Python
For Data Science

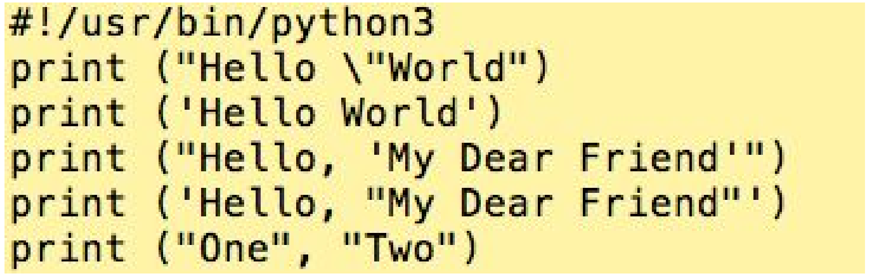
Course Infrastructure

- Python 3
  - Develop Scripts
  - Interactive Shell
- Jupyter - would be preffered way
  - Effective to learn using Pandas, Numpy, Matlab

Topics

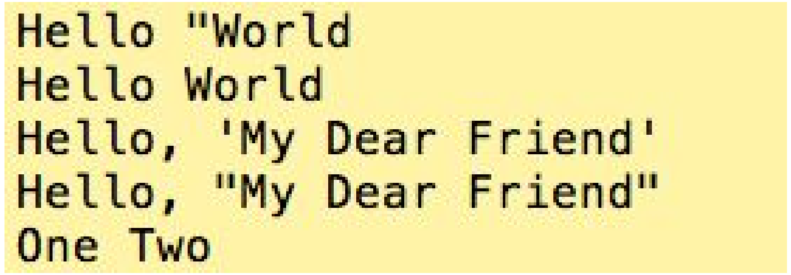
- Hello Python
- Comments
- Mathematics
- Variables
- Data Types
- Making Decisions
- Strings
- Python Lists
- For, While - Loops
- Lists Revisits
- Handling Errors
- Functions
- Modules
- Packages
- Dictionaries
- Classes
- Iterators
- String Formatting
- Regular Expressions

Hello to Python



```
#!/usr/bin/python3
print ("Hello \"World")
print ('Hello World')
print ("Hello, 'My Dear Friend'")
print ('Hello, "My Dear Friend"')
print ("One", "Two")
```

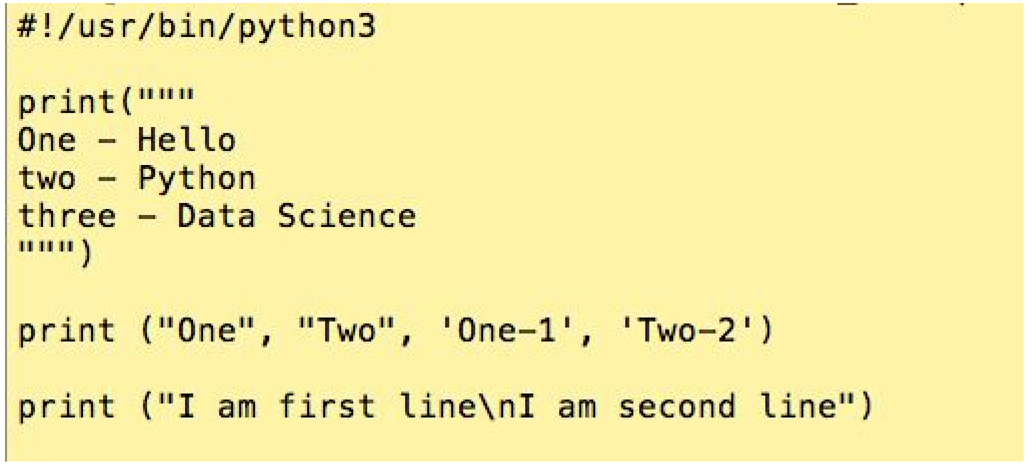
output



```
Hello "World
Hello World
Hello, 'My Dear Friend'
Hello, "My Dear Friend"
One Two
```

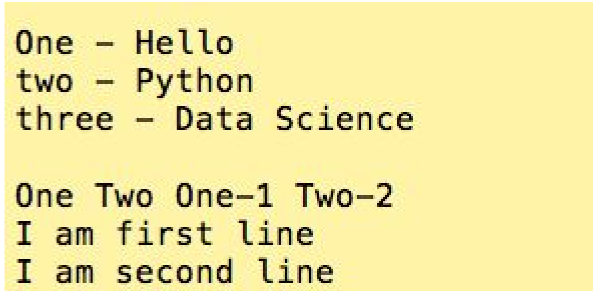
hello.py

Hello to Python (Cont…)



```
#!/usr/bin/python3

print("""
One - Hello
two - Python
three - Data Science
""")

print ("One", "Two", 'One-1', 'Two-2')

print ("I am first line\nI am second line")
```

output

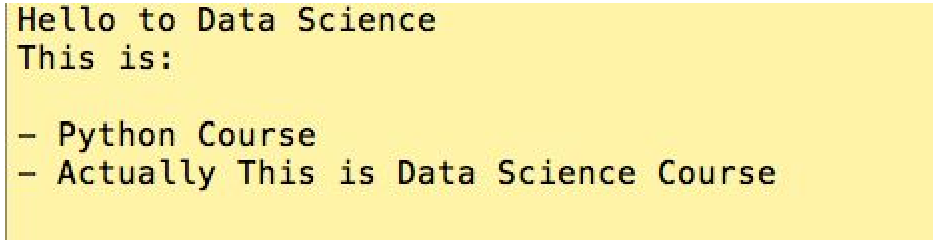


```
One - Hello
two - Python
three - Data Science

One Two One-1 Two-2
I am first line
I am second line
```

hello_multiple.py

Assignment - 1

- Write “hello_datascience.py” Script
- It should print following output

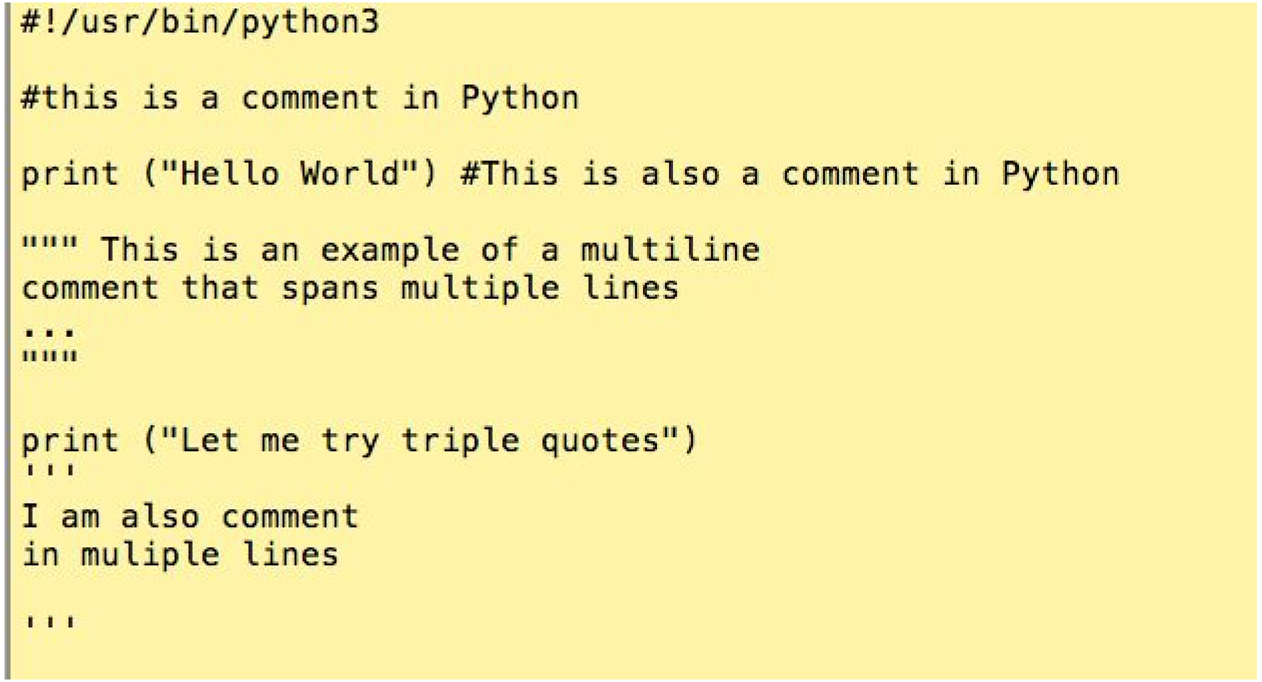


```
Hello to Data Science
This is:

- Python Course
- Actually This is Data Science Course
```

Topics

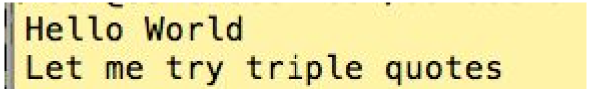
- Hello Python
- Comments
- Mathematics
- Variables
- Data Types
- Making Decisions
- Strings
- Python Lists
- For, While - Loops
- Lists Revisits
- Handling Errors
- Functions
- Modules
- Packages
- Dictionaries
- Classes
- Iterators
- String Formatting
- Regular Expressions

Comments



```
#!/usr/bin/python3

#this is a comment in Python

print ("Hello World") #This is also a comment in Python

""" This is an example of a multiline
comment that spans multiple lines
...
"""


print ("Let me try triple quotes")
'''
I am also comment
in muliple lines

'''
```

output



```
Hello World
Let me try triple quotes
```

comments.py

Mathematics

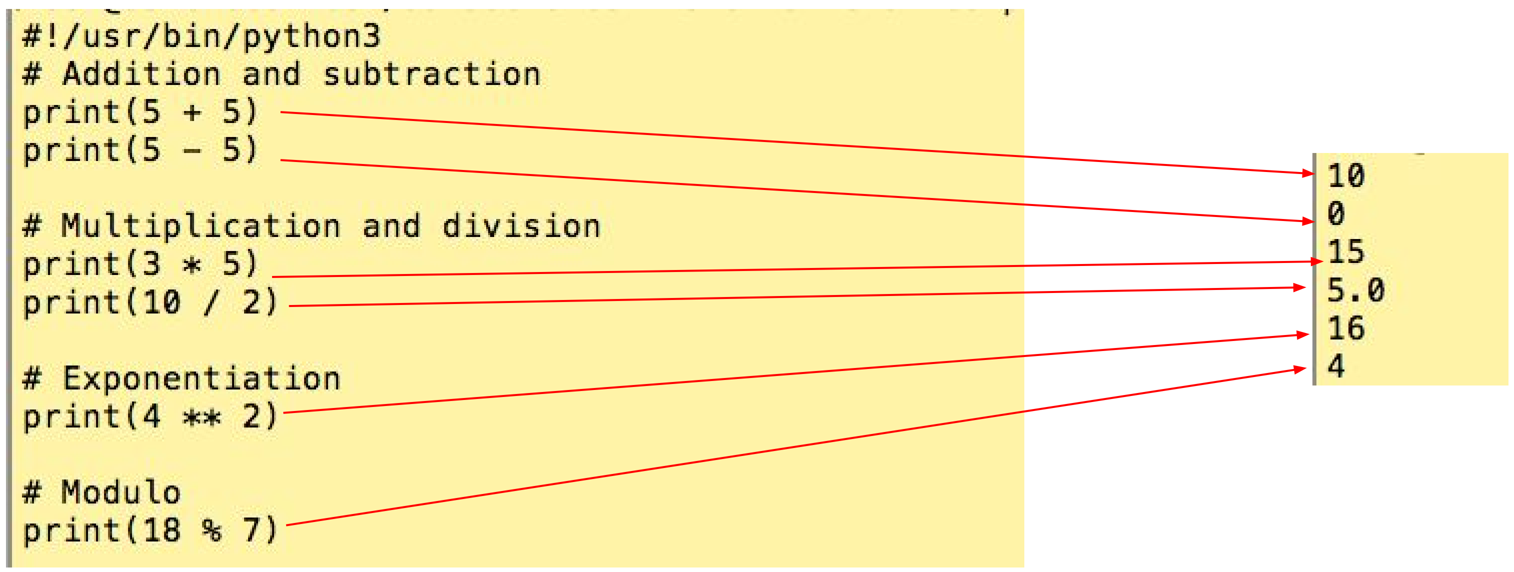
Python is perfectly suited to do Mathematics

- addition +
- subtraction -
- multiplication *
- division /

There is also support for more advanced operations such as

- Exponentiation **
- Modulo: %

Mathematics (Cont...)



```
#!/usr/bin/python3
# Addition and subtraction
print(5 + 5)
print(5 - 5)

# Multiplication and division
print(3 * 5)
print(10 / 2)

# Exponentiation
print(4 ** 2)

# Modulo
print(18 % 7)
```

```
10
0
15
5.0
16
4
```

mathematics.py

Assignment 2

Suppose you have Rs 1000. You have invested that with a 10% return per year scheme

After one year, you earnings = $1000 \times 1.1 = 1100$

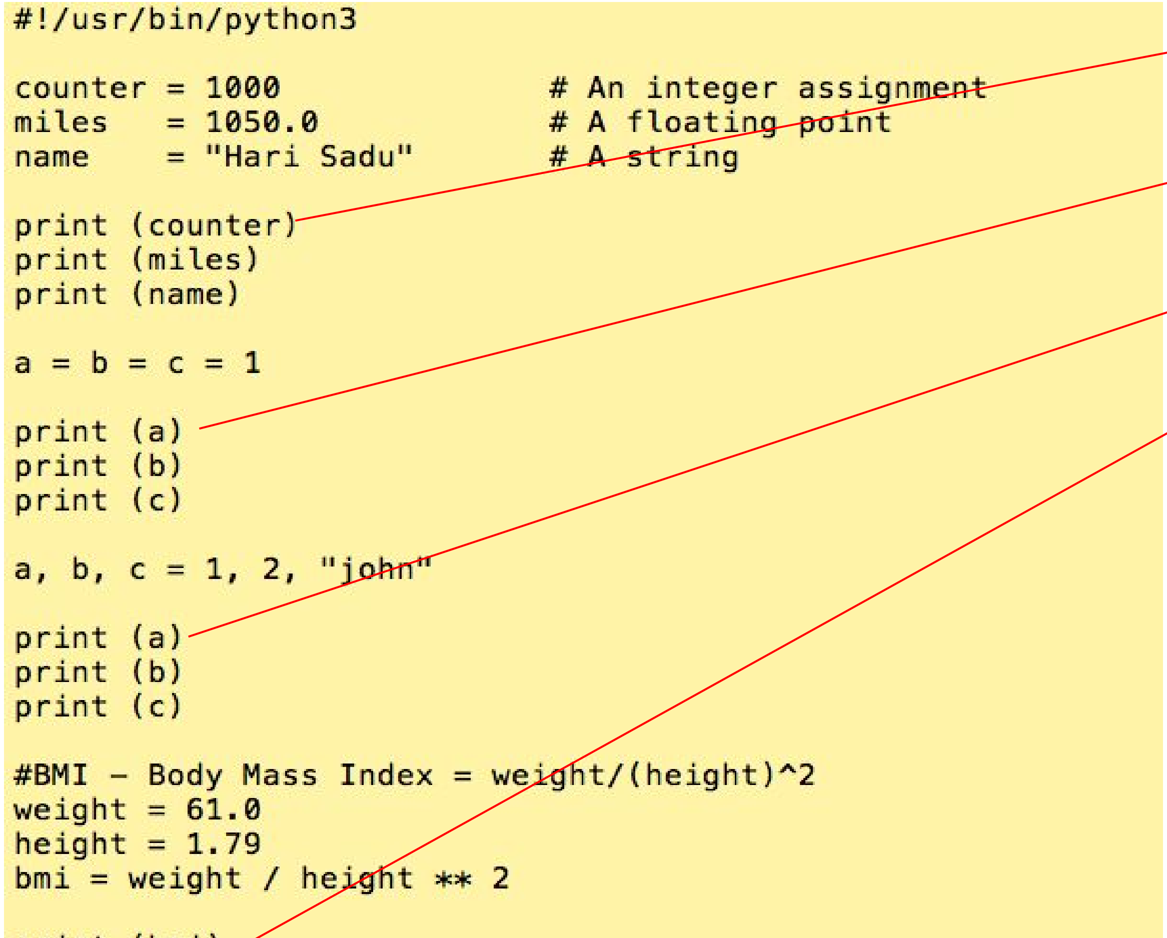
After two years, earnings = $1000 \times 1.1 \times 1.1 = 1210$

- How much money you end up earning after 5 years?
- How much money you end up earning after 10 years?

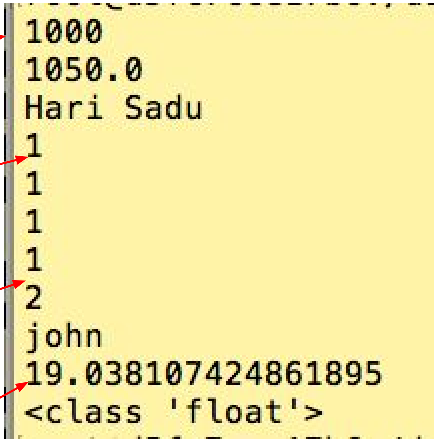
Topics

- Hello Python
- Comments
- Mathematics
- Variables
- Data Types
- Making Decisions
- Strings
- Python Lists
- For, While - Loops
- Lists Revisits
- Handling Errors
- Functions
- Modules
- Packages
- Dictionaries
- Classes
- Iterators
- String Formatting
- Regular Expressions

variables



```
#!/usr/bin/python3

counter = 1000                # An integer assignment
miles   = 1050.0              # A floating point
name    = "Hari Sadu"         # A string

print (counter)
print (miles)
print (name)

a = b = c = 1

print (a)
print (b)
print (c)

a, b, c = 1, 2, "john"

print (a)
print (b)
print (c)

#BMI – Body Mass Index = weight/(height)^2
weight = 61.0
height = 1.79
bmi = weight / height ** 2
```



```
1000
1050.0
Hari Sadu
1
1
1
1
2
john
19.038107424861895
<class 'float'>
```

variables.py

Assignment - 3

- Create a variable *savings* equal to 1000
- Create a variable *interest*, equal to 1.10.
- Use *savings* and *interest* to calculate the amount of money you end up earning with after 8 years.
- Store the result in a new variable, *earnings*
- Print out the value of *earnings*

Word on Data Types

- Numbers
  - Int
  - float
- String
- Boolean - True, False
- List
- Tuple
- Dictionary

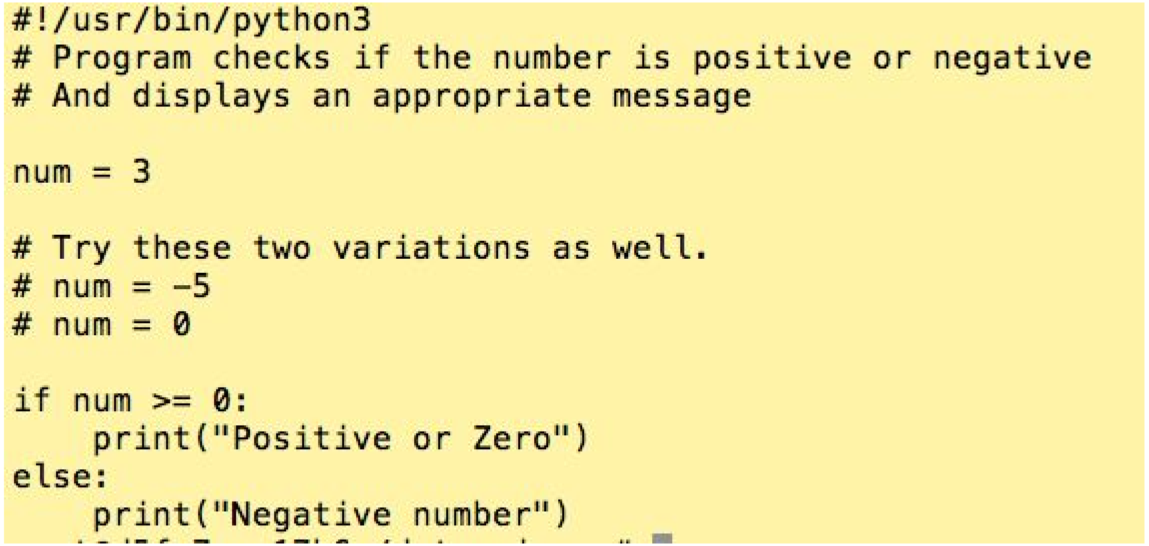
Making Decisions

```
#!/usr/bin/python3
# If the number is positive, we print an appropriate message

num = 3
if num > 0:
    print(num, "is a positive number.")
print("This is always printed.")

num = -1
if num > 0:
    print(num, "is a positive number.")
print("This is also always printed.")
```

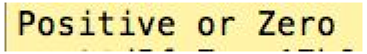
```
3 is a positive number.
This is always printed.
This is also always printed.
```

if_demo.py

Making Decisions (Cont...)



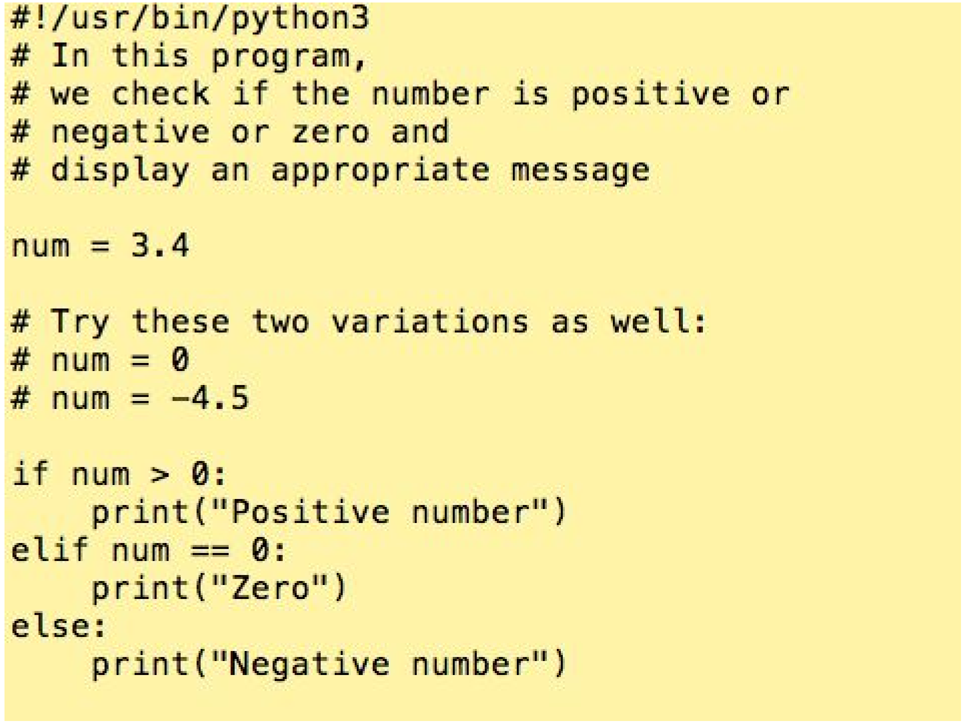
```
#!/usr/bin/python3
# Program checks if the number is positive or negative
# And displays an appropriate message

num = 3

# Try these two variations as well.
# num = -5
# num = 0

if num >= 0:
    print("Positive or Zero")
else:
    print("Negative number")
```



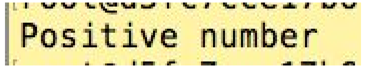
```
Positive or Zero
```

if_else_demo.py

Making Decisions (Cont...)



```python
#!/usr/bin/python3
# In this program,
# we check if the number is positive or
# negative or zero and
# display an appropriate message

num = 3.4

# Try these two variations as well:
# num = 0
# num = -4.5

if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```



```
Positive number
```

if_elif_demo.py

Topics

- Hello Python
- Comments
- Mathematics
- Variables
- Data Types
- Making Decisions
- Strings
- Python Lists
- For, While - Loops
- Lists Revisits
- Handling Errors
- Functions
- Modules
- Packages
- Dictionaries
- Classes
- Iterators
- String Formatting
- Regular Expressions

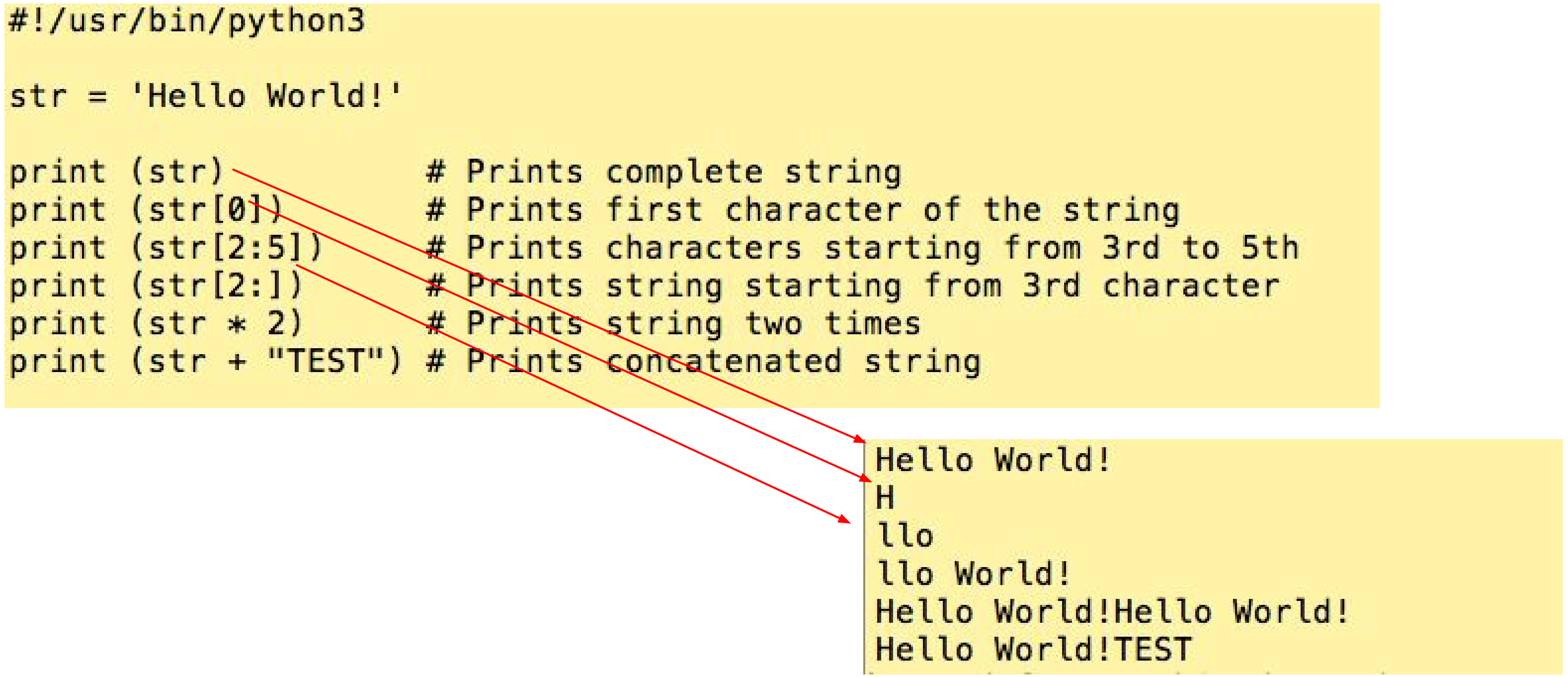
Strings

- Strings in Python are identified as a contiguous set of characters represented in the quotation marks.
- Python allows for either pairs of single or double quotes.
- Subsets of strings can be taken using the slice operator ([ ] and [:])
    - with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

OPERATIONS

- Concatenation +
- Repetition with the help of asterisk *

Strings (Cont...)

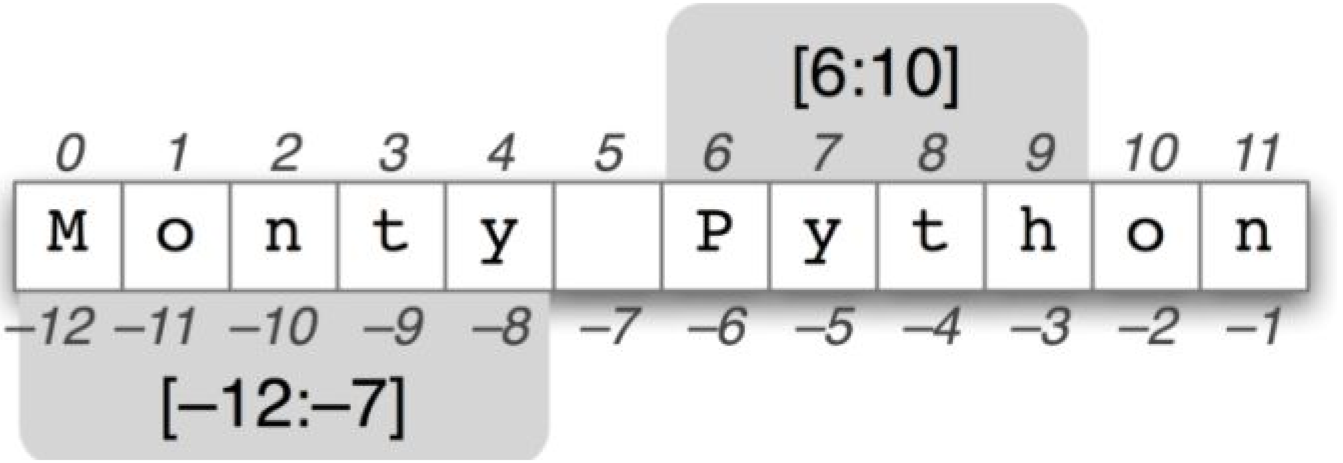


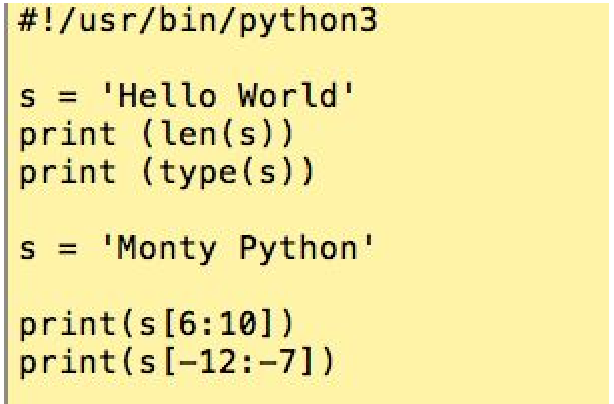
```
#!/usr/bin/python3

str = 'Hello World!'

print (str)          # Prints complete string
print (str[0])       # Prints first character of the string
print (str[2:5])     # Prints characters starting from 3rd to 5th
print (str[2:])      # Prints string starting from 3rd character
print (str * 2)      # Prints string two times
print (str + "TEST") # Prints concatenated string
```

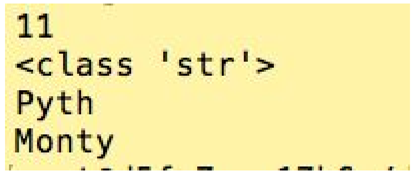
```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

strings_demo.py

Strings (Cont...)




```
#!/usr/bin/python3

s = 'Hello World'
print (len(s))
print (type(s))

s = 'Monty Python'

print(s[6:10])
print(s[-12:-7])
```


```
11
<class 'str'>
Pyth
Monty
```

strings_len.py

Strings - Quiz

- ❖ What is the output when following statements is executed ?
  - ➢ >>>"a"+"bc"
  - ➢ >>>"abcd"[2:]
- ❖ What arithmetic operators cannot be used with strings ?
  - ➢ +, -, *, **
- ❖ What is the output when following code is executed ?
  - ➢ >>>print r"\nhello"
  - ➢ print '\x97\x98'
- ❖ What is the output when following code is executed ?
  - ➢ >>>str1="helloworld"
  - ➢ >>>str1[::-1]
- ❖ What is the output of the following?
  - ➢ print("xyyzxyzxzxyy".count('yy'))
  - ➢ print("xyyzxyzxzxyy".count('yy', 1))
  - ➢ print("xyyzxyzxzxyy".count('xyy', 0, 100))

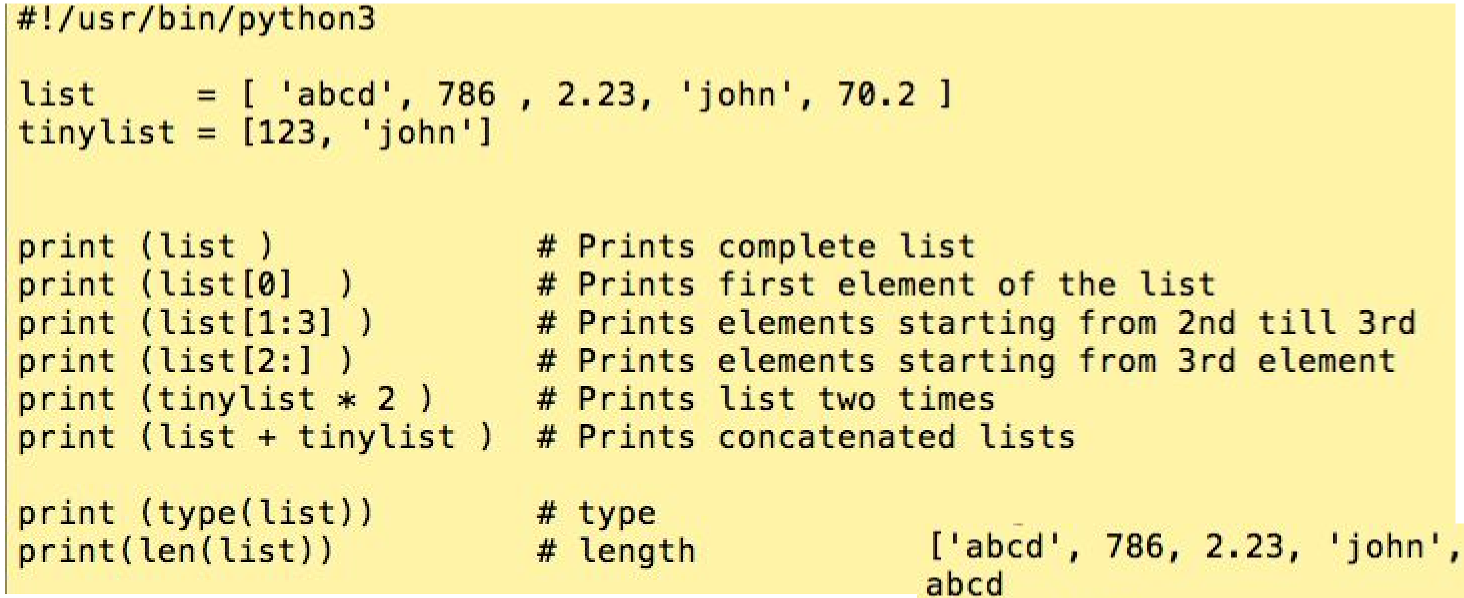
Topics

- Hello Python
- Comments
- Mathematics
- Variables
- Data Types
- Making Decisions
- Strings
- Python Lists
- For, While - Loops
- Lists Revisits
- Handling Errors
- Functions
- Modules
- Packages
- Dictionaries
- Classes
- Iterators
- String Formatting
- Regular Expressions

Python Lists

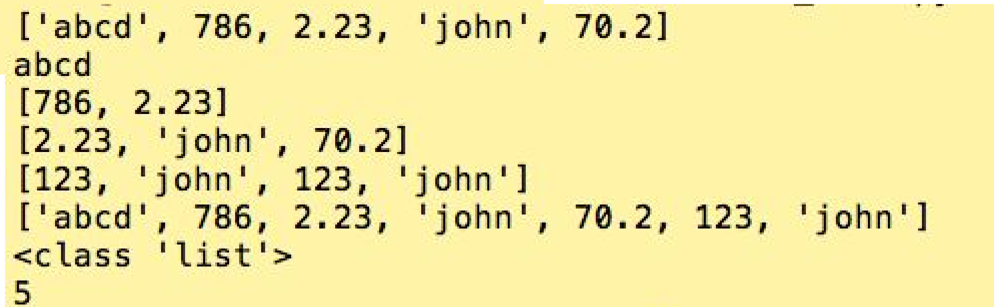
A list contains items separated by commas

and enclosed within square brackets ([])



```
#!/usr/bin/python3

list     = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']


print (list )           # Prints complete list
print (list[0]   )      # Prints first element of the list
print (list[1:3] )      # Prints elements starting from 2nd till 3rd
print (list[2:] )       # Prints elements starting from 3rd element
print (tinylist * 2 )   # Prints list two times
print (list + tinylist )  # Prints concatenated lists

print (type(list))      # type
print(len(list))        # length
```

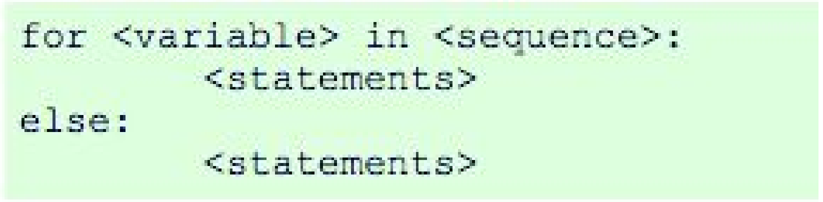
List is sequence



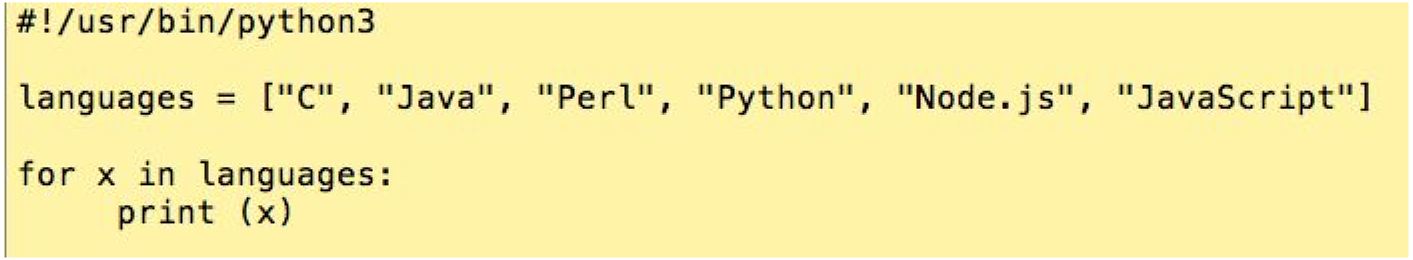
```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
<class 'list'>
5
```

lists_demo.py

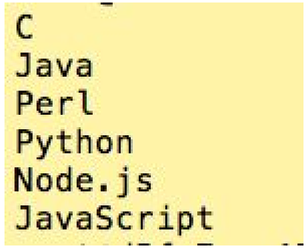
for - Loop



```
for <variable> in <sequence>:
        <statements>
else:
        <statements>
```



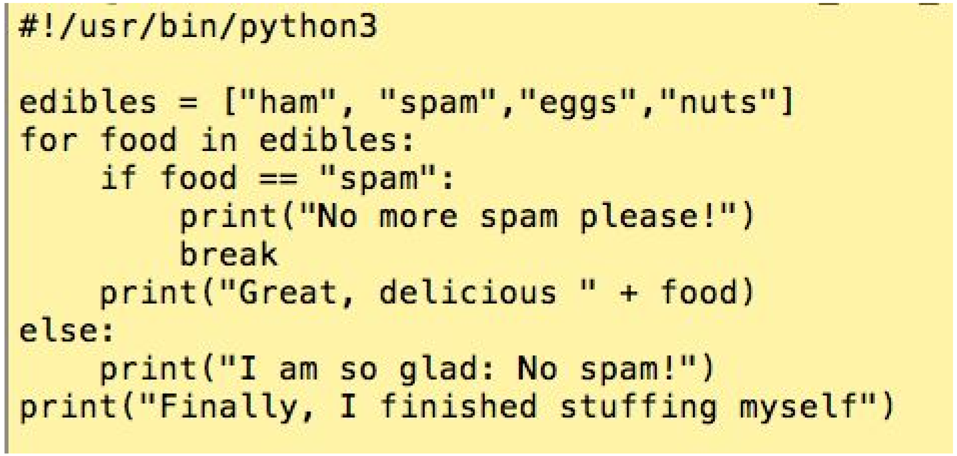
```
#!/usr/bin/python3

languages = ["C", "Java", "Perl", "Python", "Node.js", "JavaScript"]

for x in languages:
     print (x)
```



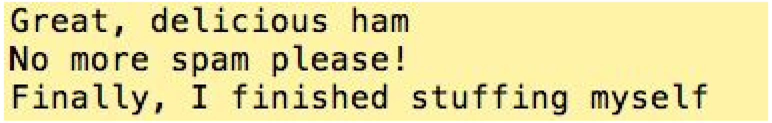
```
C
Java
Perl
Python
Node.js
JavaScript
```

for_demo.py

for - loop (Cont...)



```
#!/usr/bin/python3

edibles = ["ham", "spam","eggs","nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        break
    print("Great, delicious " + food)
else:
    print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

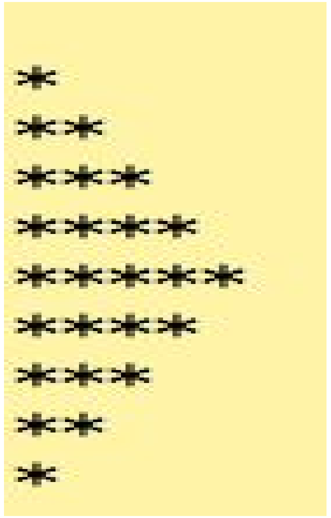


```
Great, delicious ham
No more spam please!
Finally, I finished stuffing myself
```

for_demo_adv.py

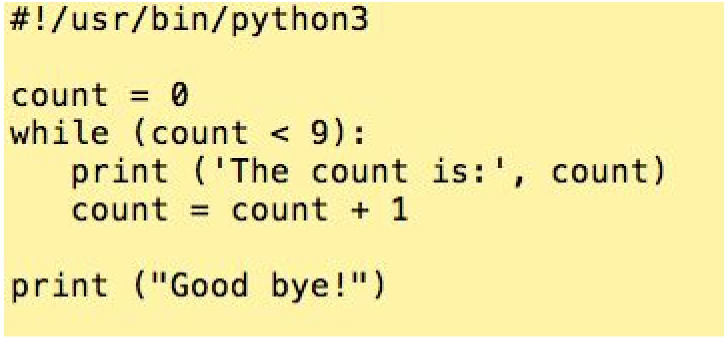
Assignment - 4

Write a script to print pattern like below.

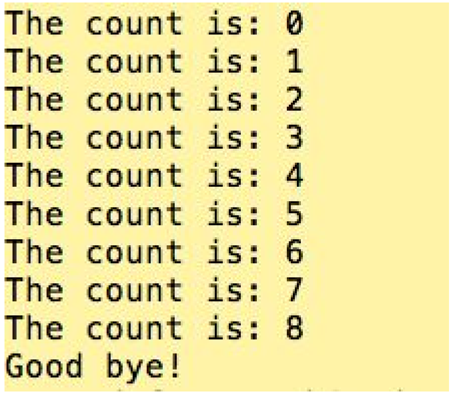


```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

while - loop



```
#!/usr/bin/python3

count = 0
while (count < 9):
   print ('The count is:', count)
   count = count + 1

print ("Good bye!")
```



```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

while_demo.py

Assignment - 5

Write a program - to count “even” and “odd” numbers which are less than 50.

Revisit Lists

Accessing Lists

- n = len(L)
- item = L[index]
- seq = L[start:stop]
- seq = L[start:stop:step]
- seq = L[::2]   # get every other item, starting with the first
- seq = L[1::2]   # get every other item, starting with the second

Revisit Lists (Cont...)

```
for item in L:
    print (item)
```

```
for index, item in enumerate(L):
        print index, item
```

Revisit Lists (Cont...)

Other important Operations

- L.append(item)
- L.extend(sequence)
- L.insert(index, item)
- del L[i]
- del L[i:j]
- item = L.pop() # last item
- item = L.pop(0) # first item
- item = L.pop(index)
- L.remove(item)
- L.reverse()
- L.sort()
- out = sorted(L)

Quiz

- Q.1. Select the right options

```
names = ['Amir', 'Sahrukh', 'Chales', 'Dao']
print names[-1][-1]
```

So what's the output?

- i. A
- ii. r
- iii. Amir
- iv. Dao

- Q. 2. What gets printed?

```
names1 = ['Amir', 'Sahrukh', 'Chales', 'Dao']
names2 = names1
names3 = names1[:]

names2[0] = 'Alice'
names3[1] = 'Bob'

sum = 0
for ls in (names1, names2, names3):
    if ls[0] == 'Alice':
        sum += 1
    if ls[1] == 'Bob':
        sum += 10

print (sum)
```

Quiz (Cont...)

- Q. 3. What gets printed?

```
names1 = ['Amir', 'Sahrukh', 'Chales', 'Dao']
loc = names1.index("Edward")

print (loc)
```

- Q. 4. What's printed in following code?

```
names1 = ['Amir', 'Sahrukh', 'Chales', 'Dao']

if 'amir' in names1:
    print (1)
else:
    print (2)
```

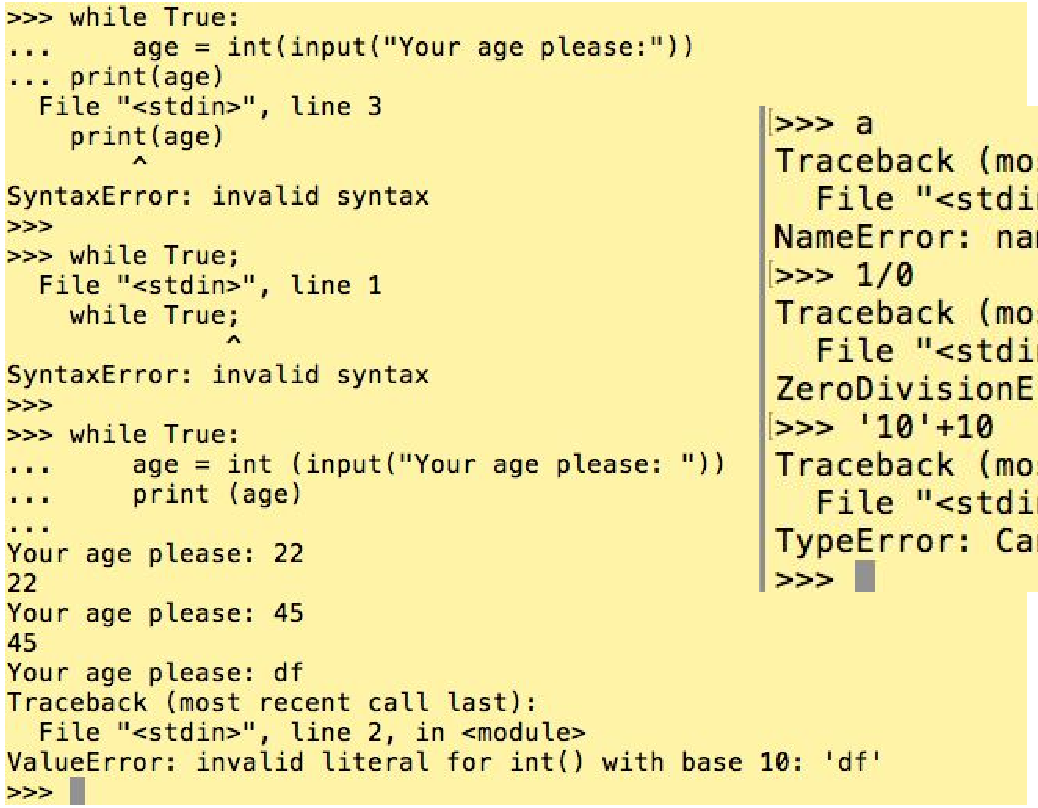
- Q. 5. What gets printed?

```
numbers = [1, 2, 3, 4]
numbers.append([5,6,7,8])
print (len(numbers))
```

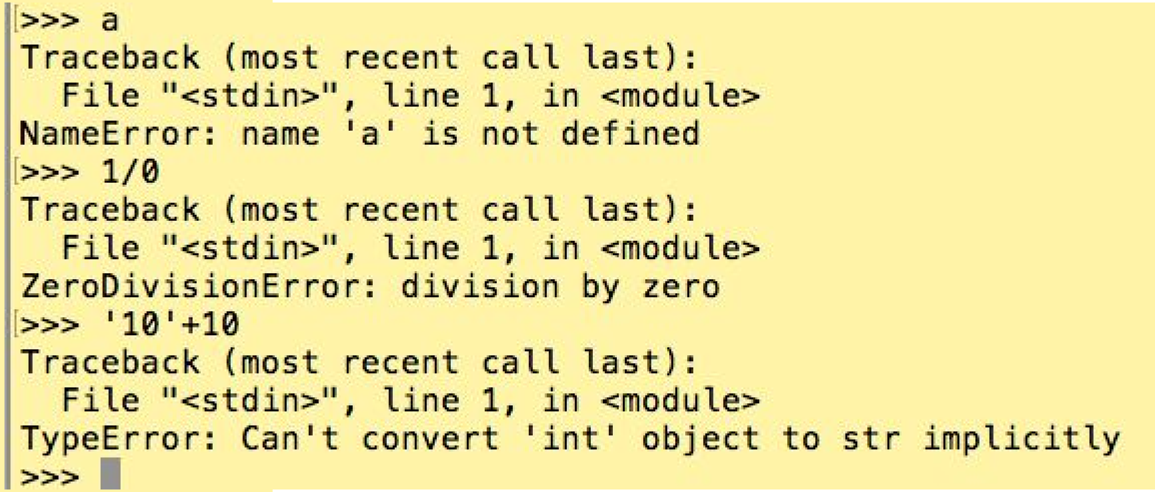
Topics

- Hello Python
- Comments
- Mathematics
- Variables
- Data Types
- Making Decisions
- Strings
- Python Lists
- For, While - Loops
- Lists Revisits
- Handling Errors
- Functions
- Modules
- Packages
- Dictionaries
- Classes
- Iterators
- String Formatting
- Regular Expressions

Handling Errors



```
>>> while True:
...     age = int(input("Your age please:"))
... print(age)
  File "<stdin>", line 3
    print(age)
        ^
SyntaxError: invalid syntax
>>>
>>> while True;
  File "<stdin>", line 1
    while True;
              ^
SyntaxError: invalid syntax
>>>
>>> while True:
...     age = int (input("Your age please: "))
...     print (age)
...
Your age please: 22
22
Your age please: 45
45
Your age please: df
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'df'
>>>
```



```
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> '10'+10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>>
```

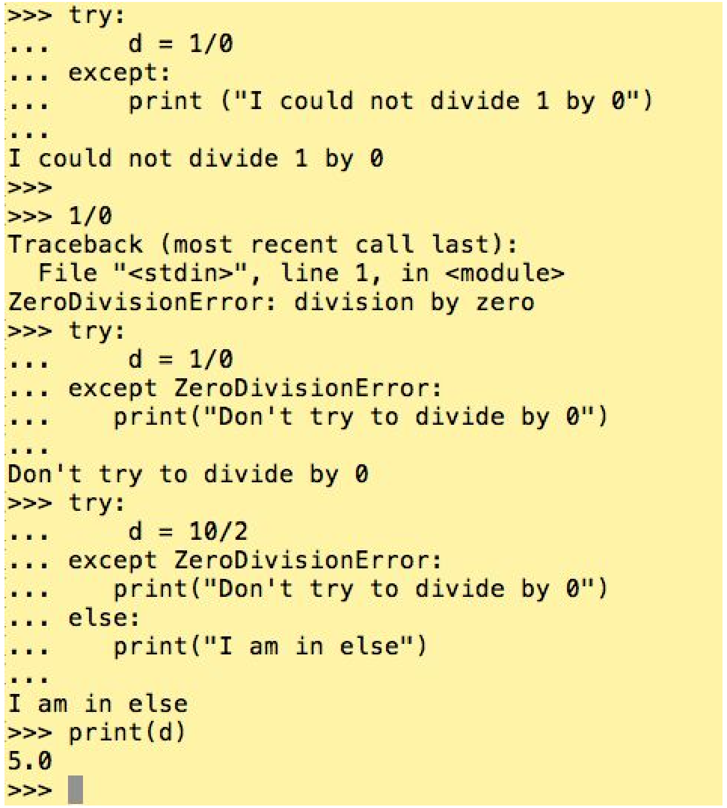
Assignment - 6

Q 1 Develop a script to create list of lists. The element list should be factors of index of list.

(Note - We are yet to talk about “functions”)

Q 2. Develop a script to find the Highest Common Factor (HCF) of given two numbers.

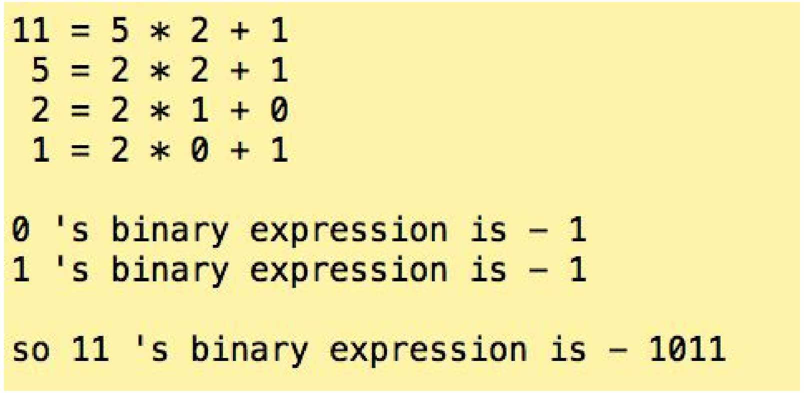
Handling Errors(Cont..)



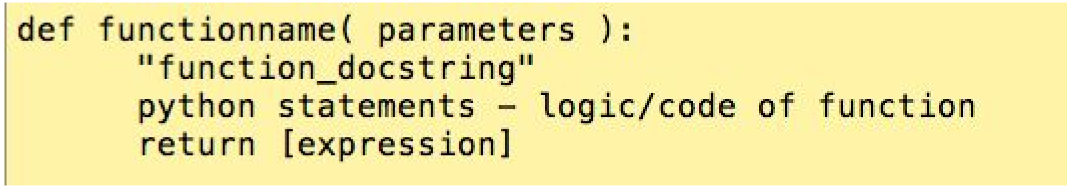
```
>>> try:
...     d = 1/0
... except:
...     print ("I could not divide 1 by 0")
...
I could not divide 1 by 0
>>>
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> try:
...     d = 1/0
... except ZeroDivisionError:
...    print("Don't try to divide by 0")
...
Don't try to divide by 0
>>> try:
...     d = 10/2
... except ZeroDivisionError:
...    print("Don't try to divide by 0")
... else:
...    print("I am in else")
...
I am in else
>>> print(d)
5.0
>>> 
```

Functions

Let us look at the problem of converting decimal number into binary number



```
11 = 5 * 2 + 1
 5 = 2 * 2 + 1
 2 = 2 * 1 + 0
 1 = 2 * 0 + 1

0 's binary expression is - 1
1 's binary expression is - 1

so 11 's binary expression is - 1011
```



```
def functionname( parameters ):
      "function_docstring"
      python statements - logic/code of function
      return [expression]
```

Functions

```
#!/usr/bin/python3

name = "Aegis"

def hellofunction(name=None):
    '''hello function'''
    if name:
       print ("Hello " + name )
    else:
      print ("Hello World!")

#hellofunction(name)
hellofunction()
```

```
#!/usr/bin/python3
def decToBin(n):
    if n==0:
         return '0'
    else:
        return decToBin(int(n/2)) + str(n%2)

d = decToBin(11)

print (d)
```

function_dec2binary.py

function_hello.py

Assignment - 7

Q 1 Develop a script to create list of lists. The element list should be factors of index of list.

(Note - We have talked about “functions”)

Q 2. Develop a script to find the Highest Common Factor (HCF) of given two numbers.

Topics

- Hello Python
- Comments
- Mathematics
- Variables
- Data Types
- Making Decisions
- Strings
- Python Lists
- For, While - Loops
- Lists Revisits
- Handling Errors
- Functions
- Modules
- Packages
- Dictionaries
- Classes
- Iterators
- String Formatting
- Regular Expressions

Modules

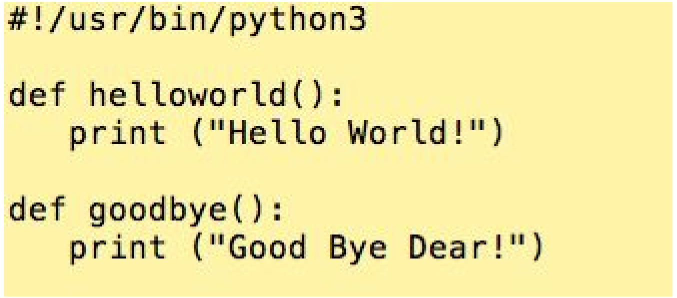
- Modules are Python files with the .py extension.
- These files implement a set of functions and can have python statements.
- The modules can be imported from other modules using the import command.

The modules are best way to share your work, your tools with others.

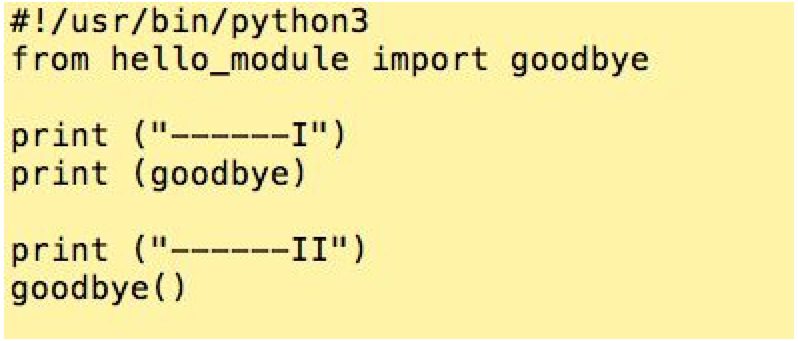
Have a look at:

- https://docs.python.org/3/library/
- https://docs.python.org/2/library/

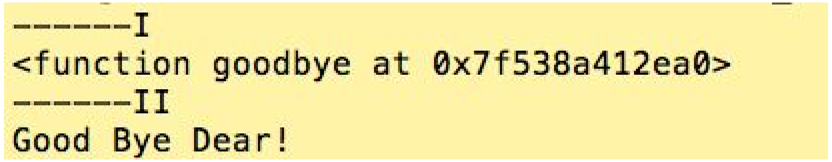
Modules (cont...)



```
#!/usr/bin/python3

def helloworld():
   print ("Hello World!")

def goodbye():
   print ("Good Bye Dear!")
```



```
#!/usr/bin/python3
from hello_module import goodbye

print ("------I")
print (goodbye)

print ("------II")
goodbye()
```

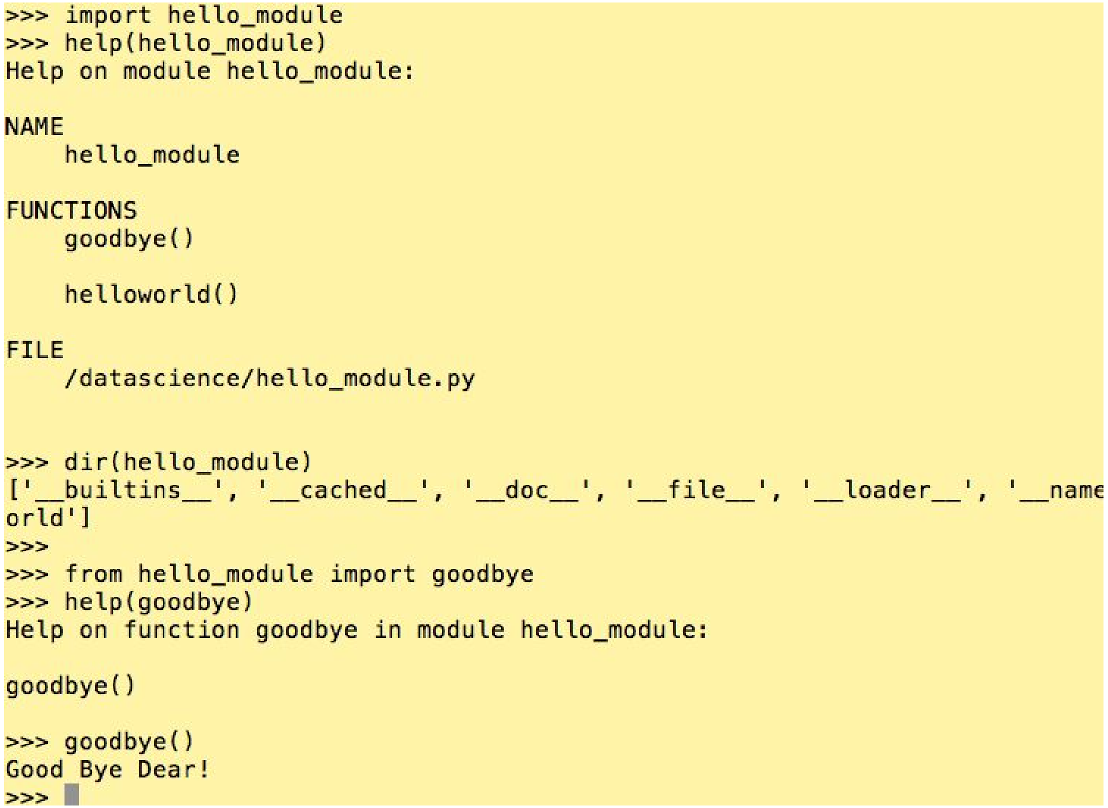


```
------I
<function goodbye at 0x7f538a412ea0>
------II
Good Bye Dear!
```

hello_module.py

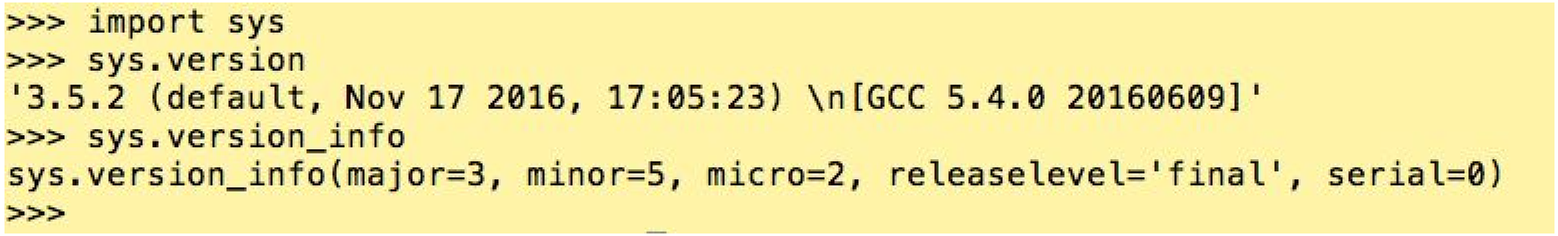
test_hello_module.py

Modules (Cont...)

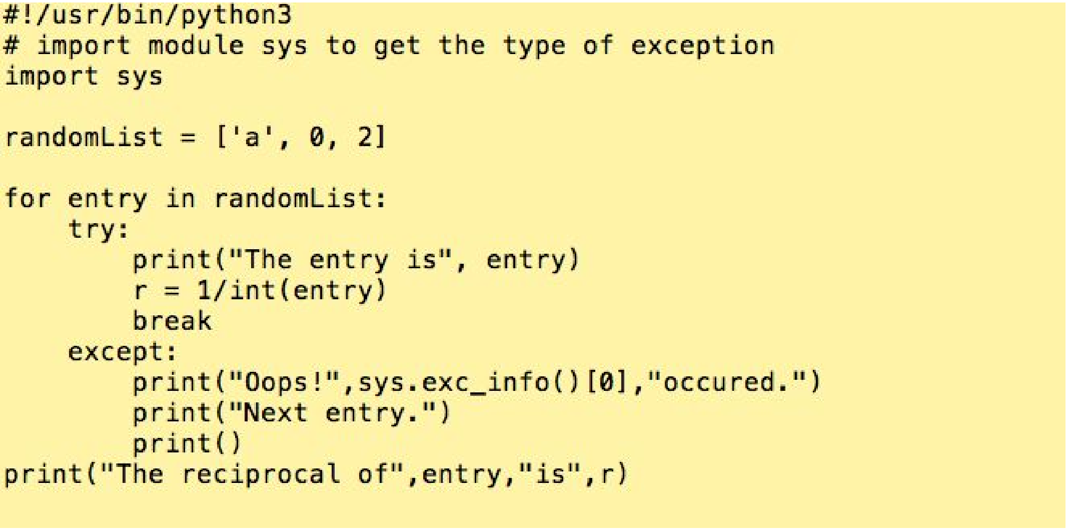


```
>>> import hello_module
>>> help(hello_module)
Help on module hello_module:

NAME
    hello_module

FUNCTIONS
    goodbye()

    helloworld()

FILE
    /datascience/hello_module.py


>>> dir(hello_module)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name
orld']
>>>
>>> from hello_module import goodbye
>>> help(goodbye)
Help on function goodbye in module hello_module:

goodbye()

>>> goodbye()
Good Bye Dear!
>>> 
```

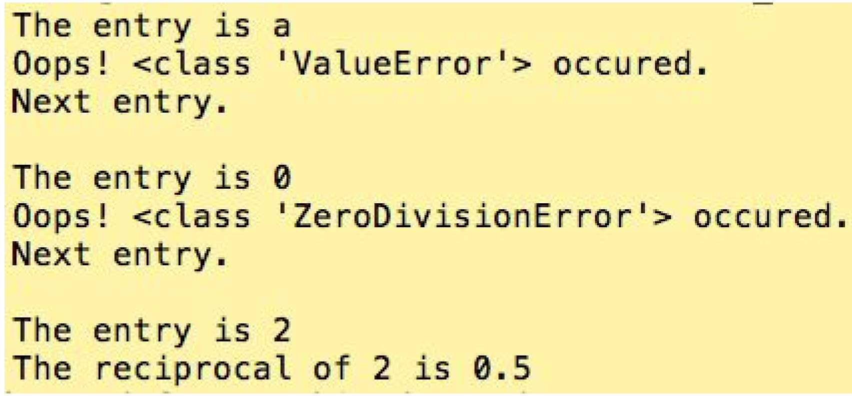
Modules (Cont …)



```
>>> import sys
>>> sys.version
'3.5.2 (default, Nov 17 2016, 17:05:23) \n[GCC 5.4.0 20160609]'
>>> sys.version_info
sys.version_info(major=3, minor=5, micro=2, releaselevel='final', serial=0)
>>>
```



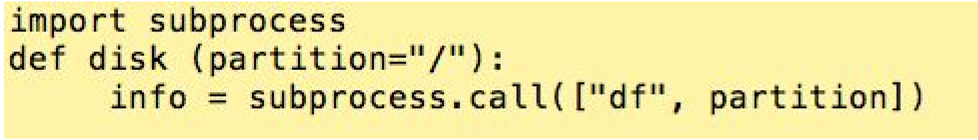
```
#!/usr/bin/python3
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!",sys.exc_info()[0],"occured.")
        print("Next entry.")
        print()
print("The reciprocal of",entry,"is",r)
```



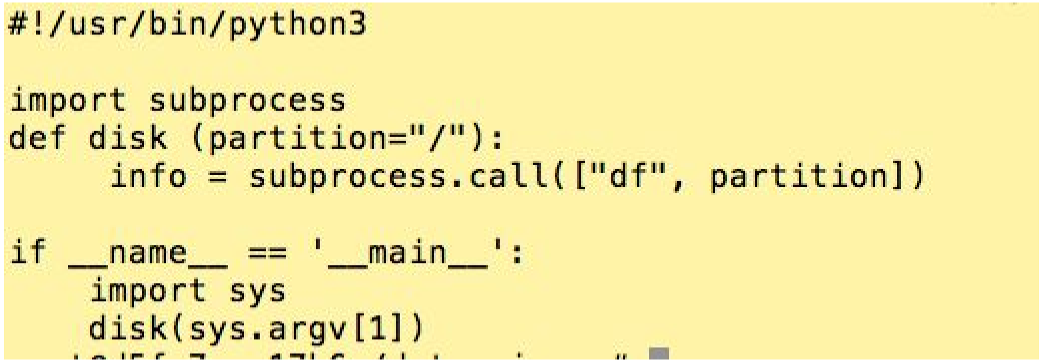
```
The entry is a
Oops! <class 'ValueError'> occured.
Next entry.

The entry is 0
Oops! <class 'ZeroDivisionError'> occured.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5
```

one_more_exception.py

Module (Cont ...)



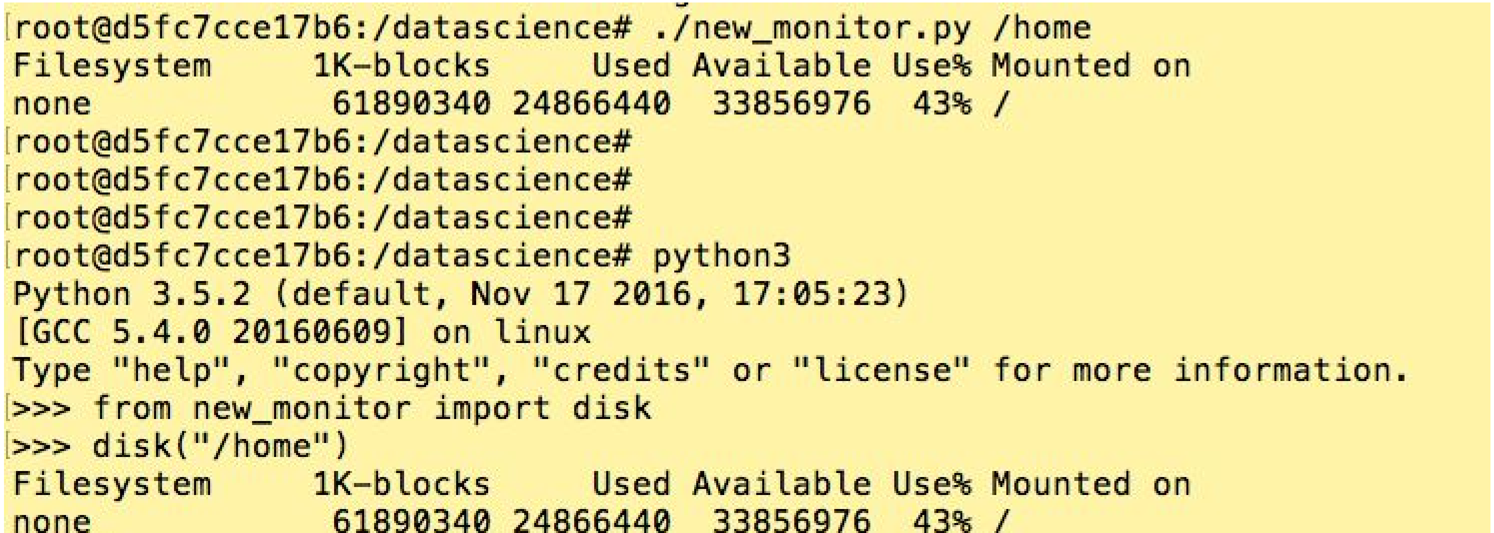
```
import subprocess
def disk (partition="/"):
     info = subprocess.call(["df", partition])
```



```
#!/usr/bin/python3

import subprocess
def disk (partition="/"):
     info = subprocess.call(["df", partition])

if __name__ == '__main__':
    import sys
    disk(sys.argv[1])
```

monitor.py

new_monitor.py



```
root@d5fc7cce17b6:/datascience# ./new_monitor.py /home
Filesystem     1K-blocks     Used Available Use% Mounted on
none            61890340 24866440  33856976  43% /
root@d5fc7cce17b6:/datascience#
root@d5fc7cce17b6:/datascience#
root@d5fc7cce17b6:/datascience#
root@d5fc7cce17b6:/datascience# python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from new_monitor import disk
>>> disk("/home")
Filesystem     1K-blocks     Used Available Use% Mounted on
none            61890340 24866440  33856976  43% /
```

Packages

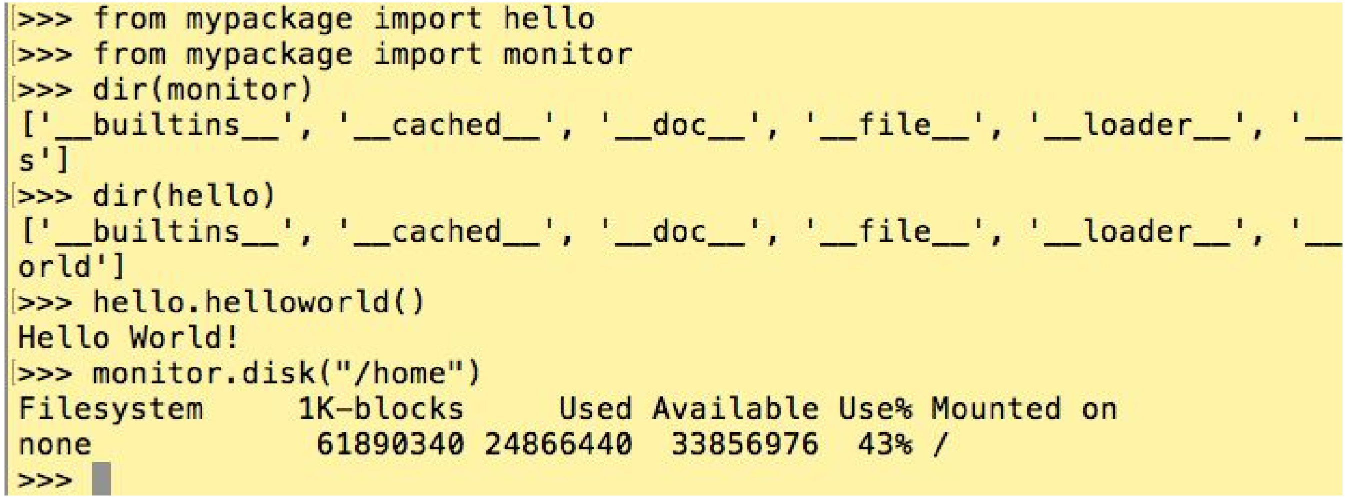
- Help structure Python’s module namespace by using “dotted module names”.
  - The module name *package_a.mod_b* designates a submodule named *mod_b* in a package named *package_a.*
- Use of dotted module names saves the developers of multi-module packages from having to worry about each other's global variable names.

Let us assume we have the following directory structure. Here, hello.py & monotor.py are same modules described in *Module* section, and init.py is an empty file:

Packages (Cont …)

```
mypackage
|-- __init__.py
|-- hello_module.py
`-- monitor.py

0 directories, 3 files
```

init.py helps Python to treats this directory (/*mypackage*) as package directory.



```
>>> from mypackage import hello
>>> from mypackage import monitor
>>> dir(monitor)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__
s']
>>> dir(hello)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__
orld']
>>> hello.helloworld()
Hello World!
>>> monitor.disk("/home")
Filesystem     1K-blocks     Used Available Use% Mounted on
none            61890340 24866440  33856976  43% /
>>> 
```

Assignment - 8

Q1. Create Package - which has modules to

- Factors of Number
- Calculate HCF
- Calculate the Factorial

Dictionaries

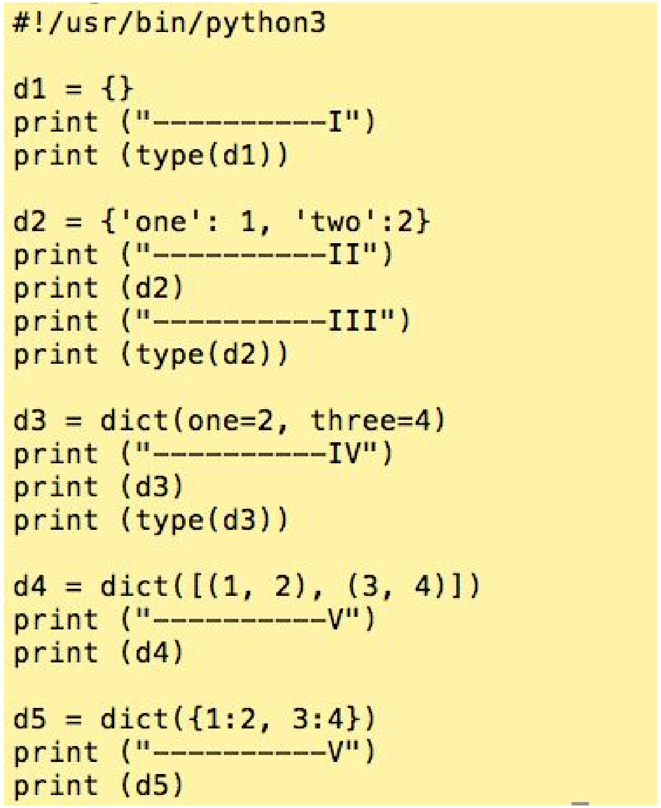
Also known as:

- *Associate Array*
- *Map*
- *Hash Map*
- *Unordered Map*

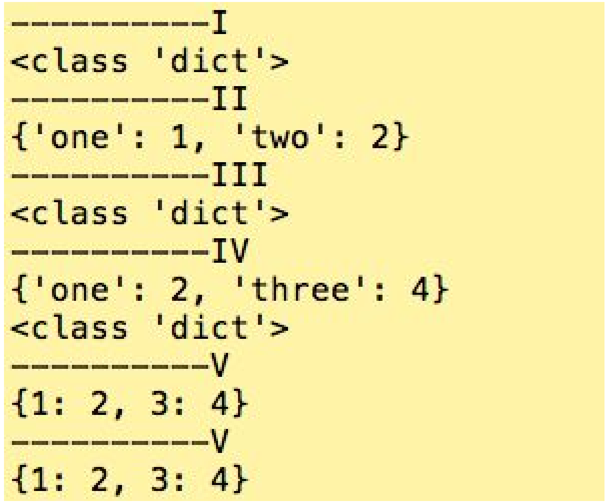
Topics

- Hello Python
- Comments
- Mathematics
- Variables
- Data Types
- Making Decisions
- Strings
- Python Lists
- For, While - Loops
- Lists Revisits
- Handling Errors
- Functions
- Modules
- Packages
- Dictionaries
- Classes
- Iterators
- String Formatting
- Regular Expressions

Dictionaries (Cont …)



```
#!/usr/bin/python3

d1 = {}
print ("----------I")
print (type(d1))

d2 = {'one': 1, 'two':2}
print ("----------II")
print (d2)
print ("----------III")
print (type(d2))

d3 = dict(one=2, three=4)
print ("----------IV")
print (d3)
print (type(d3))

d4 = dict([(1, 2), (3, 4)])
print ("----------V")
print (d4)

d5 = dict({1:2, 3:4})
print ("----------V")
print (d5)
```



```
----------I
<class 'dict'>
----------II
{'one': 1, 'two': 2}
----------III
<class 'dict'>
----------IV
{'one': 2, 'three': 4}
<class 'dict'>
----------V
{1: 2, 3: 4}
----------V
{1: 2, 3: 4}
```

dictionaries.py

Dictionaries (Cont …)

- d.clear()
- d.copy()
- del k[d]
- dict.fromkeys(seq[, value])
- iteration/accessing elements of dictionaries
- for key in my_dictionary:
- for key, value in my_dictionary.items():
- for value in my_dictionary.values():
- iter(d)
- len(d)
- d.keys()
- d.values()
- d.items()

Assignment - 9

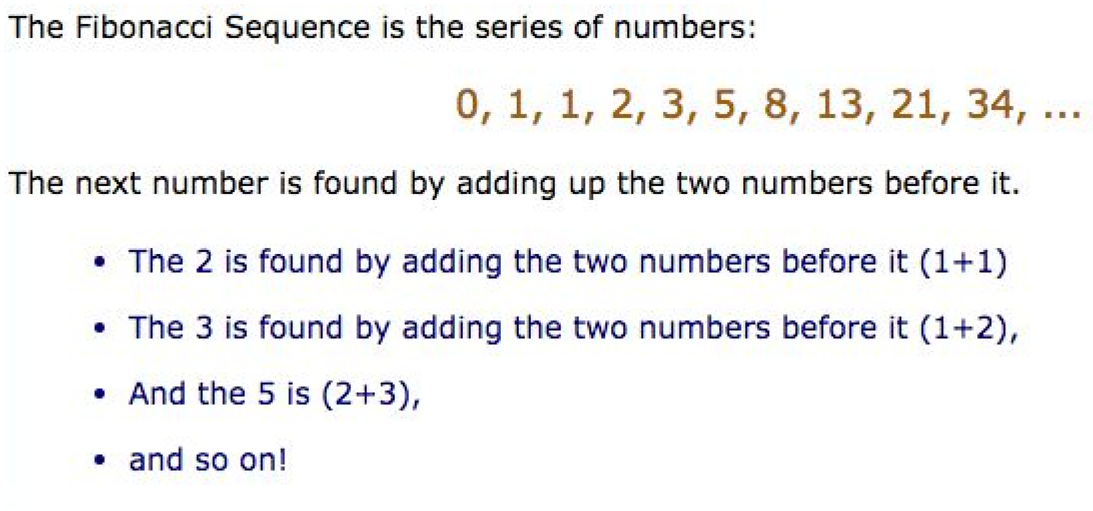
- Create a List - "A", elements are integers. List may have repeated elements
- Create new "B", which has same element from "A" but all elements of "B" are Unique Elements
- Develop a function - to take generic list and return sorted, unique element list (we are still talking about integers)

Caching or Memoization

```
def some_function (arg1, arg2, arg3,..., argN):
        result = some computation involving arg1, arg2, arg3, ... argN
        return result
```

```
catche = {}
def some_function_with_catching(arg1, arg2, arg3, ..., argN):
        key = str(arg1) + str(arg2) + str(arg3) + .... + str(argN)
        if key in cache:
              return catche[key]
        else:
             result = same computation involving arg1, arg2, arg3, ..., argN
             catche[key]=result
             return result
```

Fibonacci Numbers



The Fibonacci Sequence is the series of numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

The next number is found by adding up the two numbers before it.

- The 2 is found by adding the two numbers before it (1+1)
- The 3 is found by adding the two numbers before it (1+2),
- And the 5 is (2+3),
- and so on!

Caching or Memoization - Fibonacci Numbers

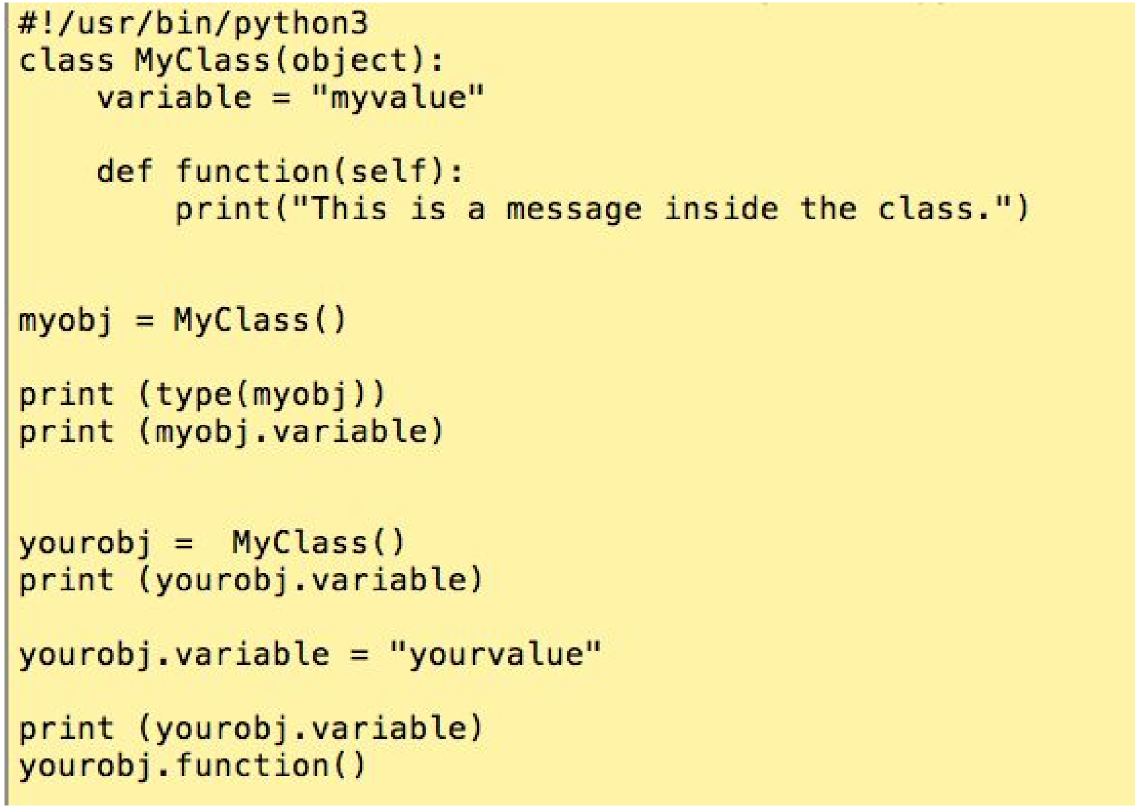
```
def fib(n):
    return n if n < 2 else fib(n-2) + fib(n-1)
```

```
__fib_cache = {}
def fib(n):
    if n in __fib_cache:
        return __fib_cache[n]
    else:
        __fib_cache[n] = n if n < 2 else fib(n-2) + fib(n-1)
        return __fib_cache[n]
```

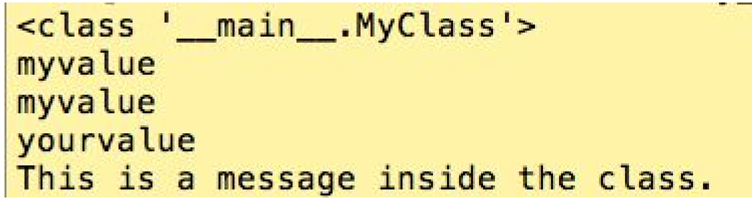
Topics

- Hello Python
- Comments
- Mathematics
- Variables
- Data Types
- Making Decisions
- Strings
- Python Lists
- For, While - Loops
- Lists Revisits
- Handling Errors
- Functions
- Modules
- Packages
- Dictionaries
- Classes
- Iterators
- String Formatting
- Regular Expressions

Class



```
#!/usr/bin/python3
class MyClass(object):
    variable = "myvalue"

    def function(self):
        print("This is a message inside the class.")


myobj = MyClass()

print (type(myobj))
print (myobj.variable)


yourobj =  MyClass()
print (yourobj.variable)

yourobj.variable = "yourvalue"

print (yourobj.variable)
yourobj.function()
```

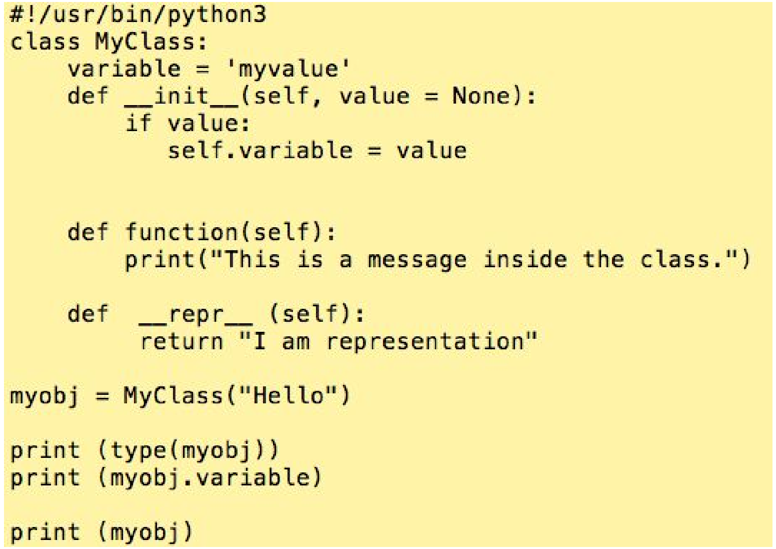


```
<class '__main__.MyClass'>
myvalue
myvalue
yourvalue
This is a message inside the class.
```

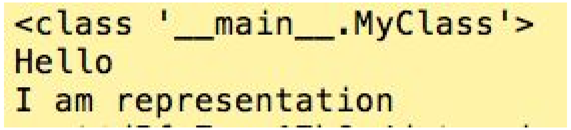
my_class.py

Class (Cont …)

- Classes are essentially a template to create your objects.
- init is the constructor for a class.
  - The self parameter refers to the instance of the object



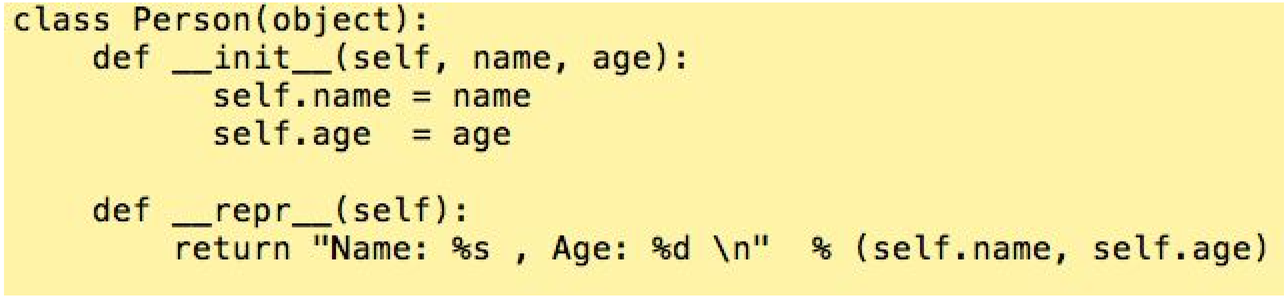
```
#!/usr/bin/python3
class MyClass:
    variable = 'myvalue'
    def __init__(self, value = None):
        if value:
           self.variable = value


    def function(self):
        print("This is a message inside the class.")

    def  __repr__ (self):
         return "I am representation"

myobj = MyClass("Hello")

print (type(myobj))
print (myobj.variable)

print (myobj)
```



```
<class '__main__.MyClass'>
Hello
I am representation
```

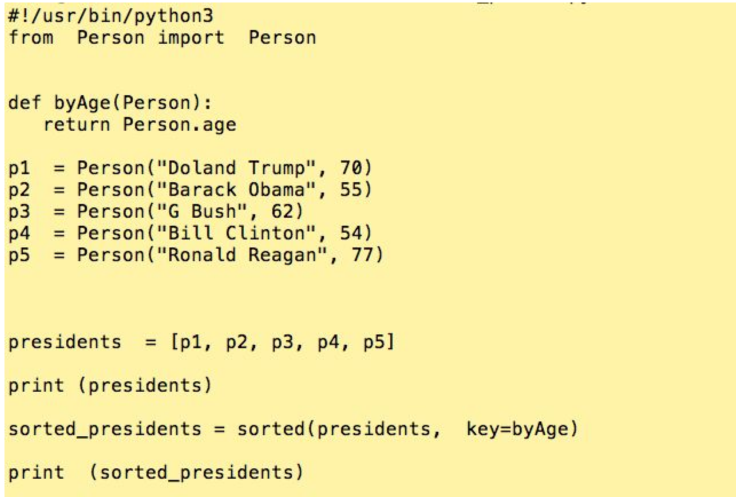
my_advance_class.py

Class (Cont ...)



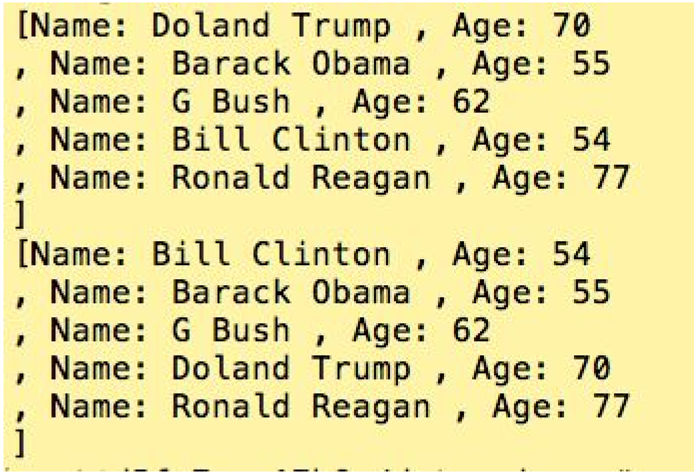
```
class Person(object):
    def __init__(self, name, age):
          self.name = name
          self.age  = age

    def __repr__(self):
        return "Name: %s , Age: %d \n"  % (self.name, self.age)
```

Person.py



```
#!/usr/bin/python3
from  Person import  Person


def byAge(Person):
   return Person.age

p1  = Person("Doland Trump", 70)
p2  = Person("Barack Obama", 55)
p3  = Person("G Bush", 62)
p4  = Person("Bill Clinton", 54)
p5  = Person("Ronald Reagan", 77)



presidents  = [p1, p2, p3, p4, p5]

print (presidents)

sorted_presidents = sorted(presidents,  key=byAge)

print  (sorted_presidents)
```

sort_person.py



```
[Name: Doland Trump , Age: 70
, Name: Barack Obama , Age: 55
, Name: G Bush , Age: 62
, Name: Bill Clinton , Age: 54
, Name: Ronald Reagan , Age: 77
]
[Name: Bill Clinton , Age: 54
, Name: Barack Obama , Age: 55
, Name: G Bush , Age: 62
, Name: Doland Trump , Age: 70
, Name: Ronald Reagan , Age: 77
]
```

Assignment - 10

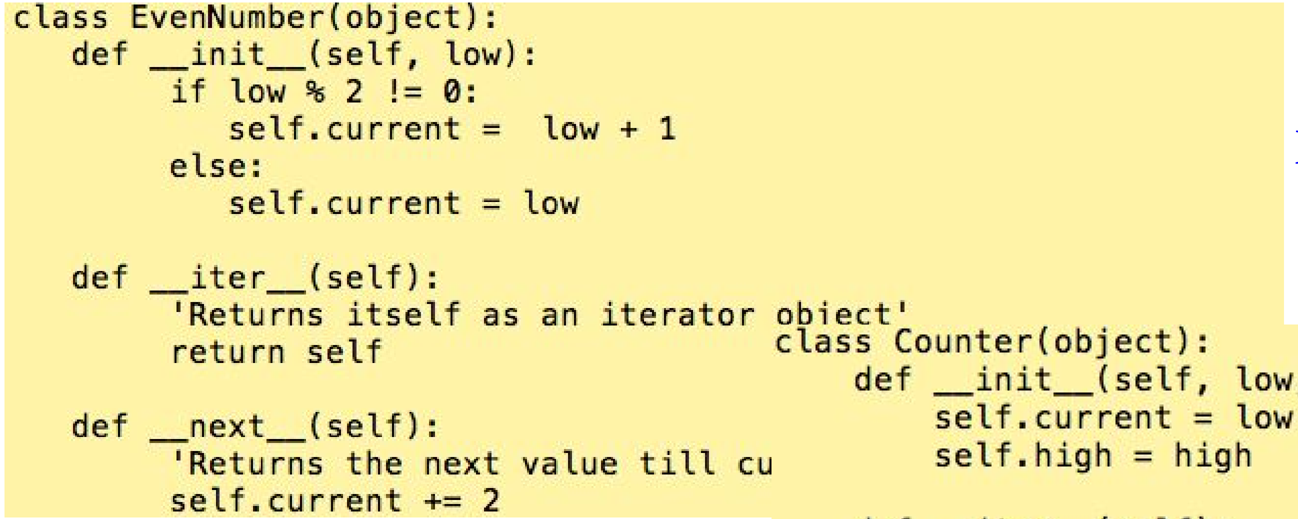
- Create Class - "City"
- Attributes - [1] Population [2] Country
- Sort by Population
- How about adding GDP - and sort by GDP?

Revisit Iterators

Python iterator objects are required to support two methods while following the iterator protocol.

- *iter* returns the iterator object itself. This is used in for and in statements.
- *next* method returns the next value from the iterator. If there is no more items to return then it should raise StopIteration exception.

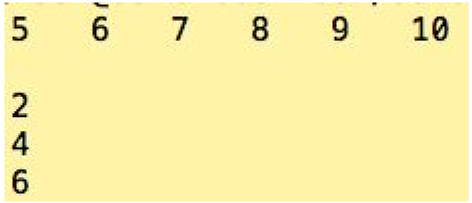
Iterator Revisits (Cont …)



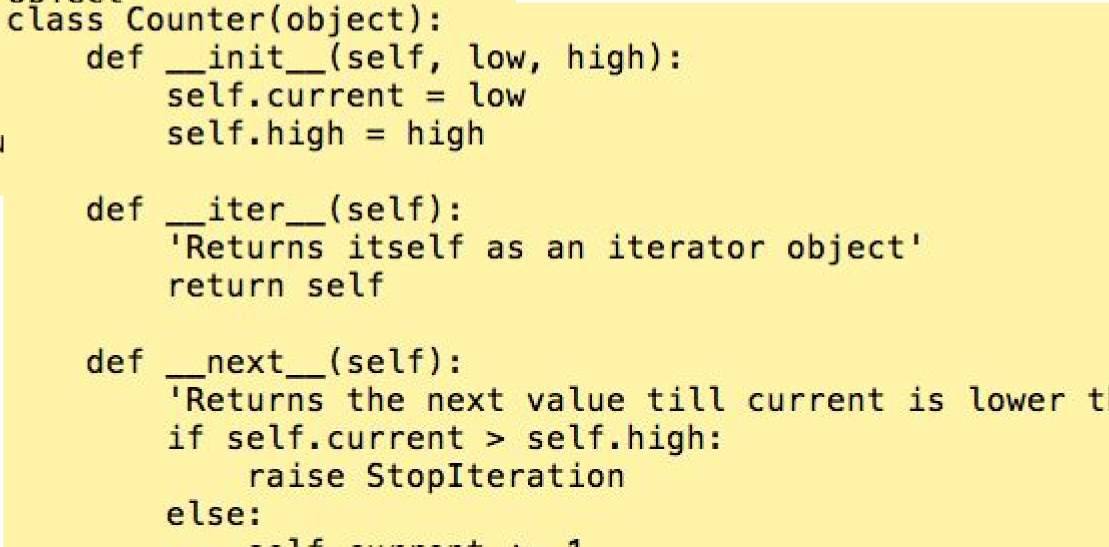
```
class EvenNumber(object):
   def __init__(self, low):
        if low % 2 != 0:
           self.current =  low + 1
        else:
           self.current = low

   def __iter__(self):
        'Returns itself as an iterator object'
        return self

   def __next__(self):
        'Returns the next value till cu
        self.current += 2
```

EvenNumber.py

Counter.py



```
5   6   7   8   9   10

2
4
6
```

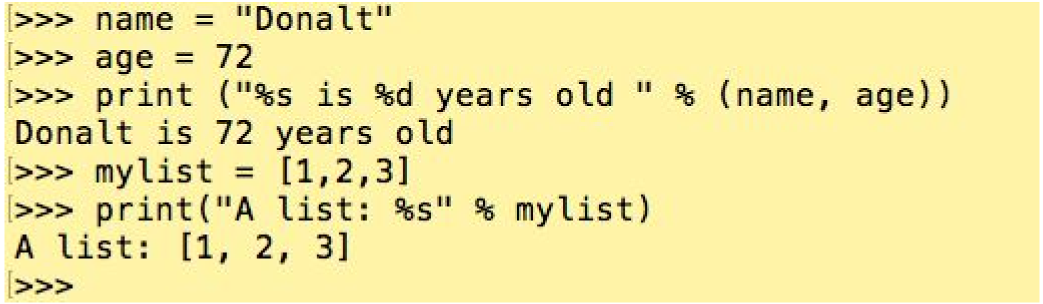


```
class Counter(object):
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        'Returns itself as an iterator object'
        return self

    def __next__(self):
        'Returns the next value till current is lower t
        if self.current > self.high:
            raise StopIteration
        else:
```

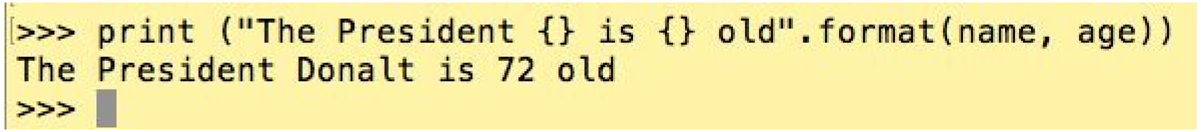
test_counter_evennumber.py

String Formatting

- %s - String (or any object with a string representation, like numbers)
- %d - Integers
- %f - Floating point numbers
- %.f - Floating point numbers with a fixed amount of digits to the right of the dot.
- %x/%X - Integers in hex representation (lowercase/uppercase)
- New style {} & format function on strings



```
>>> name = "Donalt"
>>> age = 72
>>> print ("%s is %d years old " % (name, age))
Donalt is 72 years old
>>> mylist = [1,2,3]
>>> print("A list: %s" % mylist)
A list: [1, 2, 3]
>>>
```



```
>>> print ("The President {} is {} old".format(name, age))
The President Donalt is 72 old
>>>
```

Regular Expressions

Let us define some rules to form some strings:

- Write a letter "a" at least once
- Append to this the letter "b" exactly five times
- Append to this the letter "c" any even number of times
- Optionally, write the letter "d" at the end

Examples of such strings are:

`aaaabbbbbccccd`

`aabbbbbcc`

`...`

Topics

- Hello Python
- Comments
- Mathematics
- Variables
- Data Types
- Making Decisions
- Strings
- Python Lists
- For, While - Loops
- Lists Revisits
- Handling Errors
- Functions
- Modules
- Packages
- Dictionaries
- Classes
- Iterators
- String Formatting
- **Regular Expressions**

Regular Expression (Cont ...)

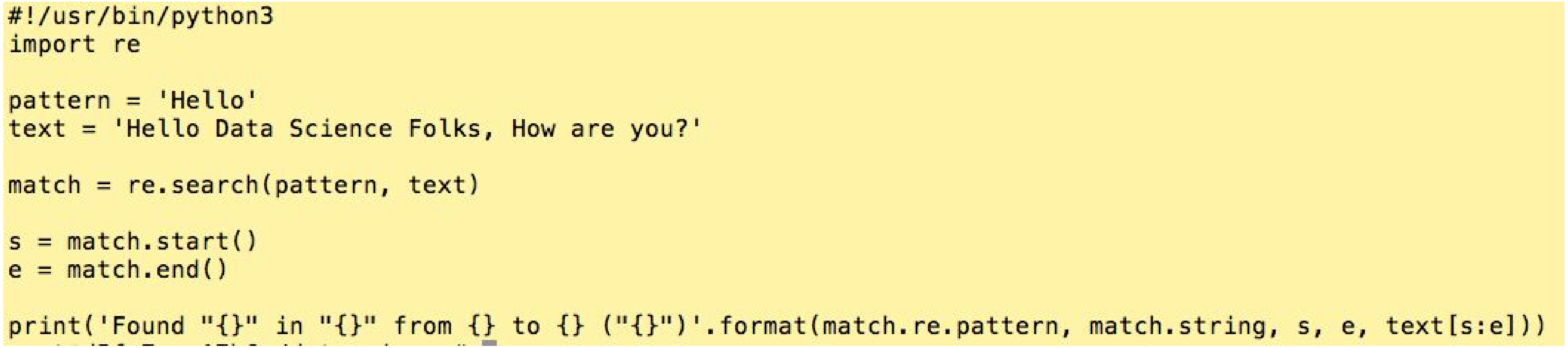
There are infinitely many such strings which satisfy above rules.

Regular Expressions are merely a shorthand way of expressing these sets of rules

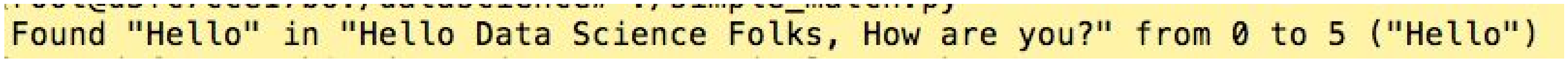
- Regex are text matching patterns described with a formal syntax
- The patterns which are executed on text as input to produce either matching subset or modified version of original text
- Regular Expression is kind of programming language itself
- "re" module provides this functionality in Python Programming

Your friendship with re will always add advantage to your skills if you ever need to deal with Text Processing in your project.

Regular Expressions



```
#!/usr/bin/python3
import re

pattern = 'Hello'
text = 'Hello Data Science Folks, How are you?'

match = re.search(pattern, text)

s = match.start()
e = match.end()

print('Found "{}" in "{}" from {} to {} ("{}")'.format(match.re.pattern, match.string, s, e, text[s:e]))
```

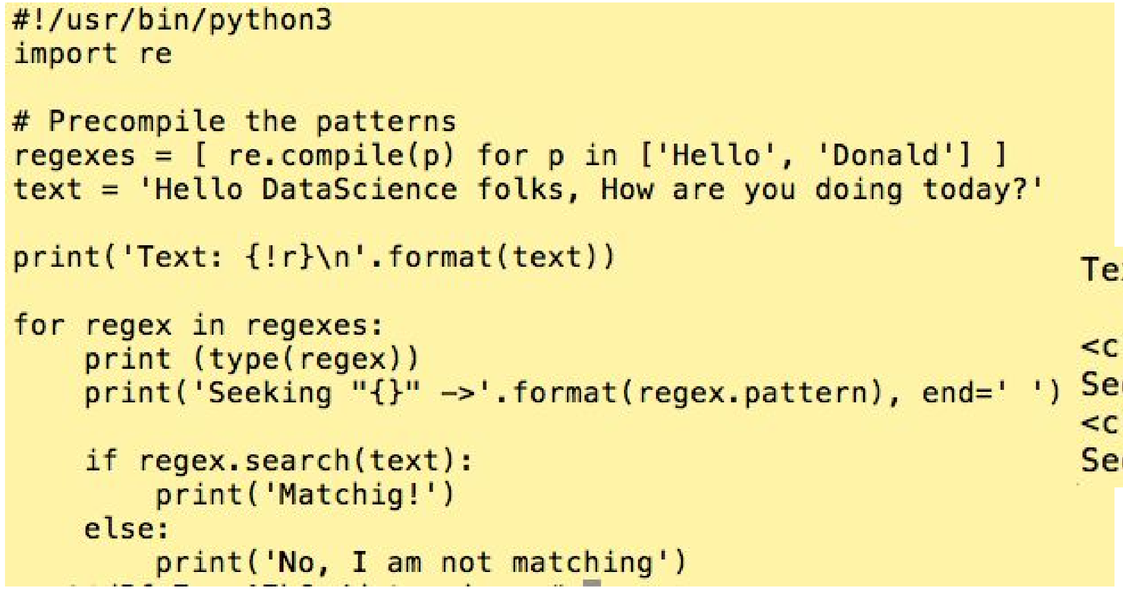


```
Found "Hello" in "Hello Data Science Folks, How are you?" from 0 to 5 ("Hello")
```

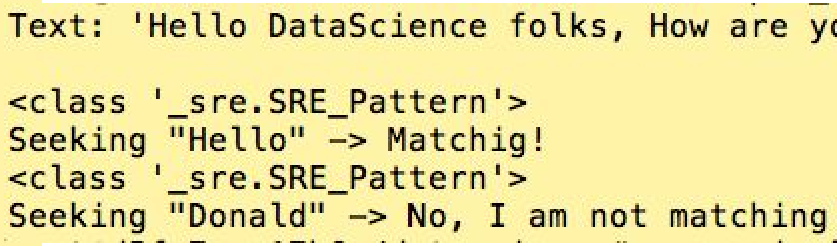
simple_match.py

Regular Expression (Cont ...)

Python supports compilation of pattern - it's more efficient to compile the pattern and use it. The compile() function converts an expression string into a RegexObject.



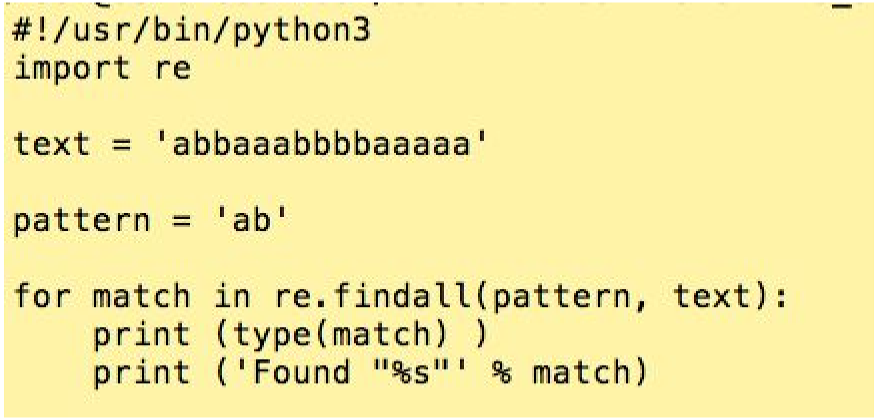
```
#!/usr/bin/python3
import re

# Precompile the patterns
regexes = [ re.compile(p) for p in ['Hello', 'Donald'] ]
text = 'Hello DataScience folks, How are you doing today?'

print('Text: {!r}\n'.format(text))

for regex in regexes:
    print (type(regex))
    print('Seeking "{}" ->'.format(regex.pattern), end=' ')

    if regex.search(text):
        print('Matchig!')
    else:
        print('No, I am not matching')
```



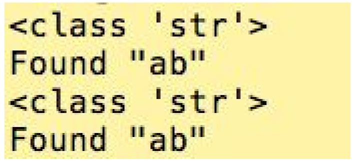
```
Text: 'Hello DataScience folks, How are yo

<class '_sre.SRE_Pattern'>
Seeking "Hello" -> Matchig!
<class '_sre.SRE_Pattern'>
Seeking "Donald" -> No, I am not matching
```

simple_compiled.py

Regular Expression (Cont …)

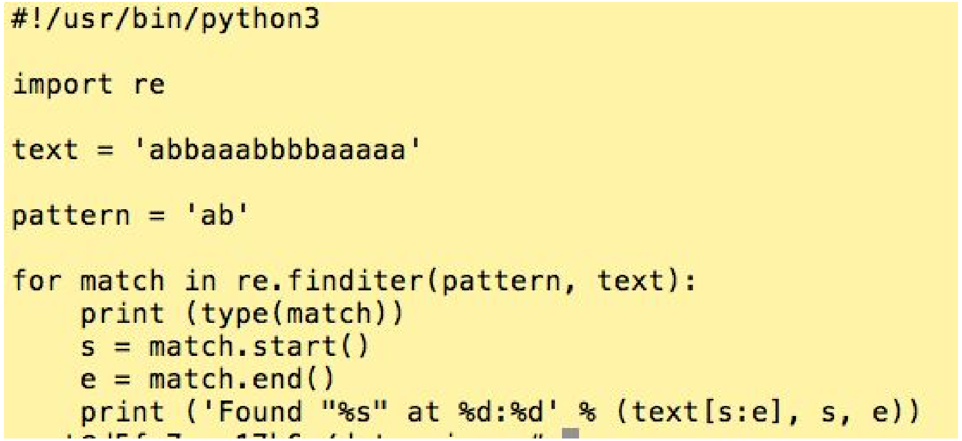


```
#!/usr/bin/python3
import re

text = 'abbaaabbbbaaaaa'

pattern = 'ab'

for match in re.findall(pattern, text):
    print (type(match) )
    print ('Found "%s"' % match)
```

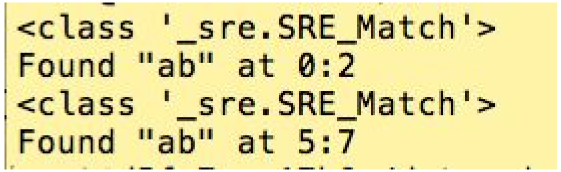


```
<class 'str'>
Found "ab"
<class 'str'>
Found "ab"
```

find_iter.py



```
#!/usr/bin/python3

import re

text = 'abbaaabbbbaaaaa'

pattern = 'ab'

for match in re.finditer(pattern, text):
    print (type(match))
    s = match.start()
    e = match.end()
    print ('Found "%s" at %d:%d' % (text[s:e], s, e))
```

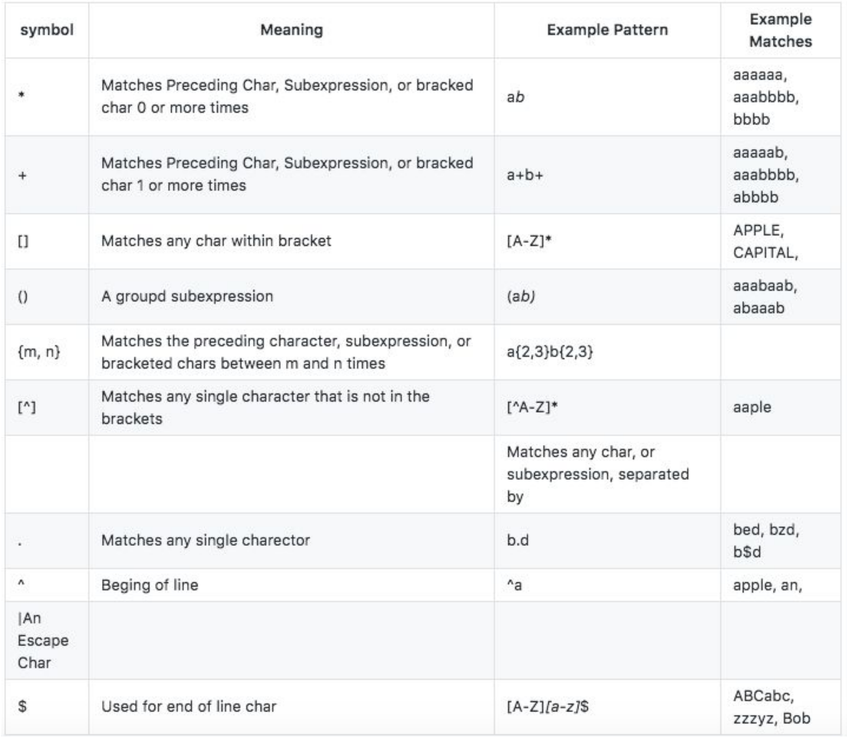


```
<class '_sre.SRE_Match'>
Found "ab" at 0:2
<class '_sre.SRE_Match'>
Found "ab" at 5:7
```

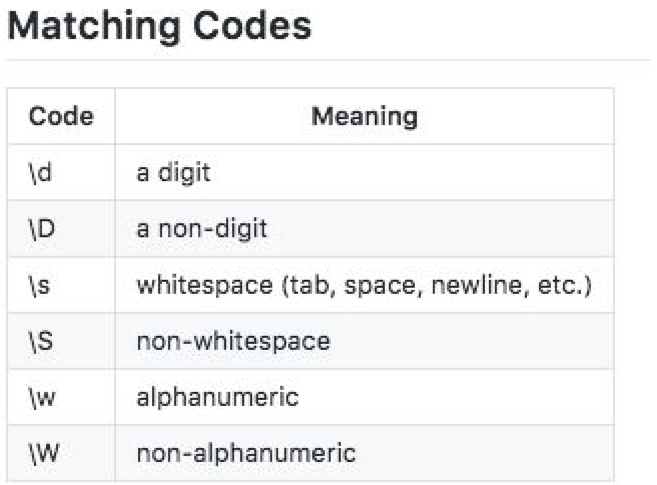
find_all.py

Regular Expression (Cont ...)

A Few Rules: Commonly Used RegEx symbols



symbol	Meaning	Example Pattern	Example Matches
*	Matches Preceding Char, Subexpression, or bracked char 0 or more times	*ab*	aaaaaa, aaabbbb, bbbb
+	Matches Preceding Char, Subexpression, or bracked char 1 or more times	a+b+	aaaaab, aaabbbb, abbbb
[]	Matches any char within bracket	[A-Z]*	APPLE, CAPITAL,
()	A groupd subexpression	*(ab)*	aaabaab, abaaab
{m, n}	Matches the preceding character, subexpression, or bracketed chars between m and n times	a{2,3}b{2,3}	
[^]	Matches any single character that is not in the brackets	[^A-Z]*	aaple
		Matches any char, or subexpression, separated by	
.	Matches any single charector	b.d	bed, bzd, b$d
^	Beging of line	^a	apple, an,
|An Escape Char			
$	Used for end of line char	[A-Z]*[a-z]*$	ABCabc, zzzyz, Bob



Matching Codes

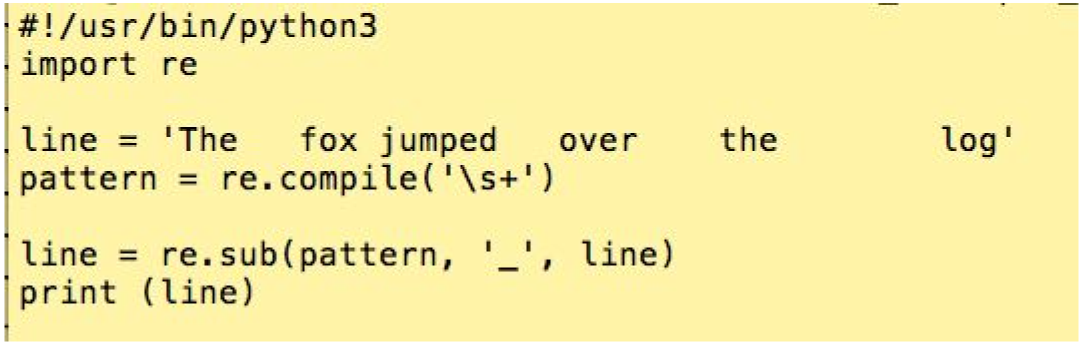
Code	Meaning
\d	a digit
\D	a non-digit
\s	whitespace (tab, space, newline, etc.)
\S	non-whitespace
\w	alphanumeric
\W	non-alphanumeric

Regular Expression (Cont …)

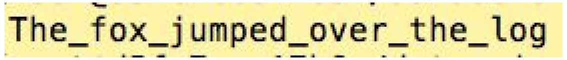
Mostly used functions from re module,

- compile(pattern, flags=0) – it compiles a regular expression pattern into a regular expression object, which can be used for matching using the match and search methods.
- match(pattern, string, flags=0) – if zero or more characters from the beginning of the string match, it returns a Match object, otherwise, it returns None.
- search(pattern, string, flags=0) – similar to match(), but it scans all the string, not only it’s beginning.
- sub(pattern, repl, string, count=0, flags=0) - Return the string obtained by replacing the leftmost non-overlapping occurrences of the pattern in string by the replacement repl. repl can be either a string or a callable; if a string, backslash escapes in it are processed. If it is a callable, it's passed the match object and must return a replacement string to be used.

Regular Expression (Cont ...)

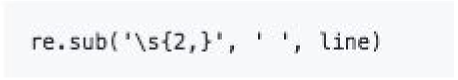


```
#!/usr/bin/python3
import re

line = 'The   fox jumped   over    the          log'
pattern = re.compile('\s+')

line = re.sub(pattern, '_', line)
print (line)
```



```
The_fox_jumped_over_the_log
```

How about following expression?



```
re.sub('\s{2,}', ' ', line)
```

remove_multiple_spaces.py

Assignment - 11

- Remove starting spaces in line
- Remove ending spaces in line
- How about removing all digits from line?

Develope Regular Expression to extract numbers (float, integers) from given text string. The numbers can be in any of the following format

```
#    '10.5', '-10.5', '- 10.5', '+ .2e10', ' 1.01e-2', '        1.01', '-.2', ' 456', ' .123'
```

Topics

- Hello Python
- Comments
- Mathematics
- Variables
- Data Types
- Making Decisions
- Strings
- Python Lists
- For, While - Loops
- Lists Revisits
- Handling Errors
- Functions
- Modules
- Packages
- Dictionaries
- Classes
- Iterators
- String Formatting
- Regular Expressions

File Read & Write Operations

Cover later

Thank you!