UC Berkeley Extension

Python for Data Analysis and Scientific Computing

X433.3 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

Language specifics

- Iterative programming (for/continue/while/break)
  - the for loop
    - commonly used loop for iterative calculation of certain portion of a program beginning from '0'
    - passing through the for line the first time around evaluates 'k' to the first element of a given set
    - the increment of the variable is done in the for line after the sign ':' the second time around
    - incrementing is automatically taken care of
    - this is safer as the programmer doesn't need to think about the increment leading to less errors

Examples: *simple for loops*

```
In [152]: for k in ('Sarah', 'cars', 'Python'):
              print('John likes %s' % k)
John likes Sarah
John likes cars
John likes Python

In [153]: for j in range(3):
              print(j)
0
1
2
...
```

Language specifics

- Iterative programming (for/continue/while/break)
  - the for loop
    - commonly used loop for iterative calculation of certain portion of a program beginning from '0'
    - passing through the for line the first time around evaluates 'k' to the first element of a given set
    - the increment of the variable is done in the for line after the sign ':' the second time around
    - incrementing is automatically taken care of
    - this is safer as the programmer doesn't need to think about the increment leading to less errors

Examples: *special range generator with yield*

```
In [154]: def new_range(start, end, step):
              while start <= end:
                  yield start          # yield is a generator preserving funct. local value
                  start += step
In [155]: for x in new_range(2, 5, 0.2): print(x)
2
2.2
2.4
...
```

class exercise

Language specifics

- Iterative programming (for/continue/while/break)
  - the 'continue' option
    - skips the current iteration and continues to the next iteration in a loop

Example:

```
 99  ## Example using 'continue':
100  x = [41, 12, 34, 52]
101  for k in x:
102      if k == 34:
103          continue
104      print(k)
```

… will produce:

```
41
12
52
```

Language specifics

- Iterative programming (for/continue/while/break)
  - the while loop
    - just like for with main difference that the increment is done manually inside the loop
    - the increment doesn't have to start from '0' like in the for loop
    - the increment can be done anywhere in the while loop
    - there is always the need to include one extra line for incrementing unlike in the for loop
    - this is not very safe as the programmer may forget and other problems may occur
  - break
    - provides an alternative exit from for or while when certain condition is met
    - the iteration in the loop stops after the break condition is met

Language specifics

- Iterative programming (for/continue/while/break)
  - while loop and break

Example:

```
87  ## Example for 'while' loop:
88  a = 6 + 4.5j
89  b = 1
90  while b<a.real:
91      a=a**0.5+0.3
92      print(a)
93      print(b)
94      b=b+1
95      if a.imag < 0.5:
96          print('The imaginary part fell below 0.5. Will exit now!')
97          break
```

… will produce:

```
(2.898076211353316+0.8660254037844387j)
1
(2.020869271954432+0.25162440224203464j)
2
The imaginary part fell below 0.5. Will exit now!
```

… class exercise

Language specifics

- The with statement
  - with is used when working with unmanaged resources (like file streams)
  - It allows us to ensure that a resource is cleaned up when the code that uses it finishes running, even if exceptions are thrown

Example 1:

```
with expression [as variable]:
    with-block
```

Example 2:

```
with open('/etc/passwd', 'r') as f:
    for line in f:
        print line
        ... more code here ...
```

  - The file object in f above will automatically close, even if the for loop raised an exception through the block

Language specifics

- Functions: definition, return values, local vs. global variables
  - functions are separate blocks of code in Python's program that are dedicated to perform a specific routine
  - they can be called multiple times
  - they must be defined before being used
  - defining a function happens with the keyword def followed by the name of the function, parenthesis, that take arguments, and colon at the end ':'

    ```
    def alex_fun_test():          -> it does not take any parameters
    ```

  - they may or may not take values when executing their routine

    ```
    def alex_fun_test(n):         -> it takes 'n' as input parameter to be used inside the function call
    ```

Language specifics

- Functions: definition, return values, local vs. global variables
  - they may or may not return values after being executed

    def alex_fun_test(n): -> it takes 'n' as input parameter to be used inside the function call

    return n=n**2 -> this is the body of the function
  - after the definition of the function there is the body
  - functions return 'None' by default
  - once defined functions can be called any time in the code
  - functions work with local and global variables

Example:

```
106  ## Example of function definition, return values, local vs. global variables:
107  a = 12                          # -> define global variable 'a' of type 'int'
108  def alex_fun_test(b):           # -> call 'alex_fun_test' with input argument 'b'
109      c = 41                      # -> define local variable 'c' of type 'int'
110      return a + b + c
```

… so the call:

alex_fun_test(34)

… will produce:

87

… class exercise

Language specifics

- Functions: definition, return values, local vs. global variables
  - when a function that must take at least one input parameter is called without it, this results in error

Example:

```
106  ## Example of function definition, return values, local vs. global variables:
107  a = 12                          # -> define global variable 'a' of type 'int'
108  def alex_fun_test(b):           # -> call 'alex_fun_test' with input argument 'b'
109      c = 41                      # -> define local variable 'c' of type 'int'
110      return a + b + c
```

… so the call:

```
alex_fun_test()
```

… will produce:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/Users/alex/1.HD/Alex/1.new/Work/3.Berkeley Extension/3. final course material/2.
de/lecture2.py in <module>()
----> 1 alex_fun_test()

TypeError: alex_fun_test() missing 1 required positional argument: 'b'
```

Language specifics

- Functions: definition, return values, local vs. global variables
  - functions can be called with optional parameters as well

Example:

```
112  ## Example of function definition, return values, local vs. global variables:
113  def fun_optional(d=12):
114      return d + 34
```

… so the call:

fun_optional ()

… will produce:

46

… and the call:

fun_optional(41)

… will produce:

75

Course Content Outline

- **Introduction to Python®**
- Python - pros and cons
- Installing the environment with core packages
- Python modules, packages and scientific blocks
- Working with the shell, IPyton and the editor HW1

- **Language specifics 1/2**
- Basic arithmetic operations, assignment operators, data types, containers
- Control flow (if/elif/else)
- Conditional expressions
- Iterative programming (for/continue/while/break)
- Functions: definition, return values, local vs. global variables

- **Language specifics 2/2**
- Classes / Functions (cont.): objects, methods, passing by value and reference
- Scripts, modules, packages
- I/O interaction with files
- Standard library
- Exceptions

- **NumPy 1/3**
- Why NumPy?
- Data type objects
- NumPy arrays
- Indexing and slicing of arrays HW2

- **Matplotlib**
- What is Matplotlib?
- Basic plotting
- Tools: title, labels, legend, axis, points, subplots, etc.
- Advanced plotting: scatter, pie, bar, 3D plots, etc. HW3

Language specifics

- Functions: objects, methods, passing by value and reference
  - an object is something allocated in memory
  - variables are objects
  - functions are objects
  - functions can be assigned to variables
  - they can be passed as an argument to different functions
  - a function can also be an item in a collection
  - everything in Python is an object. Example: *int* is an object
  - objects have identity (names) so that we can tell if they are the same or different objects

Language specifics

- Functions: objects, methods, passing by value and reference
  - the name of an object is not part of the object
  - the name of an object exist in the namespace
  - every object has a specific location in memory
  - in Python objects have an ID that reveals their memory location
  - objects have a particular type (*int, list, tuple, etc.*)
  - every object has only one type
  - every object has a value with different attributes

Example: X = 34 -> here, 34 is of type 'int'. Objects are: '34', 'int' itself, the 'type' applied to 'X' in order to find that it is of 'int' 'type'

Language specifics

- Classes : objects, methods, passing by value and reference
  - methods are functions that are members of a class
  - they are functions attached to objects
  - a class can be called with different methods (functions) that it consist of

Example:

```
1  ## Class example:
2  # Simple class:
3  class simple_class:
4      """This class shows basic functionality"""
5      a = 12
6      def f():
7          return 'hello world'
```

after executing the code above we try:

We create an ---> instance of a class

```
In [3]: id(Class_1)
Out[3]: 4536329720

In [4]: id(simple_class)
Out[4]: 140367794155560
```

```
Python

In [1]: Class_1 = simple_class()

In [2]: whos
Variable       Type            Data/Info
-----------------------------------------
Class_1        simple_class    <__main__.simple_class object at 0x10603deb8>
name           str             __loader__
simple_class   type            <class '__main__.simple_class'>
```

Language specifics

- Classes : objects, methods, passing by value and reference
  - methods are functions that are members of a class
  - they are functions attached to objects
  - a class can be called with different methods (functions) that it consist of

Example:

```
1  ## Class example:
2  # Simple class:
3  class simple_class:
4      """This class shows basic functionality"""
5      a = 12
6      def f():
7          return 'hello world'
```

after executing the code above we try:

```
Python

In [1]: Class_1 = simple_class   <-- notice the difference

In [2]: whos
Variable       Type    Data/Info
--------------------------------
Class_1        type    <class '__main__.simple_class'>
name           str     __loader__
simple_class   type    <class '__main__.simple_class'>
```

We create an ---> instance of a class

```
In [3]: id(Class_1)
Out[3]: 140558349856568

In [4]: id(simple_class)
Out[4]: 140558349856568
```

Language specifics

- Classes : objects, methods, passing by value and reference
  - methods are functions that are members of a class
  - they are functions attached to objects
  - a class can be called with different methods (functions) that it consist of

Example:

```
1  ## Class example:
2  # Simple class:
3  class simple_class:
4      """This class shows basic functionality"""
5      a = 12
6      def f():
7          return 'hello world'
```

after executing the code above we try:

```
Python

In [3]: Class_1.a
Out[3]: 12

In [4]: simple_class.f()
Out[4]: 'hello world'

In [5]: simple_class.a = 9

In [6]: Class_1.a
Out[6]: 9

In [7]: Class_1.__doc__
Out[7]: 'This class shows basic functionality'

In [8]: Class_1?
Init signature: Class_1()
Docstring:      This class shows basic functionality
File:           ~/1.HD/Alex/Work/3.Berkeley Extension/
Type:           type
```

… class exercise

Language specifics

- Classes Inheritance : objects, methods, passing by value and reference
  - Inheritance is when a class behavior can be copied by an instance of a class object
  - Classes can inherit attributes and behavior methods from other classes called the superclasses
  - A class that inherits from a superclass is called a subclass, (also *heir* or *child* class)
  - There is a hierarchy among classes

Example:

```
12  # Base class - super class:
13  class class_test:
14      def method_one():
15          print('This is method 1')
16      def method_two():
17          print('I am method 2')
18
19  # Inheritance - subclass / derived class:
20  class class_test_two(class_test):
21      def method_one():
22          print('This is new method 1') # method overriding from super class
23      def method_three():
24          print('This is method 3')
```

Language specifics

- Classes Inheritance : objects, methods, passing by value and reference
  - Inheritance is when a class behavior can be copied by an instance of a class object
  - Classes can inherit attributes and behavior methods from other classes called the superclasses
  - Unlike Java and C#, python allows multiple inheritance - inherit from multiple classes at the same time like this: >>> class Subclass(SuperClass1, SuperClass2, ...)

Example:

```
12  # Base class - super class:
13  class class_test:
14      def method_one():
15          print('This is method 1')
16      def method_two():
17          print('I am method 2')
18
19  # Inheritance - subclass / derived class:
20  class class_test_two(class_test):
21      def method_one():
22          print('This is new method 1') # method overriding
23      def method_three():
24          print('This is method 3')
```

```
Python

In [1]: class_test.method_one()
This is method 1

In [2]: class_test.method_two()
I am method 2

In [3]: class_test_two.method_one()
This is new method 1

In [4]: class_test_two.method_two()
I am method 2

In [5]: class_test_two.method_three()
This is method 3

In [6]: class_test_two.__base__
Out[6]: __main__.class_test
```

... class exercise

Language specifics

- Classes Polymorphism : objects, methods, passing by value and reference
  - Polymorphism is based on the Greek words: Poly (*many*) and morphism (*forms*)
  - So it is a structure that can take or use many forms of objects

Example:

```
26  ## Polymorphism example:
27  class Animal:
28      def __init__(self, name):    # Constructor of the class
29          self.name = name
30      def talk(self):              # Abstract method, defined by convention only
31          raise NotImplementedError("Subclass must implement abstract method")
32
33  class Cat(Animal):
34      def talk(self):
35          return 'Meow!'
36
37  class Dog(Animal):
38      def talk(self):
39          return 'Woof! Woof!'
40
41  animals = [Cat('Tiger'),
42             Cat('Kitty'),
43             Dog('Maxie')]
44
45  def animal_sounds():
46      for animal in animals:
47          print(animal.name + ': ' + animal.talk())
```

```
In [69]: Dog.talk(Dog)
Out[69]: 'Woof! Woof!'
```

```
In [70]: animal_sounds()
Tiger: Meow!
Kitty: Meow!
Maxie: Woof! Woof!
```

… class exercise

Language specifics

- Functions: objects, methods, passing values to functions

Example:

```
1  ## Example of functions: passing values
2
3  def fun_two(a, b, c):
4      a = 12
5      print(a)
6      b = [34]
7      print(b)
8      c.append(41)
9      print(c)
```

after executing the code above we enter these lines:

a_2 is immutable (int)
b_2 is a mutable variable (list)
c_2 is a mutable variable (list)

```
Python

In [5]: a_2 = 51

In [6]: b_2 = [85]

In [7]: c_2 = [47]

In [8]: fun_two(a_2,b_2,c_2)
12
[34]
[47, 41]

In [9]: print(a_2)
51

In [10]: print(b_2)
[85]

In [11]: print(c_2)
[47, 41]
```

Language specifics

- Functions: objects, methods, passing by value and reference

  - Pass by value call means the called functions' parameter will be **a copy** of the callers passed argument
    - user is dealing with the actual value of a variable directly
  - Pass a reference to a function means the called functions' parameter will be **==** as the callers' passed argument (not the **value**, but the identity - the variable obj. itself)
    - user is dealing with the memory location of a variable rather than its actual value
  - the question is if a variable can be modified after being passed to a function or not
  - in Python the difference between the two ways of sharing variables is somewhat elusive

Language specifics

- Functions: objects, methods, passing by value and reference

  Rules:

  - when a variable is passed to a function, the reference to the object, to which the variable refers is actually passed, and not the variable
  - when an immutable value is passed to a function, the function can not change the variable
  - when a mutable value is passed to a function, the function can change the variable
  - variables declared in functions exist in a local table known as *local namespace* and have local scope

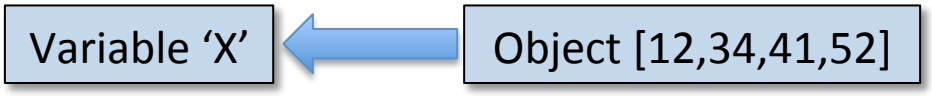
Language specifics

- Functions: objects, methods, passing by value and reference

  Pass-by-reference

  – Suppose we have the list:
    - X = [12,34,41,52]

    X is the variable that points to the object that is the list [12,34,41,52], but X itself is NOT the list



  – Variables can be looked at as containers that hold objects

  – in the pass-by-reference case the variable is passed directly into the function along with its contents, that is the object

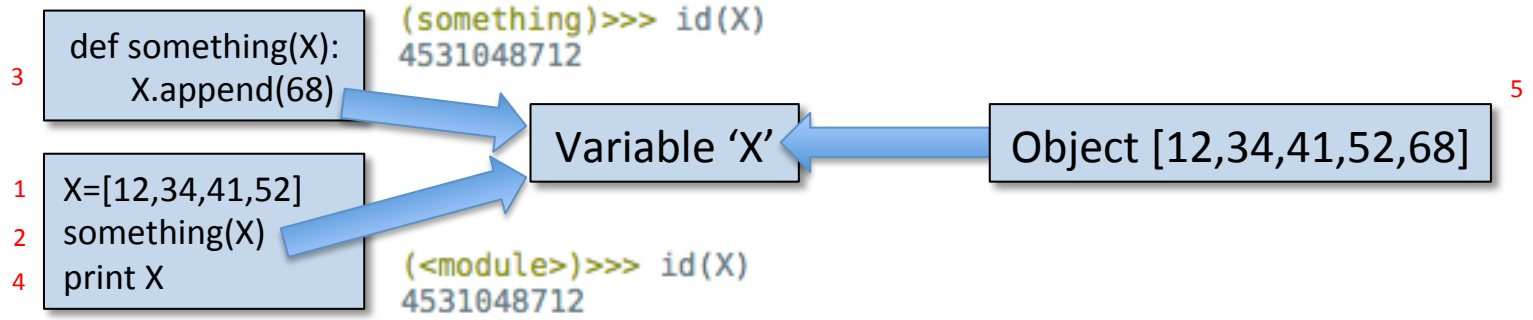
Language specifics

- Functions: objects, methods, passing by value and reference

  Pass-by-reference

  – the argument created inside the function is exactly the same container as the one passed by the caller

  – it refers to the same object in memory



  – it follows that any action performed to the variable or the object inside the function 3 will be visible by the caller 2

  – so the changes will be visible outside the function

Conclusion: when using pass-by-reference, both the function and the caller use the same exact variable and object / memory location

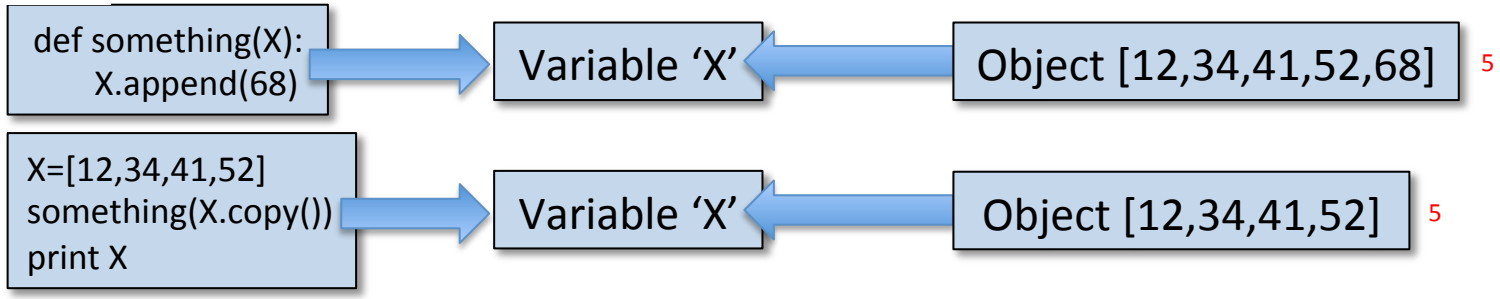
Language specifics

- Functions: objects, methods, passing by value and reference

Pass-by-value

- in this case the function receives a copy of the object's argument passed by the caller
- it is stored in a new memory location
- in essence the function now provides it's own container for the value
- there is no relationship between the variables nor the objects referred to by the function and the caller
- the objects have the same value, but they are different
- it follows that nothing that happens to one will affect the other
- so the changes inside the function will not be visible outside

```
(something)>>> id(X)
4572863176
```




```
(<module>)>>> id(X)
4573462088
```

Conclusion: the different copies of the variable and the object do not affect one another

Language specifics

- Functions: objects, methods, passing by object in Python
  - in Python there is not exactly a reference to a value or reference in the same way as in C++
  - Python uses a reference to an object, which binds the name of a variable to an object in memory
  - Python creates a new name for the same object when calling functions, so changing the object shows in the caller, however assigning to the function-local variable is not reflected in the caller just like in Java or Lisp
  - To make this slightly easier, always think of the mutable and immutable objects:
    - Changing mutable objects can change the object directly, hence changing an object inside a function or a method will also change the original object outside
    - Changing immutable objects inside a function or a method will create a new instance and the original instance outside that function or a method is not changed

Language specifics

- Scripts, modules, packages
  - scripts are longer code collections stored in a file
  - scripts are used so that there is no need to type everything in the interpreter
  - it is a convenient way to store larger pieces of code to be executed any time
  - scripts contain sequence of instructions
  - indentation used in scripts is very convenient because increases readability
  - scripts can be written in any text editor of choice (we have one in Pyzo)
  - scripts are also referred to as modules
  - scripts have the extension .py

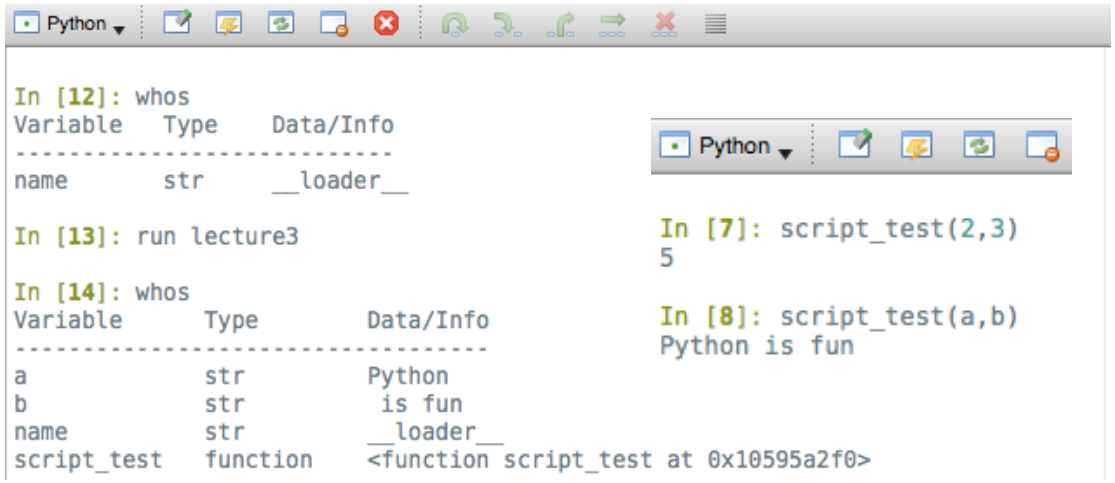
Language specifics

- Scripts, modules, packages

Example: … save the code below in a file called 'lecture3.py':

```
17  a = ('Python')
18  b = (' is fun')
19  def script_test(a,b):
20      print(a+b)
```

… check what's loaded in namespace and run the script from the same directory :



… after the script was executed we see new objects exist in namespace

Language specifics

- Scripts, modules, packages

Example: … now call the already defined function and observe the result:

```
In [15]: script_test(a,b)
Python is fun
```

- this script can also be executed as a standalone file in the Unix/Linux terminal or using the cmd Win console
- make sure you are in the same directory where 'lecture3.py' resides

Example: … we slightly modify the script by adding "print('It works')" before we run it:

```
17  a = ('Python')
18  b = (' is fun')
19  def script_test():
20      print(a+b)
21  print('It works')
```

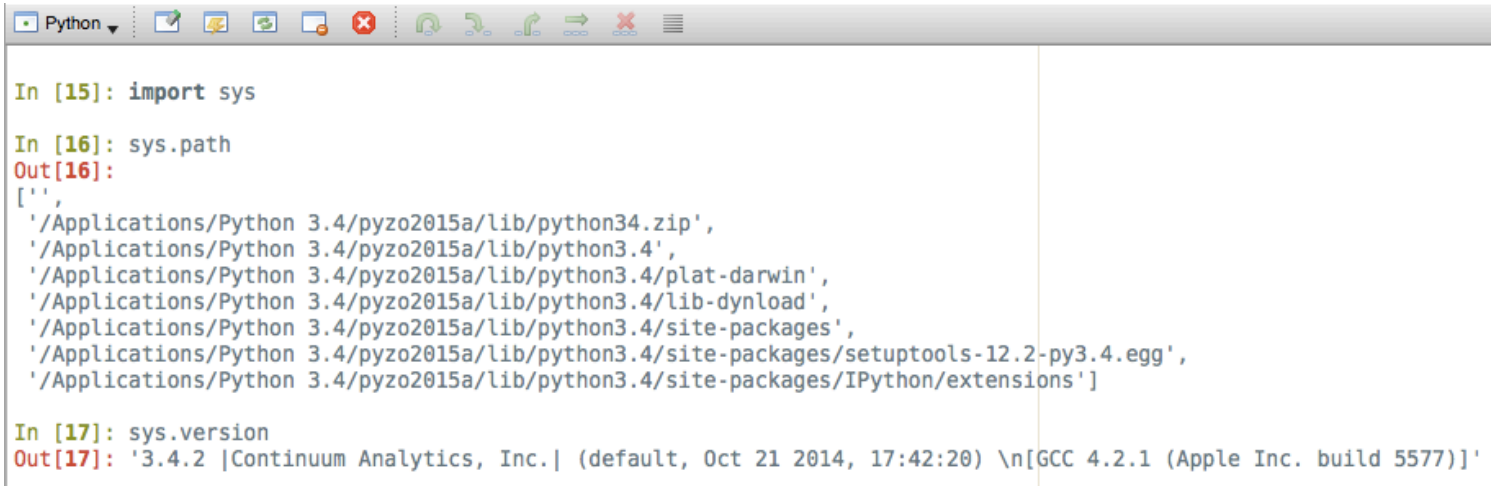
… now run the code using your terminal or cmd window:

```
Macintosh:code alex$ python lecture3.py
It works
Macintosh:code alex$
```

Language specifics

- Scripts, modules, packages
  - modules are scripts that provide a better way of organizing your code in a hierarchical way
  - the tools for scientific computing provided by numpy and scipy are modules, but they themselves are packages
  - modules, packages and sub-packages must be imported before they are used
  - then you can use the extended functionality a module provides

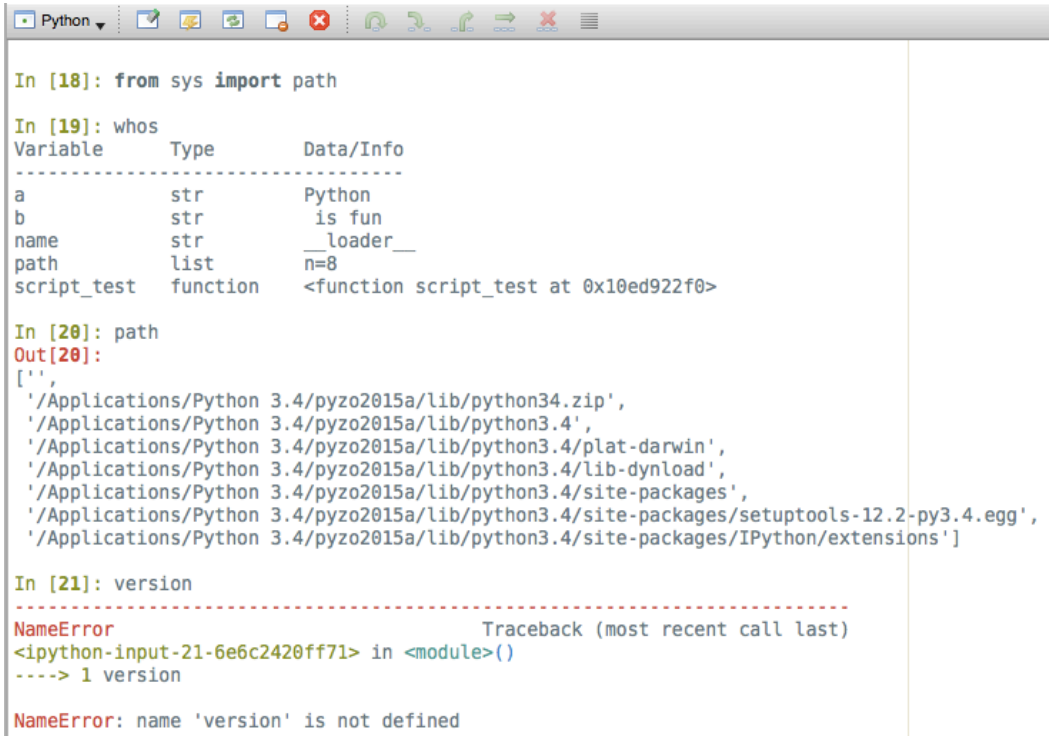
  Example:



Language specifics

- Scripts, modules, packages
  - in some cases it is better to import only parts of modules / packages we need
  - this is good because we won't take extra memory for functionality we won't use from a module
  - the drawback in this approach is that you won't be able to use other methods of the module
  - in this way the user can type shorter commands like: 'path' rather than 'sys.path'

Example:



Language specifics

- Scripts, modules, packages
  - sometimes it is better to refer to a module with a different alias after being imported

```
In [23]: import numpy as np
```

```
In [23]: whos
Variable   Type       Data/Info
------------------------------
a          ndarray    2x2: 4 elems, type `int64`, 32 bytes
b          ndarray    49: 49 elems, type `float64`, 392 bytes
c          ndarray    49: 49 elems, type `float64`, 392 bytes
name       str        __loader__
np         module     <module 'numpy' from '/Ap<...>kages/numpy/__init__.py'>
```

Example:

```
23  import numpy as np
24  a = np.array([[12, 34], [41, 54]])
25  b = np.arange(0.5,4*np.pi,0.25)
26  c = np.sin(b)
27  print(c)
```

… the code above will produce:

```
In [24]: run lecture3
[ 0.47942554  0.68163876  0.84147098  0.94898462  0.99749499  0.98398595
  0.90929743  0.7780732   0.59847214  0.38166099  0.14112001 -0.10819513
 -0.35078323 -0.57156132 -0.7568025  -0.89498936 -0.97753012 -0.99929279
 -0.95892427 -0.85893449 -0.70554033 -0.50827908 -0.2794155  -0.03317922
  0.21511999  0.45004407  0.6569866   0.82308088  0.93799998  0.99459878
  0.98935825  0.92260421  0.79848711  0.62472395  0.41211849  0.17388949
 -0.07515112 -0.31951919 -0.54402111 -0.73469843 -0.87969576 -0.96999787
 -0.99999021 -0.967808   -0.87545217 -0.72866498 -0.53657292 -0.31111935
 -0.0663219 ]
```

Language specifics

- Scripts, modules, packages
  - we can also use a different way to import a module using the '*' sign:

```
In [26]: from sys import *
```

  - this notation is called star import and it means import all … names for access from the module
  - **import** * will import everything, except the names that start with _ as they are private
  - every name that does not have _ is considered public and access to it will be granted
  - an exception to this rule is when the module has the __all__ option in the beginning
  - __all__ specifies what names exactly will be made available when an **import** * call is made regardless if they are meant to be public or private

Language specifics

- Scripts, modules, packages

Example: … consider we have the following code:

```
30  public_var = 12
31  _private_var = 34
32
33  def public_fun():
34      print('This function is public')
35  def _private_fun():
36      print('... now this function is set to be private')
37
38  class PublicClass():
39      print('This is a public class')
40  class _PrivateClass():
41      print('... now this Class is private')
```

… then the only exposed names in this '*' call will be:

```
Python

In [27]: from lecture3 import *

In [28]: whos
Variable      Type        Data/Info
-----------------------------------
PublicClass   type        <class 'lecture3.PublicClass'>
name          str         __loader__
public_fun    function    <function public_fun at 0x10ed0cd90>
public_var    int         12
```

… class exercise

Language specifics

- Scripts, modules, packages

Example: … lets try the same code with the only addition the '__all__' option:

```
30  public_var = 12
31  _private_var = 34
32
33  __all__ = ['public_var','_private_fun']
34
35  def public_fun():
36      sum_pub_priv = public_var + _private_var
37      print('This function is public. The sum is', sum_pub_priv)
38  def _private_fun():
39      print('... now this function is set to be private')
40
41  class PublicClass():
42      print('This is a public class')
43  class _PrivateClass():
44      print('... now this Class is private')
```

… then the only exposed names in this '*' call will be:

```
Python

In [29]: from lecture3 import *

In [30]: whos
Variable     Type    Data/Info
-----------------------------
name         str     __loader__
public_var   int     12

In [31]: _private_fun()
... now this function is set to be private

In [32]: public_var
Out[32]: 12

In [33]: private_var
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-33-2e9c0dc51fa1> in <module>()
----> 1 private_var

NameError: name 'private_var' is not defined
```

… class exercise

Language specifics

- Scripts, modules, packages

Caution: star import is NOT recommended and should be avoided for the following reasons:

  - it is not explicit, which means that we don't know anything about what is being imported, which is why it is much more clear to import variables, functions or classes one by one as needed
  - the namespace can be cluttered
  - this may create name overrides between different modules that are loaded
  - it is impossible to understand the functionality of everything that is loaded just by looking at the names of different variables, functions or classes that were imported

Language specifics

- Scripts, modules, packages

Caution: star import is NOT recommended and should be avoided for the following reasons:

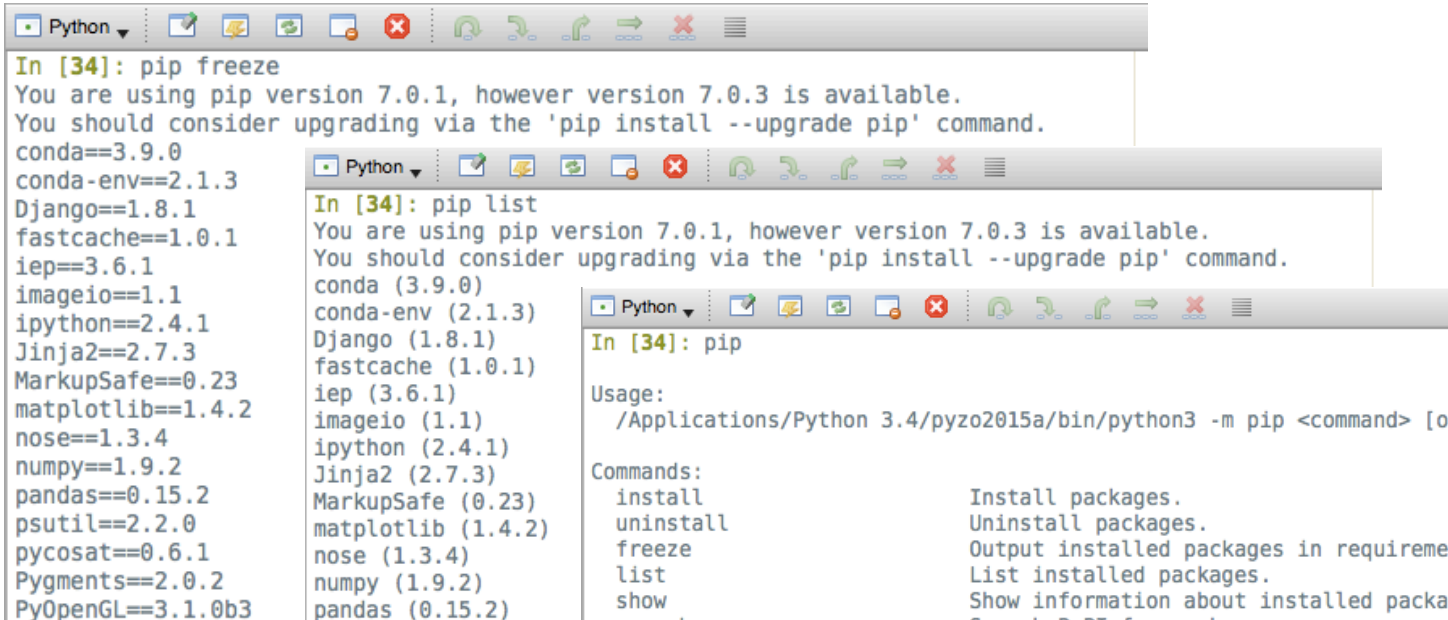
  - it decreases readability of the code and it is hard to understand it
  - the tab completion is not functioning as it is supposed to – to provide simplicity
  - it is impossible to check for different undefined symbols
  - there may be problems with visibility of different variables or functions declared in '__all__'
  - finally, this way of loading has its benefit because it is quick, but it is a lazy way of doing it and the user must know what exactly needs before using it

Language specifics

- Scripts, modules, packages
  - a module is a single file whereas a package is a folder containing number of modules
  - in the package folder there must be a file named '__init__.py' that describes what is included in this package
  - the '__init__.py' file distinguishes a Python package from a regular file folder
  - packages can be nested, so that subfolders may have an '__init__.py' file and can contain more modules
  - when importing a module or a package Python creates the same type of *module* object
  - when importing a package you are importing bunch of modules (files), which is why it is always better to only import particular modules from a package in order not to clutter the namespace
  - when a package is imported, only variables, functions and classes specified in the '__init__.py' file are loaded. That rule does not go for any sub-packages or modules
  - when using star import for packages, '__all__' specifies the modules that will be loaded into the current namespace (**not what is specified in** the '__all__' file inside each module)
  - when the '__all__' declaration is omitted in the '__init__.py' file of a package, the statement 'from <package> import *' will not import anything at all

Language specifics

- Scripts, modules, packages
  - some packages come with Python core installation:
    - email, http, html (modules), io, json, test, xml (modules), etc.
  - others have to be installed separately:
    - pandas, numpy, scipy, matplotlib, sympy, requests, django, pillow, SQLAlchemy, pygame, pyglet etc.
  - to check for available packages and their versions on your machine use the pip command



Language specifics

- Scripts, modules, packages
    - Some of the top extended Python packages are:

    - NumPy – provides an advance math functionality
    - SciPy – provides a rich library for scientific computations and works well with NumPy
    - Matplotlib – advanced plotting capability that is well integrated with NumPy and SciPy
    - Pandas – provides a high-level data manipulation toolset built on top of NumPy
    - SymPy – providing algebraic evaluation, differentiation, complex numbers, etc.
    - Requests – the top choice for any http web design
    - Pillow – great tool for image processing
    - IPython – makes Python easy to use with its shell, history, library and own text editor
    - Pygame – for 2D game development
    - Pyglet – for 3D game development
    - Django – a rich web framework for pro development
    - Kyvi - Open source *Python* framework for rapid development of mobile applications