the module name each time you call the function. Then you can use a variant of the `import` command.

```
>>> from math import sqrt
>>> sqrt(9)
3.0
```

After using the `from module import function`, you can use the function without its module prefix.

One may use variables to refer to functions (and most other things in Python)

```
>> foo = math.sqrt
   then use
>> foo(4)
   2
```

* For complex numbers we need to import "cmath" [Another module]
  Complex math

```
>> import cmath
>> cmath.sqrt(-1)
   1j
```

```
>>> (1+3j) * (9+4j)
(-3+31j)
```

Phython has complex number support for sum.

Modules

pandas : graphs and lists
sys : Systems
twepy

Chapter 2: Lists and Tuples (Sequences)

One can change a list, but not a tuple

Name and Age:

```
>>> edward = ['Edward Gumby', 42]
>>> john = ['John Smith', 50]
```

Sequence can contain other sequences

```
>>> database = [edward, john]
>>> database
[['Edward Gumby', 42], ['John Smith', 50]]
```

Indexing:

- All elements in a sequence are numbered - from zero and upward. You can access them individually with a number, like this:

```
>>> greeting = 'Hello'
>>> greeting[0]
'H'
>>> greeting[-1]
'o'
```

- If a function call returns a sequence, you can index it directly

I want the fourth number of the year inputed

```
>>> fourth = input('Year: ')[3]
Year: 2005
>>> fourth
'5'
```

When fourth called it will always get the last number

One can also put [-1] instead

Slicing

- Just as you use indexing to access individual elements, you can use slicing to access ranges of elements. You do this by using two indices, separated by a colon.

```
>>> tag = '<a href="http://www.python.org">Phyton web site</a>'
>>> tag[9:30]
'http://www.phyton.org'
>>> tag[32:-4]
'Python web site'
```

- The first index is the number of the first element you want to include. However, the last index is number of the first element after you slice. Consider the following:

```
>>> number = [1,2,3,4,5,6,7,8,9,10]
>>> number[3:6]  [4,5,6]
>>> number[0:1]  [1]
```

Shortcut: Access allways the last three elements of any size list

```
>>> numbers[7:10]  #If known size
[8,9,10]
```

Wrong:

```
>> numbers[-3:-1]
[8,9]

>> numbers[-3:0]
[]
```

correct

```
>> numbers[-3:]
[8,9,10]
```

Also one can do:

```
>>> numbers[:3]   [1,2,3]
```

and

```
>>> numbers[:]   [1,2,3,4,5,6,7,8,9,10]
```

Longer Steps: One can put another parameter to put the steps that will it move from one element to the other.

```
>>> number = [1,2,3,4,5,6,7,8,9,10]
>>> number[0:10:1]   # one element at a time
[1,2,3,4,5,6,7,8,9,10]
>>> number[0:10:2]
[1,3,5,7,9]
>>> number[8:6:3]
[4]
>>> number[8:7:3]
[4,7]
```

Other examples:

```
>>> numbers[::4]
[1,5,9]

>>> numbers[8:3:-1]
[9,8,7,6,5]

>>> numbers[10:0:-2]
[10,8,6,4,2]

>>> numbers[0:10:-2]
[]

>>> numbers[::-2]
[10,8,6,4,2]

>>> numbers[5::-2]
[6,4,2]

>>> numbers[:5:-2]
[10,8]
```

Adding and multiplying Sequences

```
>>> [1,2,3] + [4,5,6]
[1,2,3,4,5,6]
>>> 'Hello,' + 'world!'
'Hello,world!'
```

Cannot add strings and lists

```
>>'python' * 5
'pythonpythonpythonpythonpython'
>>> [42] * 5
[42,42,42,42,42]
```

- None, Empty Lists, and Initialization -

'None' is actually a value of nothing

one can make a sequence of nothing

```
>>> sequence = [None] * 10
[None, ...., None]
```

- Membership (Check if something is in a sequence) -

One can use Boolean operators to see if a value can be found in sequence. Boolean checks is something is true or false.

Examples:

```
>>permissions = 'rw'
>>> 'w' in permissions
True
>>> 'x' in permissions
False
```

```
>>> users = ['mlh', 'foo', 'bar']
>>> input('Enter your username: ') in users
Enter your username: mlh
True
```

* See example of Checking if a username and a Pin code is in a sequence in Indexing Problems

- Length, Minimum, and Maximum -

Preaty Straight Fouard

```
>>> numbers = [100, 34, 678]
>>> len(numbers)
3
```

```
>>> max(numbers)
678
>>> min(numbers)
34
```

Lists: Phython WorKhorse (Everything you can do with lists)

Commands

List

```
>>> list('Hello')
['H', 'e', 'l', 'l', 'o']
```

Join ??

```
''.join(somelist)
```

Item Assigment

```
>>> x = [1, 1, 1]
>>> x[1] = 2
[1, 2, 1]
```

Delete Element

```
>>> name = ['Alice', 'Beth', 'Juan']
>>> del names[2]
['Alice', 'Beth']
```

Slicing

```
>>> name = list('Pearl')
['P', 'e', 'r', 'l']
>>> name[1:] = list('ython')
>>> name
['P', 'y', 't', 'h', 'o', 'n']
```

- Inserting Slice

```
>>> numbers = [1, 5]
>>> numbers[1:1] = [2, 3, 4]
>>> numbers
[1, 2, 3, 4, 5]
```

- Deleting Slice

```
> numbers
[1, 2, 3, 4, 5]
>>> numbers[1:4] = []
[1, 5]
```

Methods

object.method(arguments)

append

```
>>> lst = [1, 2, 3]
>>> lst.append(4)
[1, 2, 3, 4]
```

clear

```
>>> lst = [1, 2, 3]
>>> lst.clear()
[]
```

copy

```
>>a = [1, 2, 3]
>b = a.copy()
>>b[1] = 4
[1, 4, 3]
```

- different lists

- same lists just

```
>> b = a
>> b[1] = 4
>>> a
[1, 4, 3]
```

count

```
>>> ['to', 'be', 'or', 'not', 'to', 'be'].count('to')
2
>>> x = [[1, 2], 1, 1, [2, 1, [1, 2]]]
>>> x.count(1)
2
>>> x.count([1, 2])
1
```

extend

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 5, 6]
```

careful

```
>>> a + b
[1, 2, 3, 4, 5, 6]
>>> a
[1, 2, 3]
```

index

```
>>> knights = ['We', 'are', 'the', 'knights', 'who', 'say', 'ni']
>>> knights.index('who')
4
```

pop (removes last element by default)

```
>>x = [1, 2, 3]
>>> x.pop()
3
>>> x.pop(0)
1
>>> x
[2]
```

stack can be created with LIFO

x.append(x.pop)

* FIFO

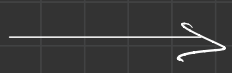
x.insert(x.pop) or x.append(x.pop(0))

insert

```
>>> numbers = [1, 2, 3, 5, 6, 7]
>>> numbers.insert(3, 'four')
>>> numbers
[1, 2, 3, 'four', 5, 6, 7]
```

remove

```
>>> x = ['to', 'be', 'or', 'be']
>>> x.remove('be')
['to', 'or', 'be']
```



reverse

```
>>x = [1, 2, 3]
>> x. reverse()
>>> x
 [3, 2, 1]
```

sort and sorted function

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> x. sort()
 [1, 2, 4, 6, 7, 9]
```

sorted will always return a list

```
>>> y = sorted(x)
>> x
 [4, 6, 2, 1, 7, 9]
>>> y
 [1, 2, 4, 6, 7, 9]
```

sort examples

```
>>x = ['aardvark', 'abalone', 'acme', 'add', 'aerate']
>> x. sort(key = len) # Will sort by the length of the word
```

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> x. sort(reverse = True)
>>> x
 [9, 7, 6, 4, 2, 1]
```

Tuples: Immutable Sequences

Written with parentesis

```
>>> 1, 2, 3
 (1, 2, 3)
>>> (1, 2, 3)
 (1, 2, 3)
```

Empty tuple

```
>>> ()
 ()
```

Tuples with one value

```
>>> 42,
 (42,)
>>> (42,)
 (42,)
```

Some examples

```
>>> 3 * (40 + 2)
 126
```

, but

```
>>> 3 * (40 + 2,)
 (42, 42, 42)
```

Tuple function

```
>>>tuple ([1,2,3])
 (1,2,3)
>>>tuple ('abc')
 ('a', 'b', 'c')
```

One can only create them and access their elements

```
>>> x = 1, 2, 3
>>> x[1]
 2
>>> x[0:2]
 (1, 2)
```

Chapter 2 : Summary

New Functions in This Chapter

Function	Description
`len(seq)`	Returns the length of a sequence
`list(seq)`	Converts a sequence to a list
`max(args)`	Returns the maximum of a sequence or set of arguments
`min(args)`	Returns the minimum of a sequence or set of arguments
`reversed(seq)`	Lets you iterate over a sequence in reverse
`sorted(seq)`	Returns a sorted list of the elements of seq
`tuple(seq)`	Converts a sequence to a tuple

Chapter 3: Working with Strings

All standard sequences operations (indexing, slicing, multiplication, membership, length, minimum and maximum) work with strings.

String replacement

```
>>> format = "Hello, %s. %s enough for ya?"
>>> values = ('world', 'Hot')  # Tuple
>>> format % values
'Hello, world. Hot enough for ya?'
```

The `%s` format string are called conversion specifiers. They mark places where the values are to be inserted. The `s` means that the values should be formatted as if they were strings; if they aren't, they'll be converted with `str`

One can also use:

`%.3f` ~ value as a floating-point number with three decimals

String Methods (For more see Appendix B)

One can use the string module to use these string methods that will be available

BUT STRING ISN'T DEAD

Even though string methods have completely upstaged the `string` module, the module still includes a few constants and functions that *aren't* available as string methods. The following are some useful constants available from `string`[2]:

- `string.digits`: A string containing the digits 0–9
- `string.ascii_letters`: A string containing all ASCII letters (uppercase and lowercase)
- `string.ascii_lowercase`: A string containing all lowercase ASCII letters
- `string.printable`: A string containing all printable ASCII characters
- `string.punctuation`: A string containing all ASCII punctuation characters
- `string.ascii_uppercase`: A string containing all uppercase ASCII letters

Despite explicitly dealing with ASCII characters, the values are actually (unencoded) Unicode strings.

- String Methods : that can be use without the module

- center (centers the string by padding it on either side)

```
>>> "Juan is learning". center(39)
'          Juan is learning          '
>>> "Juan is learning". center(39, "*")
'**********Juan is learning**********'
```

- find (Finds a substring within a larger string. Returns the leftmost index where the substring is found. If not found, -1 is returned.)

```
>>> title = "Monty Python's Flying Circus"
>>> title.find ('Monty')
0
>>> title.find ('Phyton')
6
>>> title.find ('Flying')
15
```

```
>>> title.find ('Zirquss')
-1
```

One can also supply a starting point for a search and an ending point.

```
>>> subject = ' $$$ Get rich now !!! $$$'
>>> subject.find ('$$$')
0
>>> subject.find ('$$$', 1)
20
```

```
>>> subject.find ('!!!')
16
>>> subject.find ('!!!', 0, 16) # start and end
-1
```

- Join (

Phyton Data Classes 3.7

When implementing data based or data focused classes we want to use data classes *best practice

Example w/ classes first:

```
class Investor:                    # We want him with age, name, cash amount

    def __init__(self, name: str, age: int, cash: float):
        self.name = name
        self.age = age
        self.cash = cash
```

} basic constructor

```
i1 = Investor("John", 25, 9000)
i2 = Investor("Ana", 30, 12000)
i3 = Investor("Bob", 70, 800000)
```

If we print (i1) No information will be presented

So manually, we have to implement the representation by:

```
>>class Investor:                 # We want him with age, name, cash amount

    def __init__(self, name: str, age: int, cash: float):
        self.name = name
        self.age = age
        self.cash = cash

    def __repr__(self):
        return f"Name: {self.name}"
```

by adding this

```
>>i1 = Investor("John", 25, 9000)
>>i2 = Investor("Ana", 30, 12000)
>>i3 = Investor("Bob", 70, 800000)

>>print(i1)
  Name: John
```

- Basically

Now with DataClasses everything is simplified
Look at code DataClasses Inv. py for better understanding

Code goes as:

```
from dataclasses import dataclass

@dataclass
class Investor:
        name: str
        age: int
        cash: float

i1 = Investor("John", 25, 9000)
i2 = Investor("Ana", 30, 12000)
i3 = Investor("Bob", 70, 800000)

# Now we can compare
```

OOP Person Task

Phyton class atribute

def __init__: