UC Berkeley Extension

Python for Data Analysis and Scientific Computing

X433.3 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

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

- **Matplotlib**
- What is Matplotlib?
- Basic plotting
- Tools: title, labels, legend, axis, points, subplots, etc.
- Advanced plotting: scatter, pie, bar, 3D plots, etc. HW2

Language specifics

- Iterative programming (<u>for</u>/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 (<u>for</u>/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/<u>continue</u>/while/break)
  - the ‘continue’ option
    - skips the current iteration and continues to the next iteration in a loop

Example:

```python
## Example using 'continue':
x = [41, 12, 34, 52]
for k in x:
    if k == 34:
        continue
    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:

```
## Example for 'while' loop:
a = 6 + 4.5j
b = 1
while b<a.real:
    a=a**0.5+0.3
    print(a)
    print(b)
    b=b+1
    if a.imag < 0.5:
        print('The imaginary part fell below 0.5. Will exit now!')
        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 processing code ...
  ```

  - 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:

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

```
## Example of function definition, return values, local vs. global variables:
a = 12                          # -> define global variable ‘a’ of type ‘int’
def alex_fun_test(b):           # -> call ‘alex_fun_test’ with input argument ‘b’
    c = 41                      # -> define local variable ‘c’ of type ‘int’
    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:

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

… so the call:

fun_optional ()

… will produce:

46

… and the call:

fun_optional(41)

… will produce:

75

Language specifics

- Functions: <u>objects</u>, methods, passing by value and reference
  - an object is something allocated in memory
  - variables are objects
  - functions are objects
  - functions and variables are not different in Python in the way they are addressed as they point to an object
  - 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:

```
## Class example:
# Simple class:
class simple_class:
    """This class shows basic functionality"""
    a = 12
    def f():
        return 'hello world'
```

after executing the code above we try:

We create an ---> instance of a class

```
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'>
```

… class exercise

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:

```python
## Class example:
# Simple class:
class simple_class:
    """This class shows basic functionality"""
    a = 12
    def f():
        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'
```

… 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:

```
# Base class - super class:
class class_test:
    def method_one():
        print('This is method 1')
    def method_two():
        print('I am method 2')

# Inheritance - subclass / derived class:
class class_test_two(class_test):
    def method_one():
        print('This is new method 1') # method overriding from super class
    def method_three():
        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:

```python
# Base class - super class:
class class_test:
    def method_one():
        print('This is method 1')
    def method_two():
        print('I am method 2')

# Inheritance - subclass / derived class:
class class_test_two(class_test):
    def method_one():
        print('This is new method 1') # method overriding
    def method_three():
        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:

```python
## Polymorphism example:
class Animal:
    def __init__(self, name):    # Constructor of the class
        self.name = name
    def talk(self):              # Abstract method, defined by convention only
        raise NotImplementedError("Subclass must implement abstract method")

class Cat(Animal):
    def talk(self):
        return 'Meow!'

class Dog(Animal):
    def talk(self):
        return 'Woof! Woof!'

animals = [Cat('Tiger'),
           Cat('Kitty'),
           Dog('Maxie')]

def animal_sounds():
    for animal in animals:
        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, <u>passing by value and reference</u>

  - 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

Example:

```python
## Example of functions: passing by value and reference

def fun_two(a, b, c):
    a = 12
    print(a)
    b = [34]
    print(b)
    c.append(41)
    print(c)
```

after executing the code above we enter these lines:

```
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]
```

a_2 is immutable (int)

b_2 is a mutable variable (list)

c_2 is a mutable variable (list)

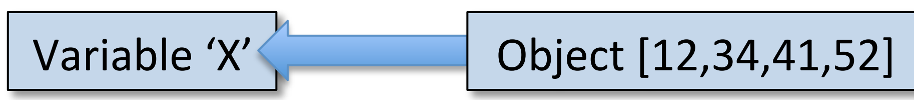
Language specifics

- Functions: objects, methods, <u>passing by value and reference</u>

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 <u>NOT</u> the list



– Variables can be looked at as containers that hold objects

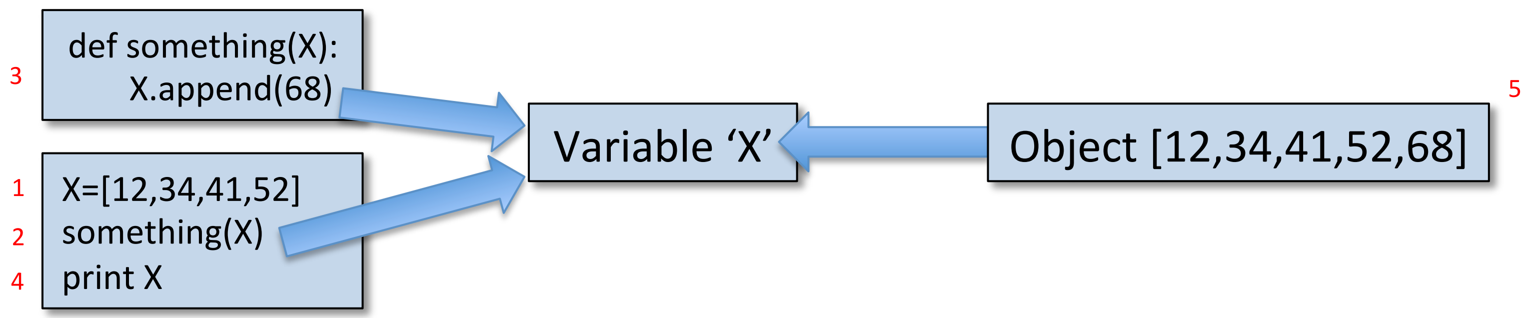
– in the pass-by-reference case the variable is passed directly into the function along with its contents, that is the object

Language specifics

- Functions: objects, methods, <u>passing by value and reference</u>

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

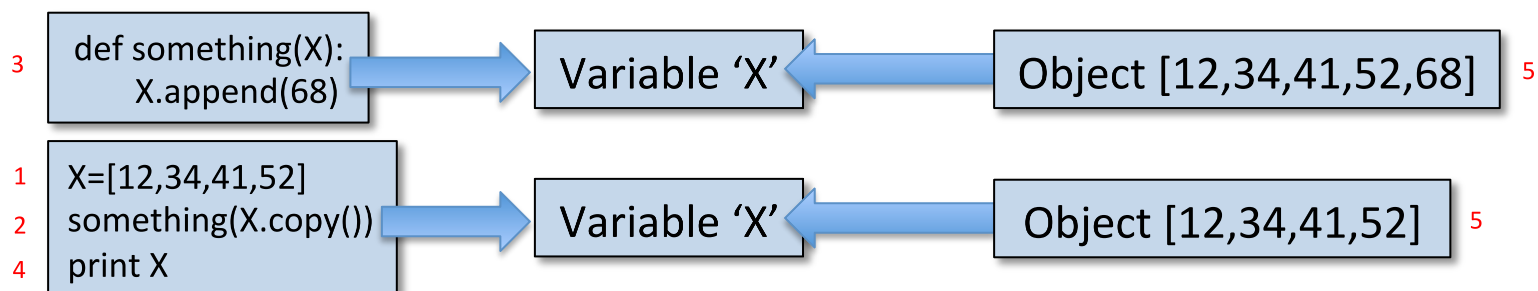
<u>Conclusion:</u> when using pass-by-reference, both the function and the caller use the same exact variable and object / memory location

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



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

- <u>Scripts</u>, 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':

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

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

```
In [12]: whos
Variable   Type    Data/Info
----------------------------
name       str     __loader__

In [13]: run lecture3

In [14]: whos
Variable      Type        Data/Info
-----------------------------------
a             str         Python
b             str          is fun
name          str         __loader__
script_test   function    <function script_test at 0x10595a2f0>
```

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

Language specifics

- Scripts, <u>modules</u>, 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:

```
In [15]: import sys

In [16]: sys.path
Out[16]:
['',
 '/Applications/Python 3.4/pyzo2015a/lib/python34.zip',
 '/Applications/Python 3.4/pyzo2015a/lib/python3.4',
 '/Applications/Python 3.4/pyzo2015a/lib/python3.4/plat-darwin',
 '/Applications/Python 3.4/pyzo2015a/lib/python3.4/lib-dynload',
 '/Applications/Python 3.4/pyzo2015a/lib/python3.4/site-packages',
 '/Applications/Python 3.4/pyzo2015a/lib/python3.4/site-packages/setuptools-12.2-py3.4.egg',
 '/Applications/Python 3.4/pyzo2015a/lib/python3.4/site-packages/IPython/extensions']

In [17]: sys.version
Out[17]: '3.4.2 |Continuum Analytics, Inc.| (default, Oct 21 2014, 17:42:20) \n[GCC 4.2.1 (Apple Inc. build 5577)]'
```

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:

```
In [18]: from sys import path

In [19]: whos
Variable      Type        Data/Info
-----------------------------------
a             str         Python
b             str          is fun
name          str         __loader__
path          list        n=8
script_test   function    <function script_test at 0x10ed922f0>

In [20]: path
Out[20]:
['',
 '/Applications/Python 3.4/pyzo2015a/lib/python34.zip',
 '/Applications/Python 3.4/pyzo2015a/lib/python3.4',
 '/Applications/Python 3.4/pyzo2015a/lib/python3.4/plat-darwin',
 '/Applications/Python 3.4/pyzo2015a/lib/python3.4/lib-dynload',
 '/Applications/Python 3.4/pyzo2015a/lib/python3.4/site-packages',
 '/Applications/Python 3.4/pyzo2015a/lib/python3.4/site-packages/setuptools-12.2-py3.4.egg',
 '/Applications/Python 3.4/pyzo2015a/lib/python3.4/site-packages/IPython/extensions']

In [21]: version
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-21-6e6c2420ff71> in <module>()
----> 1 version

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

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**

```
Python

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:

```python
import numpy as np
a = np.array([[12, 34], [41, 54]])
b = np.arange(0.5,4*np.pi,0.25)
c = np.sin(b)
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:

```python
public_var = 12
_private_var = 34

def public_fun():
    print('This function is public')
def _private_fun():
    print('... now this function is set to be private')

class PublicClass():
    print('This is a public class')
class _PrivateClass():
    print('... now this Class is private')
```

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

```
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:

```python
public_var = 12
_private_var = 34

__all__ = ['public_var','_private_fun']

def public_fun():
    sum_pub_priv = public_var + _private_var
    print('This function is public. The sum is', sum_pub_priv)
def _private_fun():
    print('... now this function is set to be private')

class PublicClass():
    print('This is a public class')
class _PrivateClass():
    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, <u>modules</u>, packages

<u>Caution</u>: star import is <u>NOT</u> 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:

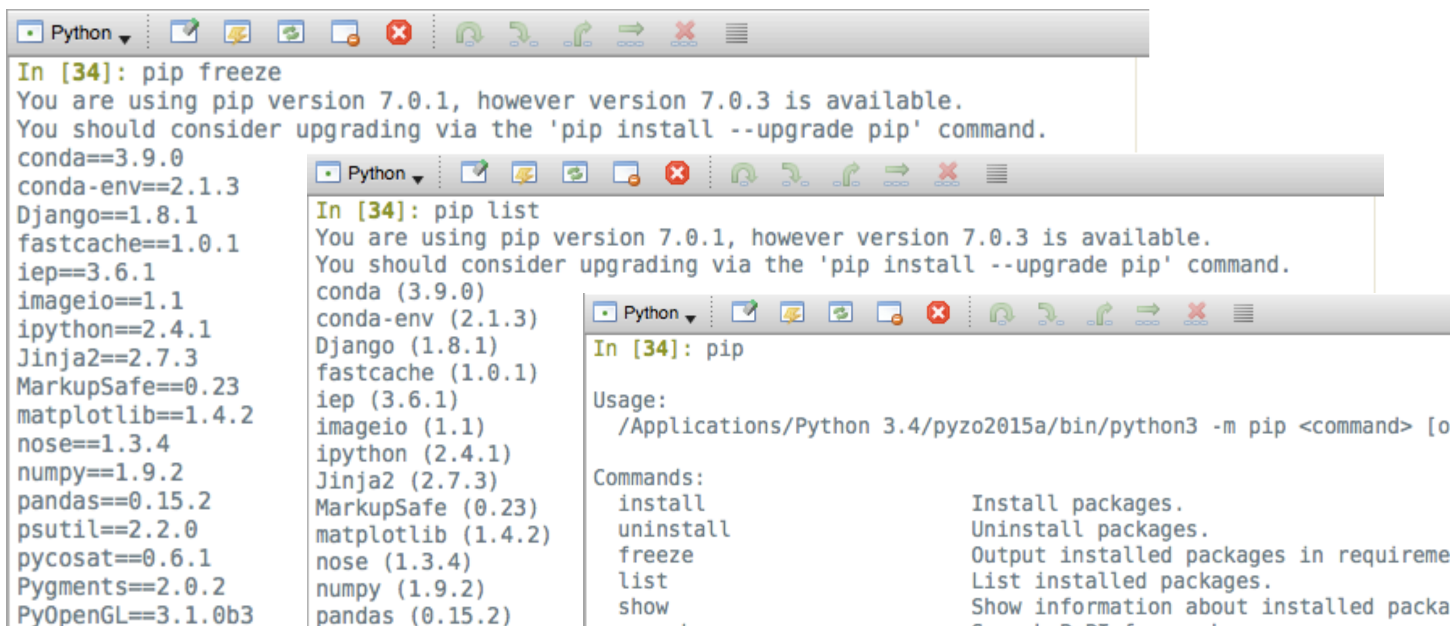
- it decreases readability of the code and it is hard to understand it
- the tab completion is not functioning as it is supposed to – to provide simplicity
- it is impossible to check for different undefined symbols
- there may be problems with visibility of different variables or functions declared 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, <u>packages</u>
  - 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

Language specifics

- I/O interaction with files
    - when working with data it is generally more convenient to read it from a file containing it
    - in order to store some result the user have to be able to write to a file
    - in order to begin reading from or writing to a file, the user has to specify it

Example:

```
file = open('files/lecture3/test.txt', 'r') # opens file for reading
sentences = file.readlines()
print(sentences)
print(len(sentences))
file.close()

file = open('files/lecture3/test.txt', 'w') # opens file for writing
file.write('We will overwrite the previous text \n and go to a new line as well')
file.close()

file = open('files/lecture3/test.txt', 'r') # opens file for reading
sentences = file.readlines()
print(sentences)
print(len(sentences))
file.close()
```

... the code above will result in:

```
['Hello all, I am a text file :)']
1
['We will overwrite the previous text \n', ' and go to a new line as well']
2
```

Language specifics

- I/O interaction with files
  - below are the possible file mode options for file I/O interaction:
    - r – open file to read-only
      - » Note: you can not write to it, but only to read from it
    - w – open the file to write-only
      - » Note: this option creates a new file or overwrites an existing one
    - a – open file to append to it
      - » Note: it does not delete any previous entries
    - r+ - open file to read and write
      - » Note: it works just like the 'w' option, but you can read the file
    - b – open file in binary mode
      - » Note: used for binary files

Language specifics

- Standard library

    - Some of the top standard library modules in Python are:

    - Os – provides a selected list of operating system level functionality
    - Sys – provides access to some variables used by the interpreter
    - Io – deals with I/O functionality for the three main types of I/O: text, binary and raw
    - Math – it gives access to mathematical functions excluding complex numbers (->cmath)
    - Wave – part of Python core installation, provides interface to the WAV format
    - Audioop – consist of useful tools for operating on digital sound sampled data
    - Html – provides an utility to work with the html language
    - Time – provides functions related to time
    - Calendar – provides various calendar capability
    - Daytime – extended way of manipulating date and time

Language specifics

- Exceptions
  - exceptions in Python are raised when the interpreter finds a problem with executing a code
  - they can be used to notify the user that certain state is reached or a condition is met
  - exceptions can pass messages from one part of the code to another
  - there are different types of errors and some of them are shown below

Example:

```
In [35]: a = [12,34,41,54]

In [36]: a[4]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/Users/alex/1.HD/Alex/1.new/Work/3.Berkeley Extension/3. final course material/2.
.py in <module>()
----> 1 a[4]

IndexError: list index out of range

In [37]: a[2]/0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/Users/alex/1.HD/Alex/1.new/Work/3.Berkeley Extension/3. final course material/2.
.py in <module>()
----> 1 a[2]/0

ZeroDivisionError: division by zero
```

Language specifics

- Exceptions

  - exceptions in Python are raised when the interpreter finds a problem with executing a code
  - they can be used to notify the user that certain state is reached or a condition is met
  - exceptions can pass messages from one part of the code to another
  - there are different types of errors and some of them are shown below

Example:

```
In [38]: a.delete
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/Users/alex/1.HD/Alex/1.new/Work/3.Berkeley Extension/3. final course material/2.
.py in <module>()
----> 1 a.delete

AttributeError: 'list' object has no attribute 'delete'

In [39]: a[3] + 'Python'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/Users/alex/1.HD/Alex/1.new/Work/3.Berkeley Extension/3. final course material/2.
.py in <module>()
----> 1 a[3] + 'Python'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Language specifics

- Exceptions
  - in order to handle exceptions, they have to be caught first

Example:

… running the code:

```
while True:
    try:
        a = int(input('Please enter a number: '))
        print('You entered the number ', a)
        print('I will now exit. Good bye!')
        break
    except ValueError:
        print('You entered an invalid number.  Please try again.')
```

… will produce the following result:

```
Please enter a number: q
You entered an invalid number.  Please try again.
Please enter a number: t
You entered an invalid number.  Please try again.
Please enter a number: 5
You entered the number  5
I will now exit. Good bye!
```

Why NumPy?

- Why NumPy?
  - Numpy is the main scientific open-source package for numerical computation in Python
  - Numpy provides:
    - functionality comparable to Matlab,
    - it allows for fast algorithm development and proof-of-concept scientific solutions
    - It provides logic manipulation functionality
    - large set of mathematical functions
    - linear algebra functionality
    - Fast Fourier transform
    - large multidimensional array objects
    - variety of routines for fast operations on arrays
    - different objects, like matrices and masked arrays
    - random simulation
    - sorting
    - statistical operations
    - … and much more

Why NumPy?

- Why NumPy?
  - NumPy's core functionality is the *ndarray*, which stands for *n*-dimensional array
  - NumPy arrays' data structure and standard sequences in Python have some important differences:
    - NumPy array elements must be of the same data type and take the same memory space
    - this gives NumPy the capability to make advanced mathematical calculations possible on large data sets
    - this kind of calculations are executed with higher efficiency and use more concise code as compared to the built-in sequences in Python
    - lists in Python can increase on the fly, while arrays in NumPy are fixed size once created
    - when the size of an ndarray is changed, a new array will be created and the reference (id) to the the original array will be released (lost and deleted)
    - in Python and NumPy, when having arrays of objects, arrays of different sized elements are possible

Why NumPy?

- Why NumPy?
  - The main differences between regular Python objects and NumPy objects are:
    - **Speed** – comparing the results from a simple test on performing addition over a regular Python list and over a NumPy array, reveals that the sum on the latter is faster
    - **Memory efficiency**:
      - NumPy's arrays are more compact than Python lists (example later in slides)
      - a list of lists in Python, would take at 3-5 times more space than a NumPy array using single-precision float type numbers
    - **Functionality** - FFT, convolution, statistics, linear algebra, histograms, etc.
    - **Convenience** – all vector and matrix operations come free with NumPy, while they are efficiently implemented and save unnecessary work

Why NumPy?

- Why NumPy?
  - The main differences between regular Python objects and NumPy objects are:
    - **Speed** – comparing the results from a simple test on performing addition over a regular Python list and over a NumPy array, reveals that the sum on the latter is faster for large calculations

Speed test example:

```
from numpy import arange
import time

N = int(input('Please enter a number: '))

x = arange(N)
y = range(N)

tic = time.clock()
x.sum()
toc = time.clock()
t_numpy = toc - tic

tic = time.clock()
sum(y)
toc = time.clock()
t_list = toc - tic

print("numpy: %.3e sec" % (t_numpy))
print("list:  %.3e sec" % (t_list))
print("diff:  %.3e sec" % (t_list-t_numpy))
print("ratio: list is %.1i times slower than numpy ndarray" % (t_list/t_numpy))
```

Python

```
In [1]: run speed_test_list_ndarray
Please enter a number: 20000
numpy: 7.300e-05 sec
list:  8.330e-04 sec
diff:  7.600e-04 sec
ratio: list is 11 times slower than numpy ndarray
```

Data type objects

- Data type objects
  - there are five basic numerical types in NumPy:
    - bool – booleans
    - int – integers
    - uint – unsigned integers
    - float – floating point
    - complex – 2 double precision numbers
  - all numerical types in NumPy are instances of the dtype object and you can find them like this:

```
In [1]:  import numpy as np
In [2]:  np.<data type>
                 or
In [3]:  dir(np)
```

Data type objects

- Data type objects
  - NymPy supports much larger variety of types than what the standard Python implementation does:

Number type	Data type	Description
Booleans	bool, bool8, bool_	Boolean (True or False) stored as a byte – 8 bits
Integers	byte	compatible: C char – 8 bits
	short	compatible: C short – 16 bits
	int, int0, int_	Default integer type (same as C `long`; normally either `int32` or `int64`) – 64 bits
	longlong	compatible: C long long – 64 bits
	intc	Identical to C `int` – 32 bits
	intp	Integer used for indexing (same as C `size_t`) – 64 bits
	int8	Byte (-128 to 127) – 8 bits
	int16	Integer (-32768 to 32767) – 16 bits
	int32	Integer (-2147483648 to 2147483647) – 32 bits
	int64	Integer (-9223372036854775808 to 9223372036854775807) – 64 bits
Unsigned integers	uint, uint0	Python int compatible, unsigned – 64 bits
	ubyte	compatible: C unsigned char, unsigned – 8 bits
	ushort	compatible: C unsigned short, unsigned – 16 bits
	ulonglong	compatible: C long long, unsigned – 64 bits
	uintp	large enough to fit a pointer – 64 bits
	uintc	compatible: C unsigned int – 32 bits
	uint8	Unsigned integer (0 to 255) – 8 bits
	uint16	Unsigned integer (0 to 65535) – 16 bits
	uint32	Unsigned integer (0 to 4294967295) – 32 bits
	uint64	Unsigned integer (0 to 18446744073709551615) – 64 bits

Data type objects

- Data type objects
  - NymPy supports much larger variety of types than what the standard Python implementation does:

Number type	Data type	Description
Floating-point numbers	half	compatible: C short – 16 bits
	single	compatible: C float – 32 bits
	double	compatible: C double – 64 bits
	longfloat	compatible: C long float – 128 bits
	float_	Shorthand for `float64` – 64 bits
	float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
	float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
	float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
	float128	128 bits
Complex floating-point numbers	csingle	64 bits
	complex, complex_	Shorthand for `complex128` – 128 bits
	complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
	complex128	Complex number, represented by two 64-bit floats (real and imaginary components)
	complex256	two 256 bit floats

  - To check how many bits each type occupies, use one of these notations:
    1) (np.dtype(np.<type>).itemsize)*8
    2) np.<type>().itemsize*8

Data type objects

- Data type objects
  - the difference between signed and unsigned integers and long type variables is:
    - the signed and unsigned types are of the same size
    - the signed can represent equal amount of values around the ‘0’ thus representing equal amount of positive and negative numbers
    - the unsigned can not represent any negative numbers, but can represent double the amount of total positive numbers as compared to the signed type
    - for 32-bit int we have:

      int: –2147483648 to 2147483647

      uint: 0 to 4294967295
    - for 64-bit long we have:

      long: -9223372036854775808 to 9223372036854775807

      ulong: 0 to 18446744073709551615

Data type objects

- Data type objects

  - some of the data types that contain numbers, explicitly specify the bit size of the particular type

  - this is an important thing to know when coding on a 32-bit or 64-bit platforms and a low-level languages are used (C or Fortran)

  - some NumPy data types can be used to:

    - convert python numbers to array scalars (used as functions)

    - convert python sequences of numbers to arrays

    - enter as arguments to the dtype keyword to call various NumPy methods

Data type objects

- Data type objects
  - Examples:
    - convert Python numbers to array scalars (used as functions)

```
In [3]: import numpy as np

In [4]: from sys import getsizeof

In [5]: x=float(2.5)

In [6]: type(x)
Out[6]: float

In [7]: getsizeof(x)
Out[7]: 24

In [8]: x = np.float128(2.5)

In [9]: type(x)
Out[9]: numpy.float128

In [10]: getsizeof(x)
Out[10]: 32
```

… try it in class

Data type objects

- Data type objects
  - Examples:
    - convert Python sequences (lists, tuples, etc.) of numbers to arrays

```
In [11]: a=([12,34,41,52])

In [12]: type(a)
Out[12]: list

In [13]: b=np.int32(a)

In [14]: type(b)
Out[14]: numpy.ndarray

In [15]: getsizeof(a)
Out[15]: 96

In [16]: getsizeof(b)
Out[16]: 80

In [17]: a
Out[17]: [12, 34, 41, 52]

In [18]: b
Out[18]: array([12, 34, 41, 52], dtype=int32)
```

Data type objects

- Data type objects
  - Examples:
    - enter as arguments to the dtype keyword to call various NumPy methods

```
Python
In [19]: c = np.arange(5, dtype=np.float16)

In [20]: type(c)
Out[20]: numpy.ndarray

In [21]: c
Out[21]: array([ 0.,  1.,  2.,  3.,  4.], dtype=float16)

In [22]: getsizeof(c)
Out[22]: 80
```

... try it in class