Introduction

Welcome to the world of Pythonista! Let's jump into course now. This course covers:

* Functions
* List Comprehensions
* Iterators and Generators
* Classes and Objects in Python
* Closures and Decorators
* Descriptors and Properties

Additional Information

* All the examples shown in this course are executed on **Python 3.5.0**.
* You can try all the examples shown and practise exercises @ https://repl.it/languages/python3

**Learn, Explore, Be a Pythonista...**

* A function is a piece of code, capable of performing a similar task repeatedly.
* It is defined using def keyword in python.

**Syntax of a function :**

def <function\_name>(<parameter1>, <parameter2>, ...):

'Function documentation'

function\_body

return <value>

* Parameters, return expression and documentation string are optional.

**Sample function - square**

def square(x):

'Returns square of a number.'

return x\*\*2

##### **Common Built-in Functions**

There are many built-in functions Python offers. Here are some of them listed.

* **len** : Returns the length of an object.
  + e.g : len('hello') -> 5
* **type** : Returns the type of an object.
  + e.g : type([2,3]) -> <type list>
* **range** : Returns a iterator of a number sequence.
  + e.g : list(range(10, 13)) -> [10, 11, 12]
  + list(range(3)) -> [0, 1, 2]

##### **Iterators**

* An Iterator is an object, which allows a programmer to traverse through all the elements of a collection, regardless of its specific implementation.
* Values of an Iterator can be accessed only once and in sequential order.

#### Sample Iterator

x = [6, 3, 1]

s = iter(x)

print(next(s)) # -> 6

print(next(s)) # -> 3

print(next(s)) # -> 1

print(next(s)) # -> StopIteration Error

##### **List Comprehensions**

* Alternative to **for loops**.
* More concise, readable, efficient and mimic functional programming style.
* Used to:
  + Apply a method to all or specific elements of a list, and
  + Filter elements of a list satisfying specific criteria.

#### Example

x = [6, 3, 1]

y = [ i\*\*2 for i in x ] # List Comprehension expression

print(y) # -> [36, 9, 1]

##### **Generators**

* A Generator object is an iterator, whose values are created at the time of accessing them.
* A generator can be obtained either from a generator expression or a generator function.

#### Example

x = [6, 3, 1]

g = (i\*\*2 for i in x) # generator expression

print(next(g)) # -> 36

5 of 6

Give a Try P3 # 1

Let us consider the multi-line string zenPython shown below. We will be using this to solve upcoming exercises.

zenPython = '''

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than \*right\* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

'''

**Task 1**

* Convert the entire string **zenPython** into a list of words.
* Capture the words in the list, **words**.
  + Hint: Use **split** method of strings.
* Print the length of list, **words**.

**Instructions**

1. Open app.py file using vim editor vi app.py
2. Press **i** for inserting content into file.
3. Copy the zenPython string into file.
4. Write the necessary python code required for solving a task.
5. Save the file by pressing **Esc** key and typing **:wq** in editor.
6. Execute the script with command python3 app.py | tee output.txt
7. Click Continue to Move to next Task.

**Task 2**

* Now, remove the word separating characters (such as , . - \* ! and space) from each of the word, present in list **words**.
* Store the obtained result again in the list **words**.
  + Hint: Use List comprehensions and **strip** method of strings.
* Print the 3rd indexed element of list, **words**.

**Instructions**

'''

Convert the entire string zenPython into a list of words. Capture the words in the list - words.

Hint: Use split method of strings. You may type python3.5 to use the latest version for the exercise

'''

zenPython = '''

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than \*right\* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

'''

#words=zenPython.split('\n')

words=zenPython.split(' ')

print(words)

print(len(words))

'''

Now, remove the flanking characters (such as , . - \* ! and space) from each of the word, present in list words.

Store the obtained result again in the list words.

Hint: Use List comprehensions and strip method of strings. You may type python3.5 to use the latest version for the exercise

'''

def strip(s):

news=''

ignore=set(',.-\*! \n')

for letter in s:

if letter not in ignore:

news+=letter

return news

words=[strip(s) for s in words]

print(words)

print(len(words))

words=[s.lower() for s in words]

print(words)

unique\_words=set(words)

print(unique\_words)

print(len(unique\_words))

word\_frequency=dict()

for u in unique\_words:

word\_frequency[u]=words.count(u)

print(word\_frequency)

'''for f in word\_frequency:

print(type(word\_frequency[f]))'''

frequent\_words = {f:word\_frequency[f] for f in word\_frequency if word\_frequency[f]>5}

print(frequent\_words)

'''for u in word\_frequency:

if u.value>5:

frequent\_words[u.key]=u.value'''

'''

Write a if-elif-else blocks to determine grade obtained by a student based on the total average obtained. Use the below criteria to determine the grade.

if total average >= 90, display "Distinct"

if in range [60 -90), display "Above average"

if in range [40 -60), display "Average"

else display "Fail"

Also determine the grade of a student with average score 68.3

'''

'''a = -10

if a:

print("a's value")

else:

print("Sorry nothing will get printed")

count = 0

while count < 2:

print (count, " is less than 2")

count = count + 2

else: print (count, " is not less than 2")

while False:

print('in while')

else: print('in else')

average=int(68.3)

if average >= 90:

print("Distinct")

elif average in range(60,90):

print("Above average")

elif average in range(40,60):

print("Average")

else :print("Fail")

'''

'''Consider the string 's' having the value 'tata consultancy services limited'

Determine the no. of vowels present in it, using for loop. Store the number in variable 'count' and print it.

'''

'''

s='tata consultancy services limited'

count = 0

vowels = set("aeiou")

for letter in s:

if letter in vowels:

count += 1

print(count)

'''

'''

Consider the string 's' having the value 'tata consultancy services limited'

Determine the no. of vowels present in it, using While loop. Store the number in variable 'count' and print it.

'''

'''

s='tata consultancy services limited'

count = 0

vowels = set("aeiou")

i=len(s)-1

while i >= 0:

if s[i]in vowels:

count += 1

i-=1

print(count)

for letter in s:

if letter in vowels:

count += 1

'''

'''

Create an empty Dictionary, 'd1' using 'dict' function

print 'd1'

Create a Dictionary 'd2' with Key values p-play , t-talk in the same order

print 'd2'.

Add two new key values : v-vibe, d-docs in the same order

print 'd2'

Remove the key value pair, 'v' - vibe, from 'd2'

print 'd2'

'''

'''

d1=dict()

print(d1)

d2=dict(p='play',t='talk')

print(d2)

d2['v']='vibe'

d2['d']='docs'

print(d2)

del(d2['v'])

print(d2)

'''

'''Create two sets 'a', and 'b' with following values.

a = ('10','20','30','40')

b = ('30','60')

Determine the following

Union; store the output in variable 'u' and print it.

Intersection; store the output in variable 'i' and print it.

Difference between set 'a' and 'b'; store the output in variable 'd' and print it.

Symmetric difference; store the output in variable 'sd' and print it.

'''

'''

a=set(('10','20','30','40'))

b = set(('30','60'))

u=a.union(b)

print(u)

i=a.intersection(b)

print(i)

d=a.difference(b)

print(d)

sd=a.symmetric\_difference(b)

print(sd)

'''

'''Write a script to generate the following list of numbers using range.

Generate the list 'k1' having first five natural numbers.

Print 'k1'

Generate the list 'k2' having five continuous numbers starting from 10.

Print 'k2'

Generate the list 'k3' having even numbers between 10 and 20 (including both of them too).

Print 'k3'

Generate the list 'k4' having numbers from 100 to 1 in decreasing order, which are also multiple of 25.

Print 'k4'

'''

'''k1=list(range(1,6))

print(k1)

k2=list(range(10,15))

print(k2)

k3=list(range(10,21,2))

print(k3)

k4=list(range(100,0,-1\*25))

print(k4)

'''

'''Create a list 'k' having seven characters from 'a' to 'g'. Perform the following tasks and print the outputs in separate lines.

Print alternative characters, using slicing option, starting from 'a'.

Print alternative characters among the first 4 elements alone

Print only the odd indexed elements of list 'k'

'''

'''k=['a','b','c','d','e','f','g']

print(k[::2])

print(k[:4:2])

print(k[1::2])'''

'''Create an empty tuple 'tup1' using tuple operation.

Print 'tup1'

Create another tuple 'tup2', by passing 'Welcome' string as argument to tuple function.

Print 'tup2'

Find and print the count of character 'e' in 'tup2'.

Determine the index of character 'e' in 'tup2' and print it.

Find the length of tuple 'tup2' and print it.'''

'''

tup1=()

print(tup1)

tup2=('Welcome')

print(tup2)

print(tup2.count('e'))

print(tup2.index('e'))

print(len(tup2))

'''

'''s="INfinity"

print(s.isalpha())

print(s.isdigit())

print(len(s))

print(s.upper())

print(s.lower())

print(s.count('i'))

print(s.index('t'))'''

'''Create an empty list 'emplist1' using list operation.

Print 'emplist1'

Append to empty list 'emplist1' created above with element 9.

Append another element 10 to 'emplist1'

Print 'emplist1'

Create an empty list 'emplist2' using [].

Print 'emplist2'

Extend the empty list 'emplist2' created above with elements 'a', 'b', 'c'.

Print 'emplist2'

Extract the last element of 'emplist2' and assign it to variable 'e'.

Print 'emplist2'

Print the variable 'e'.

emplist1=[]

print(emplist1)

emplist1.append(9)

emplist1.append(10)

print(emplist1)

emplist2=[]

print(emplist2)

emplist2.extend(['a', 'b', 'c'])

print(emplist2)

e=emplist2[2]

print(emplist2)

print(e)'''

words=[s.strip(‘,.-\*! ‘) for s in words]

zenPython = '''

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than \*right\* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

'''

words=[]

words=zenPython.split(" ")

print(len(words))

#print('before',words)

words=[s.strip(',.-\*! ') for s in words]

#print('after',words)

print(words[2])

**Problem Statement**

* Define the generator function fib\_gen, which is capable of yielding values of **infinite fibonacci series.**
* The first two values of fibonacci series are 0, and 1.
* Create a generator fs from fib\_gen function.
* Call next method on fs and print the returned value.
* Repeat the above step 3 more times, and display each returned value in a separate line.

**Instructions**

1. Open app.py file using vim editor vi app.py
2. Press **i** for inserting content into file.
3. Write the necessary python code required for solving a task.
4. Save the file by pressing **Esc** key and typing **:wq** in editor.
5. Execute the script with command python3 app.py | tee output.txt
6. Click Summary to view Status

def fib\_gen():

a,b=0,1

while a<4:

yield a

a,b=b,a+b

fs=fib\_gen()

print(next(fs))

print(next(fs))

print(next(fs))

print(next(fs))

**Problem Statement**

* Define the generator function factorial\_gen, which is capable of yielding factorial values of natural numbers.
* Create a generator fs from factorial\_gen.
* Ensure, the first two values to be yielded by fs are 1 and 1, corresponding to factorial of 0 and 1 respectively.
* Call next method on fs and print the returned value.
* Repeat the above step 3 more times, and display each returned value in a separate line.

Introduction to OOP

Object-oriented programming can model real-life scenarios and suits developing large and complex applications.

***Object***

In real life, an object is something that you can sense and feel. For example **Toys, Bicycles, Oranges** and more.

However in Software development, an **object** is a non tangible entity, which holds some data and is capable of doing certain things.

##### **Defining Classes**

***Class***

A Class is a **template** which contains

* **instructions** to build an object.
* **methods** that can be used by the object to exhibit a specific behaviour.

class keyword is used to define a class in Python.

#### Syntax

class <ClassName>(<parent1>, ... ):

class\_body

#### Example

class Person:

pass

* Above example defines Person class without any body.

##### **Creating Objects**

* An object is created by calling the class name followed by a pair of parenthesis.

class Person:

pass

p1 = Person() # Creating the object 'p1'

print(p1) # -> '<\_\_main\_\_.Person object at 0x0A...>'

* The output of print on object p1, tell you what class it belongs to and hints on memory address it is referenced to.

##### **Setting Attributes**

* You can set attributes, **one a time**, to an instantiated object and access it using the dot notation.
* The value which is set to an attribute can be anything: a Python primitive, a built-in data type, another object. It can even be a function or a class.

### Example

class Person:

pass

p1 = Person()

p1.fname = 'Jack'

p1.lname = 'Simmons'

print(p1.fname, '-', p1.lname) # -> 'Jack - Simmons'

##### **Setting Attributes Contd..**

* You can also set multiple attributes, **at once**, by defining the initializer method, **\_\_init\_\_**, inside the class.
* This method is called by default, during an object creation.
* It takes values passed inside the parenthesis, during an object creation, as it's arguments.
* It also takes self as the first argument, which refers to the current object.

##### **Setting Attributes Example**

* In the following example, Person class sets two attributes using \_\_init\_\_ method.

class Person:

def \_\_init\_\_(self, fname, lname):

self.fname = fname

self.lname = lname

p1 = Person('George', 'Smith')

print(p1.fname, '-', p1.lname) # -> 'George - Smith'

##### **Documenting a Class**

Each class or a method definition can have an optional first line, known as docstring.

#### Example

class Person:

'Represents a person.'

def \_\_init\_\_(self, fname, lname):

'Initialises two attributes of a person.'

self.fname = fname

self.lname = lname

##### **Understanding a Class**

Once documented, you can load the script into an interactive interpreter and run help command on Person class.

>>>help(Person)

Help on class Person in module \_\_main\_\_:

class Person(builtins.object)

| Represents a person.

|

| Methods defined here:

|

| \_\_init\_\_(self, fname, lname)

| Initialises two attributes of a person.

|

... and more

##### **Inheritance in Python 3**

* Inheritance describes is a kind of relationship between two or more classes, abstracting common details into super class and storing specific ones in the subclass.
* To create a child class, specify the parent class name inside the pair of parenthesis, followed by it's name.

#### Example

class Child(Parent):

pass

* Every child class inherits all the behaviours exhibited by their parent class.

##### **Inheritance**

* In Python, every class uses inheritance and is inherited from object by default.
* Hence, the below two definitions of MySubClass are same.

#### Definition 1

class MySubClass:

pass

#### Definition 2

class MySubClass(object):

pass

* object is known as **parent or super class.**
* MySubClass is known as **child or subclass or derived class.**

##### **Inheritance in Action**

class Person:

def \_\_init\_\_(self, fname, lname):

self.fname = fname

self.lname = lname

class Employee(Person):

all\_employees = []

def \_\_init\_\_(self, fname, lname, empid):

Person.\_\_init\_\_(self, fname, lname)

self.empid = empid

Employee.all\_employees.append(self)

* Employee class is derived from Person.

##### **Inheritance in Action**

p1 = Person('George', 'smith')

print(p1.fname, '-', p1.lname)

e1 = Employee('Jack', 'simmons', 456342)

e2 = Employee('John', 'williams', 123656)

print(e1.fname, '-', e1.empid)

print(e2.fname, '-', e2.empid)

#### Output

George - smith

Jack - 456342

John - 123656

* In the above example, Employee class utilizes \_\_init \_\_ method of the parent class Person to create its object.

##### **Extending Built-in Types**

* Inheritance feature can be also used to extend the built-in classes like list or dict.
* The following example extends list and creates EmployeesList, which can identify employees, having a given search word in their first name.

#### Example 2

class EmployeesList(list):

def search(self, name):

matching\_employees = []

for employee in self:

if name in employee.fname:

matching\_employees.append(employee.fname)

return matching\_employees

##### **Extending Built-in Types**

Update Card Text  
Select Card Type Extending Built-in Types Card name \* Please select a file or Click here to paste link Card content star\_rate EmployeesList object can be used to store all employee objects, just by replacing statement all\_employees = [] with all\_employees = EmployeesList().

#### Example 2

class Employee(Person):

all\_employees = EmployeesList()

def \_\_init\_\_(self, fname, lname, empid):

Person.\_\_init\_\_(self, fname, lname)

self.empid = empid

Employee.all\_employees.append(self)

e1 = Employee('Jack', 'simmons', 456342)

e2 = Employee('George', 'Brown', 656721)

print(Employee.all\_employees.search('or'))

#### Output

​

['George']

​

43616:15 Web Mobile Extending Built-in Types EmployeesList object can be used to store all employee objects, just by replacing statement all\_employees = [] with all\_employees = EmployeesList().

Example 2

class Employee(Person):

all\_employees = EmployeesList()

def \_\_init\_\_(self, fname, lname, empid):

Person.\_\_init\_\_(self, fname, lname)

self.empid = empid

Employee.all\_employees.append(self)

e1 = Employee('Jack', 'simmons', 456342)

e2 = Employee('George', 'Brown', 656721)

print(Employee.all\_employees.search('or'))

Output ['George']

##### **Polymorphism in Python 3**

* Polymorphism allows a subclass to override or change a specific behavior, exhibited by the parent class

##### **Polymorphism Example**

In the below shown example, you will find

* Improvised Employee class with two methods getSalary and getBonus.
* Definition of ContractEmployee class derived from Employee. It overrides functionality of getSalary and getBonus methods found in it's parent class Employee.

#### Example

class Employee(Person):

all\_employees = EmployeesList ()

def \_\_init\_\_(self, fname, lname, empid):

Person.\_\_init\_\_(self, fname, lname)

self.empid = empid

Employee.all\_employees.append(self)

def getSalary(self):

return 'You get Monthly salary.'

def getBonus(self):

return 'You are eligible for Bonus.'

##### **Polymoprhism Example Contd.**

class ContractEmployee(Employee):

def getSalary(self):

return 'You will not get Salary from Organization.'

def getBonus(self):

return 'You are not eligible for Bonus.'

e1 = Employee('Jack', 'simmons', 456342)

e2 = ContractEmployee('John', 'williams', 123656)

print(e1.getBonus())

print(e2.getBonus())

#### Output

You are eligible for Bonus.

You are not eligible for Bonus.

##### **Abstraction and Encapsulation**

* Abstraction means working with something you know how to use without knowing how it works internally.
* Encapsulation allows binding data and associated methods together in a unit i.e class.
* These principles together allows a programmer to define an interface for applications, i.e. to define all tasks the program is capable to execute and their respective input and output data.
* A good example is a television set. We don’t need to know the inner workings of a TV, in order to use it. All we need to know is how to use the remote control (i.e the interface for the user to interact with the TV).

##### **Abstraction**

##### **Encapsulation**

3 of 7

##### **Abstracting Data**

* Direct access to data can be restricted by making required attributes or methods private, **just by prefixing it's name with one or two underscores.**
* An attribute or a method starting with:
  + no underscores is a public one.
  + a single underscore is private, however, still accessible from outside.
  + double underscores is strongly private and not accessible from outside.

##### **Abstraction and Encapsulation Example**

* empid attribute of Employee class is made private and is accessible outside the class only using the method getEmpid.

class Employee(Person):

all\_employees = EmployeesList()

def \_\_init\_\_(self, fname, lname, empid):

Person.\_\_init\_\_(self, fname, lname)

self.\_\_empid = empid

Employee.all\_employees.append(self)

def getEmpid(self):

return self.\_\_empid

##### **Abstraction and Encapsulation Example Contd..**

e1 = Employee('Jack', 'simmons', 456342)

print(e1.fname, e1.lname)

print(e1.getEmpid())

print(e1.\_\_empid)

#### Output

Jack simmons

456342

AttributeError: Employee instance has no attribute '\_\_empid'

**Task 1**

* Define the class Point that represents x, y, and z coordinates of 3D coordinate system.
  + Hint : Define the initializer method, \_\_init\_\_ that takes three values and assigns them to attributes x, y and z respectively.
* Now create an object p1 = Point(4, 2, 9) and print it using the statement print(p1).