Data Science Python

- Anaconda3
  - Python 3.x
  - Includes ALL major Python data science packages
    - Sci-kit learn
    - Pandas
    - PlotPy
  - Jupyter Notebooks
- www.anaconda.com

Python - simple commands

Python is an interactive interpreter started from the shell:

```
lutz$ python
Python 3.5.2 |Anaconda 4.2.0 (64-bit)| (default, Jul  2 2016, 17:53:06)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 3 + 10.5
13.5
>>> 7/2
3.5
>>> print("hello world!")
hello world!
>>>
```

Python - loading and running a file

Python is also an environment to run program files:

Assume that we have the following Python program file ‘helloworld.py’:

```
"""
helloworld.py

This is the classic program every programmer writes when
he or she learns
a new programming language.
"""


def hello():
    "Just print 'hello world!' and that's it"
    print("hello world!") # print inserts a newline char
```

Red - docstring
Green - comment

Python - loading and running a file

Docstring vs Comment

- A docstring should document *what* your code does
  - Important for the *user* of your code
  - Docstrings are exported by Python into the help system
- A comment should comment on *how* your code does it
  - Important for your *peer programmers* modifying/understanding your code
  - Comments stay internal to the code

Python - loading and running a file

Load the file and run the function in Python:

```
lutz$ python
Python 3.5.2 |Anaconda 4.2.0 (64-bit)| (default, Jul  2 2016, 17:53:06)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import helloworld
>>> helloworld.hello()
hello world!
>>>
```

Functions belong to modules - if you want to execute a function in a module you have to provide the module name as a qualifier!

Python - loading and running a file

Docstrings shine!!! - automatically generated documentation of your module

The ‘help’ command

```
>>> import helloworld
>>> helloworld.hello()
hello world!
>>> help(helloworld)
```

```
Help on module helloworld:

NAME
    helloworld - helloworld.py

DESCRIPTION
    This is the classic program every programmer writes when he or she learns
    a new programming language.

FUNCTIONS
    hello()
        Just print 'hello world!' and that's it

FILE
    /home/lutz/Documents/CSC310/Python/slideset-1/helloworld.py
```

Python - import * considered dangerous

‘from <module> import *’

Any function or variable in <module> is imported into your local scope **WITHOUT** a module qualifier:

```
lutz@ip-172-31-26-47:~/Documents/CSC310/Python/slideset-1$ python
Python 3.5.2 |Anaconda 4.2.0 (64-bit)| (default, Jul  2 2016, 17:53:06)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from helloworld import *
>>> hello()
hello world!
>>>
```

Very Dangerous! - it can lead to silent name clashes with strange effects on your code!

Python - import * considered dangerous

Assume we have another file that defines the function ‘hello()’ then:

```
Python 3.5.2 |Anaconda 4.2.0 (64-bit)| (default, Jul  2 2016, 17:53:06)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more i
>>> from helloworld import *
>>> from helloagain import *
>>> hello()
hello again!
>>>
```

Silently overwrote the original hello() - the original is no longer available!!

```
"""
helloagain.py

Here we demonstrate that Python silently clobbers
names clashes if you are not careful.
"""


def hello():
    "Print out 'hello again!' and that's it"
    print("hello again!")
```

Never use ‘from <module> import *’ - you have no control over your name space!

Python – importing single functions

Assume we have another file that defines the function ‘hello()’ then:

```
Python 3.5.2 |Anaconda 4.2.0 (64-bit)| (default, Jul  2 2016, 17:53:06)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from helloworld import *
>>> from helloagain import hello
>>> hello()
hello again!
>>>
```

Still silently overwrote the original hello() but now it is easier to see why and how

Python - basic programming structures

The loop:

```
>>> for i in range(10):
...     print(i)
...
0
1
2
3
4
5
6
7
8
9
>>>
```

range(stop) -> range object
range(start, stop[, step]) -> range object

Returns an object that produces a sequence of integers from start (inclusive) to stop (exclusive) by step.
range(i, j) produces i, i+1, i+2, ..., j-1.
start defaults to 0, and stop is omitted! range(4) produces 0, 1, 2, 3.
These are exactly the valid indices for a list of 4 elements.
When step is given, it specifies the increment (or decrement).

Python - basic programming structures

The loop:

```
>>> list = [1,2,3]
>>> for e in list:
...     print(e)
...
1
2
3
>>> list = ['chicken','turkey','duck']
>>> for e in list:
...     print(e)
...
chicken
turkey
duck
>>>
```

Python - basic programming structures

The if-then-else statement:

```
>>> x = input("type a value: ")
type a value: 3
>>> x = int(x)
>>> if x==2:
...     print('x equals 2')
... else:
...     print('x is something else')
...
x is something else
>>>
```

Python - basic programming structures

The function definition statement:

```
>>> def inc(x):
...     return x+1
...
>>> inc(3)
4
>>>
```

Python - basic programming structures

A more complicated example - factorial computation:

```
>>> import fact
>>> fact.factorial(3)
6
>>>
```

```
"""
fact.py

An example of a recursive function to
 find the factorial of a number
"""


def factorial(x):
    """
    This is a recursive function to find the factorial of an
     integer x where x >= 0.  The function is not defined
     for x < 0.
    """
    if x == 0:
        return 1
    else:
        return x * factorial(x-1)
```

Python - Lists

```
>>> list = [1,2,3]
>>> list.append(4)
>>> list
[1, 2, 3, 4]
>>> list.reverse()
>>> list
[4, 3, 2, 1]
>>> list[0]
4
>>> list = [ ]
>>> list
[ ]
>>> len(list)
0
>>>
```

Things you can do with lists:

```
        append(...)
        clear(...)
        copy(...)
        count(...)
        extend(...)
        index(...)
        insert(...)
        pop(...)
        remove(...)
        reverse(...)
        sort(...)
```

See ‘help([ ])’

Python – List Comprehensions

- Comprehensions are a short hand notation for constructing lists

```
>>> S = [x**2 for x in range(10)]
>>> print(S)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Python – List Comprehensions

- Another example of list comprehensions with strings

```
>>> words = 'The quick brown fox jumps over the lazy dog'.split()
>>> print(words)
['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
>>> stuff = [[w.upper(), w.lower(), len(w)] for w in words]
>>> from pprint import pprint
>>> pprint(stuff)
[['THE', 'the', 3],
 ['QUICK', 'quick', 5],
 ['BROWN', 'brown', 5],
 ['FOX', 'fox', 3],
 ['JUMPS', 'jumps', 5],
 ['OVER', 'over', 4],
 ['THE', 'the', 3],
 ['LAZY', 'lazy', 4],
 ['DOG', 'dog', 3]]
```

Note: strings are objects with member functions!

Note: we are constructing a list of lists!

Python - Data Structures

Python has a number of data structures beyond lists that make programming much easier:

- Tuples
- Sets
- Dictionaries

Python - Tuples

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> x, y, z = t     # pattern matching!
>>> x
12345
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

- A tuple consists of a number of values separated by commas
- Though tuples may seem similar to lists, they are often used in different situations and for different purposes.
- *Tuples* are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking or indexing.
- *Lists* are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

Python - Sets

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False
```

A set is an unordered collection with no duplicate elements.

Python - Sets

```
>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> b = set('alacazam')
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

A set is an unordered collection with no duplicate elements.

Python - Sets

```
>>> # set comprehensions
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

A set is an unordered collection with no duplicate elements.

Python - Dictionaries

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

A dictionary is an unordered set of key:value pairs, with the requirement that the keys are unique (within one dictionary).

Python

Install Anaconda on your system

If you are not familiar with Python read Jake VanderPlas’ intro to Python:

http://www.oreilly.com/programming/free/files/a-whirlwind-tour-of-python.pdf