Programming for Analytics

More data types, functions, modules, packages

S. Kanungo

Tuples

- Tuples are immutable
- Differences from lists
  - No methods
  - Most list operators work
  - However, you can use **`in`** to see if an element exists in the tuple.
- So, why tuples?
  - Tuples are faster
  - Tuples make your code safer
  - Used to format strings

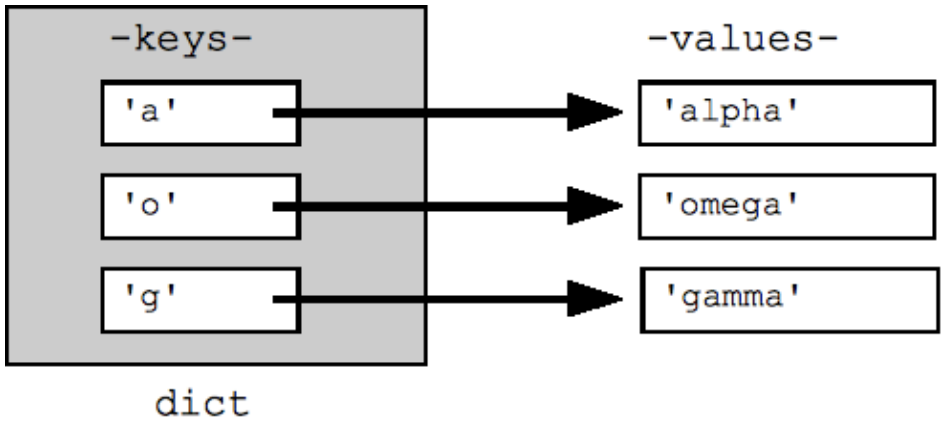
Sets

- Sets are unordered collection of unique elements.
- Set operations are shown below

Operation	Equivalent	Result
len(s)		cardinality of set *s*
x in s		test *x* for membership in *s*
x not in s		test *x* for non-membership in *s*
s.issubset(t)	s <= t	test whether every element in *s* is in *t*
s.issuperset(t)	s >= t	test whether every element in *t* is in *s*
s.union(t)	s | t	new set with elements from both *s* and *t*
s.intersection(t)	s & t	new set with elements common to *s* and *t*
s.difference(t)	s - t	new set with elements in *s* but not in *t*
s.symmetric_difference(t)	s ^ t	new set with elements in either *s* or *t* but not both
s.copy()		new set with a shallow copy of *s*

Python dictionary

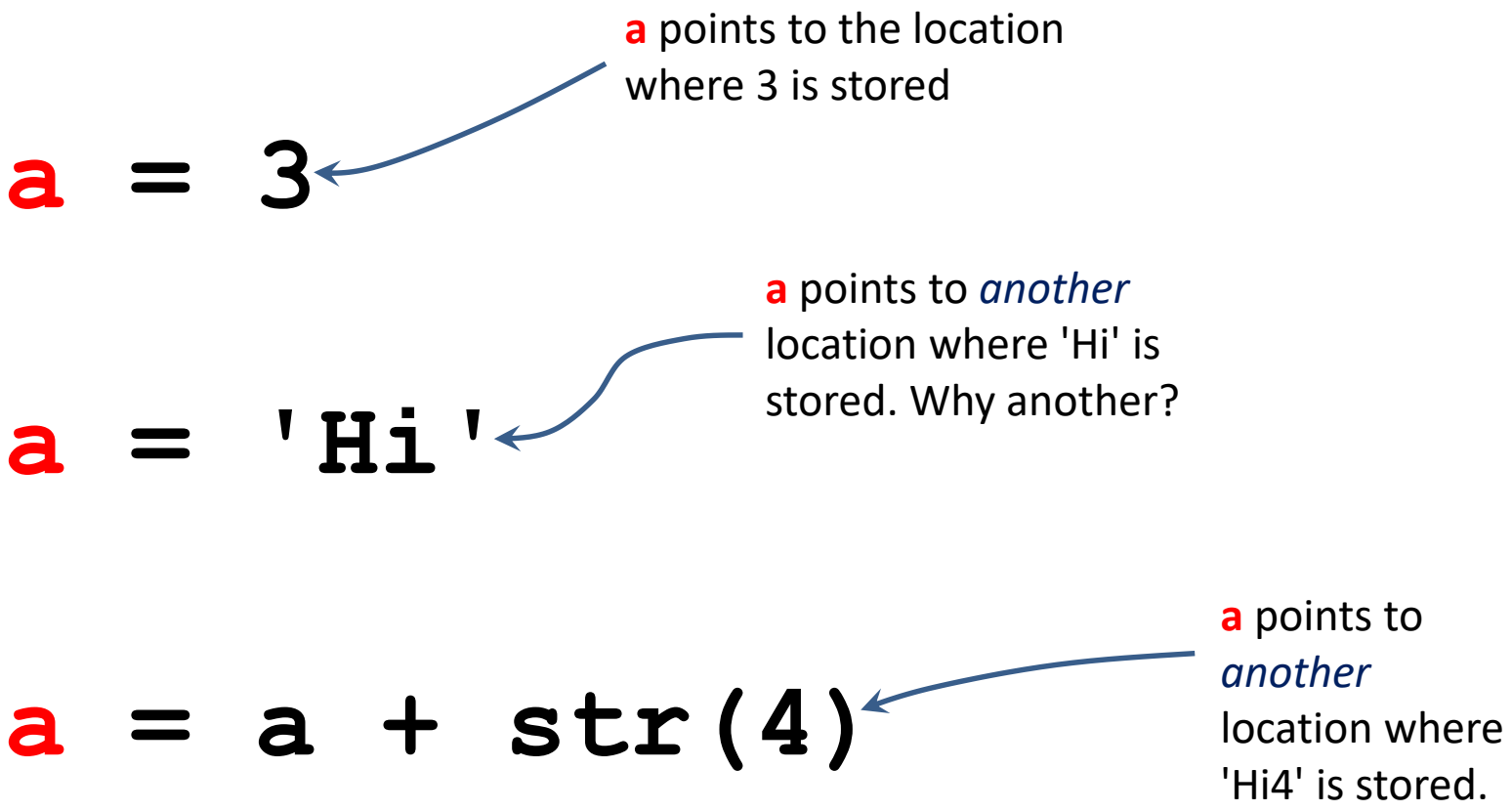
- A dictionary is mutable container type
- It that can store any number of Python objects, including other container types.
- Dictionaries consist of pairs (called items) of keys and their corresponding values



Dictionary exercise

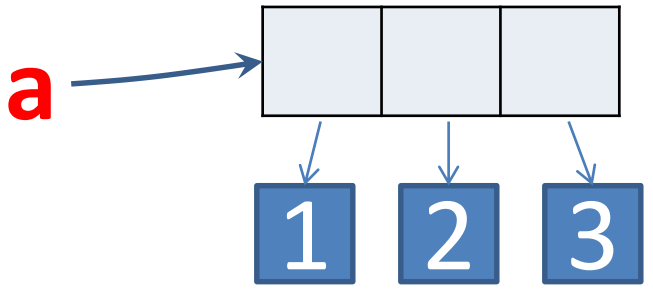
1. Create an empty dictionary and name it `d1`
2. Populate it with the key value pairs: `potato:25` and `tomato:45`
3. Create a list (`mylist`) of 2 tuples: ('apple', 15), ('orange', 45)
4. Create a dictionary `d2` from that tuple (mytuple)
5. Create two lists called `produce ["banana", "lime"]` and `quantity [23,46]`
6. Zip produce and quantity into a list named `pq`
7. Create a dictionary called `d3` from `pq`
8. Assign the value `333` to the key `"banana"` in `d3`
9. Test the membership of a key in a dictionary; specifically, test whether `"tomato"` exists in `d2` and then test whether `"tomato"` exists in `d1`
10. Delete (or remove) a key in a dictionary; specifically, remove `"tomato"` from `d1`
11. Obtain the length (or size) of a dictionary; in this case, that of `d3`
12. Perform an emptiness test on `d1`
13. Increment the value associated with the key `"lime"` by `210` in `d3`.

Mutability

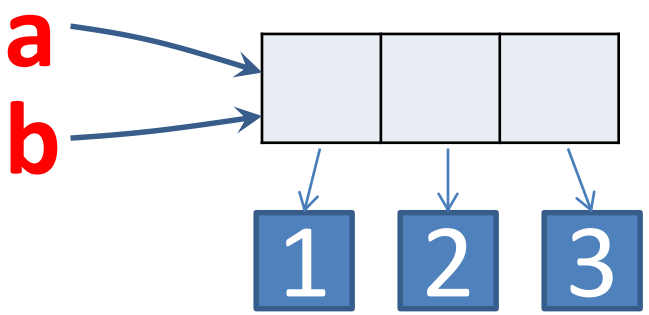

a points to the location where 3 is stored
a = 3
a points to another location where 'Hi' is stored. Why another?
a = 'Hi'
a points to another location where 'Hi4' is stored.
a = a + str(4)

Mutability

```
a = [1, 2, 3]
```




```
b = a
```




```
a.append(8)
b
```

Abstractions for modularity

- **Functions**
  - What a function does
  - Writing function
- **Modules**
  - What module does
  - Calling modules
- **Packages**
  - Why have packages
  - Using a package
- **Classes**
  - What are classes
  - How do we use them

Python functions

- A function is a block of code.
- Functions provide
  - **Modularity** and
  - **code reuse**
- Many built-in functions
- Your functions are user-defined functions

Defining a Function

- Function name
- Separate block
  - Keyword `def`
  - Function name and
  - Parentheses are required
  - Parameters optional

Function exercise # 1

- Define a function called **`names`**
- The function requests a user for her/his name and prints that name preceded by "Hello"
- This function does not have any parameters defined
- This example will also show how to use docstrings
- Call to execute that function

Function exercise # 2

- Define a function and call it **`myadd`**
- The function adds and prints two numbers after checking that both are legitimate numbers; if the number test fails the function prints "Wrong input"
- This function has two parameters
- Call to execute that function

Python Modules

- A module is a Python program
- Underlying idea is reusability
- Flexibility exists to create modules
- `import` to use module
- What does importing accomplish?
- Python modules are an example of the abstraction layers available in Python

Module usage guidelines

```
from modu import *
x = sqrt(4)   # Is sqrt part of modu?
              # builtin? Defined above?
```

Very bad

```
from modu import sqrt
x = sqrt(4)   # sqrt may be part
              # of modu, if not
              # redefined in between
```

Better

```
import modu
x = modu.sqrt(4) # sqrt is visibly
                 # part of modu's
                 # namespace
```

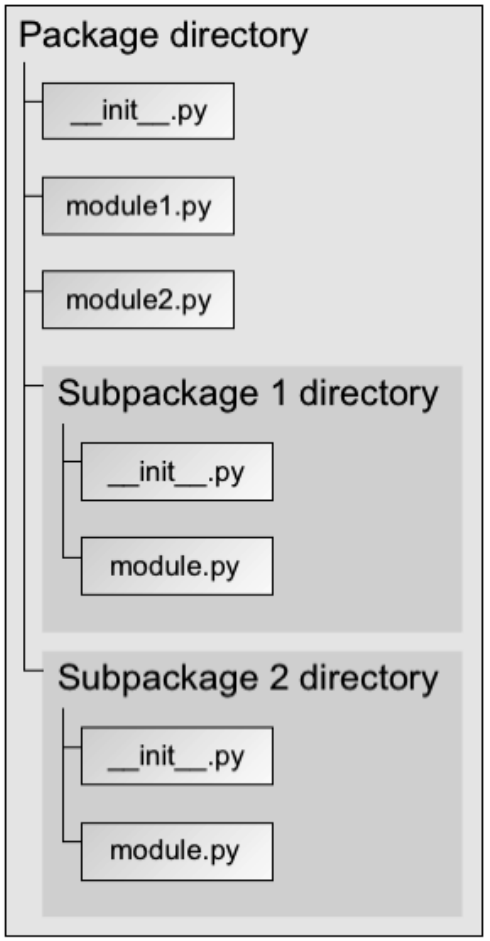
Best

Python classes

- classes similar to modules
- "class" is an object-oriented term
- "class" as template
- Difference between a class and an object
- Class is to cookie cutter as objects is to cookies
- Class made up of attributes and methods

Python Packages

- A package as a directory of module(s)
- Packages used to structure Python's module namespace
- Uses the dot notation – typically package.module.function
- A package is a directory containing Python files



Introducing list comprehensions

- "list comprehensions" used to make lists
- Natural and easy way to create lists
- "comprehension" is a math term to define sets in terms of other sets. e.g.,

$$S = \{x^2 : x \text{ in } \{0 \ldots 9\}\}$$
$$V = (1, 2, 4, 8, \ldots, 2^{12})$$
$$M = \{x \mid x \text{ in } S \text{ and } x \text{ even}\}$$

list comprehensions – syntax

- "list comprehensions" is available as a Python feature
- What a list comprehension does
  - Generate a new list by applying a function to every member of an original list
- Typical Python notation

```
[expression for name in list if condition]
```

- Three keywords (for, in, and if) can be used in the syntax of forms of list comprehensions.

Examples

With strings

```
mynames = ['Tom', 'Dick', 'Harry']
x = ['Hello ' + person for person in mynames]
```

```
>>> mynames = ['Tom', 'Dick', 'Harry']
>>> x = ['Hello ' + person for person in mynames]
>>> for greeting in x:
...     print (greeting)
...
Hello Tom
Hello Dick
Hello Harry
>>>
```

Examples

With built-in functions

```
myfloats = [1.2345, 2.3456, 3,4567]
x = [round(mynum,2) for mynum in myfloats]
```

```
>>> myfloats = [1.2345, 2.3456, 3,4567]
>>> x = [round(mynum,2) for mynum in myfloats]
>>> for anumber in x:
...     print (anumber)
...
1.23
2.35
3.0
4567.0
>>>
```

Examples

With user defined functions

```
def mysqplusone(a):
    return a**2+1

mynums = [1,2,3]
x = [mysqplusone(i) for i in mynums]
```

```
>>> def mysqplusone(a):
...     return a**2+1
...
>>> mynums = [1,2,3]
>>> x = [mysqplusone(i) for i in mynums]
>>> print (x)
[2, 5, 10]
>>>
```

Examples

With more than one list

```
a = [1,2,3]
b = [3,4,5]
x = [c+d for c in a for d in b]
```

```
>>> a = [1,2,3]
>>> b = [3,4,5]
>>> x = [c+d for c in a for d in b]
>>> x
[4, 5, 6, 5, 6, 7, 6, 7, 8]
>>>
```

Using conditions

Selective string manipulation

```
mynames = ['Tom', 'Dick', 'Harry']
x = ['Hello ' + person for person in mynames if len(person) > 3]
```

```
>>> mynames = ['Tom', 'Dick', 'Harry']
>>> x = ['Hello ' + person for person in mynames \
          if len(person) > 3]
>>> for selectedgreetings in x:
...     print (selectedgreetings)
...
Hello Dick
Hello Harry
>>>
```

Using conditions

Selective numeric manipulation

```
myfloats = [1.2345, 2.3456, 3,4567]
x = [round(mynum,2) for mynum in myfloats
                    if int(mynum)%2==0]
```

```
>>> myfloats = [1.2345, 2.3456, 3,4567]
>>> x = [round(mynum,2) for mynum in myfloats
                        if int(mynum)%2==0]
>>> print (x)
[2.35]
>>>
```

Using conditions

Selectively creating tuples

```
a = [1,2,3]
b = [3,4,5]
x = [(c,d) for c in a for d in b
           if c%2 == 0 and d%2 == 0]
```

```
>>> a = [1,2,3]
>>> b = [3,4,5]
>>> x = [(c,d) for c in a for d in b
                if c%2 == 0 and d%2 == 0]
>>> x
[(2, 4)]
>>>
```

List comprehension exercise

- Let a = [[1,2],[3,4], [5,6]]. Write a list comprehension that will give [[1,4],[9,16],[25,36]] – essentially squaring each element
- Hint you need to generate a list of lists

Function arguments and parameters

- Parameters
- Arguments

```
def greet2(uname):
    """
    This function expects a user name (string) and then
    greets the user
    Input: user name (string)
    Output: Greeting followed by the user name
    """
    print ("Hello " + uname)
```

```
>>> yourname = raw_input("Please enter your name: ")
Please enter your name: Raj
>>> greet2(yourname)
Hello Raj
>>>
```

Passing by reference

- All parameters are **passed by reference**.
  - Implies that if you change what a parameter refers to inside a function that change is reflected back in the calling function.
- Some parameters behave as if they are **passed by value**
- Class example
  - What do you expect to happen?
  - Replace the list with an int. What happens? Why?
- Bottom line: Be careful

The `return` statement

- The return statement does two things,
  - It signals the end of processing a function.
  - It enables the function to send data back to the calling function

Types of arguments

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required arguments

- Required arguments are the arguments passed to a function in correct positional order

```
>>> def greet2(uname):
...     print ("Hello " + uname)
...
>>> greet2("Raj")
Hello Raj
```

```
>>> greet2()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    greet2()
TypeError: greet2() takes exactly 1 argument (0 given)
```

Keyword arguments

- allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters

```
>>> def calculatebmi(height, weight, name):
...     bmi = float(weight) / height**2
...     print ("%s's BMI is %5.2f" % (name, bmi))
...
```

```
>>> calculatebmi(1.65, 100, "Raj")
Raj's BMI is 36.73
>>> calculatebmi(weight=75, name='Raj', height= 1.55)
Raj's BMI is 31.22
>>>
```

Default arguments

- A default argument assumes a default value if a value is not provided in the function call

```
>>> def demandperday(annualdemand, daysperyear=250):
 ... return round(annualdemand / daysperyear,0)
 ...
```

```
>>> print "Demand per day is %6.2f" % (demandperday(1123, 350))
Demand per day is 3.00
```

```
>>> print "Demand per day is %6.2f" % (demandperday(1123))
Demand per day is 4.00
>>>
```

Variable-length arguments

```
>>> def printinfo( *vartuple ):
...     print "Output is: "
...     for var in vartuple:
...         print (var)
...     return
...
>>> printinfo(1)
Output is:
1
>>> printinfo()
Output is:
>>> printinfo(2,4,6)
Output is:
2
4
6
>>>
```

Open multiple files

- Let's say we need to read and collate data from an unknown number of files. The required file is `sampledata.txt` and then we have a variable set of files …

```
>>> def collatefiledata(file1, *otherfiles):
...     myList = []
...     with open(file1) as f:
...         for line in f:
...             myList.append(int(line))
...
...     for everyfile in otherfiles:
...         with open(everyfile) as f:
...             for line in f:
...                 myList.append(int(line))
...
...     return myList
...
>>>
```

Dictionaries as variable length args

```
def print_my_args(**mydict):
    print mydict
```

```
>>> print_my_args(a = 2, b = [1,2,3])
{'a': 2, 'b': [1, 2, 3]}
```

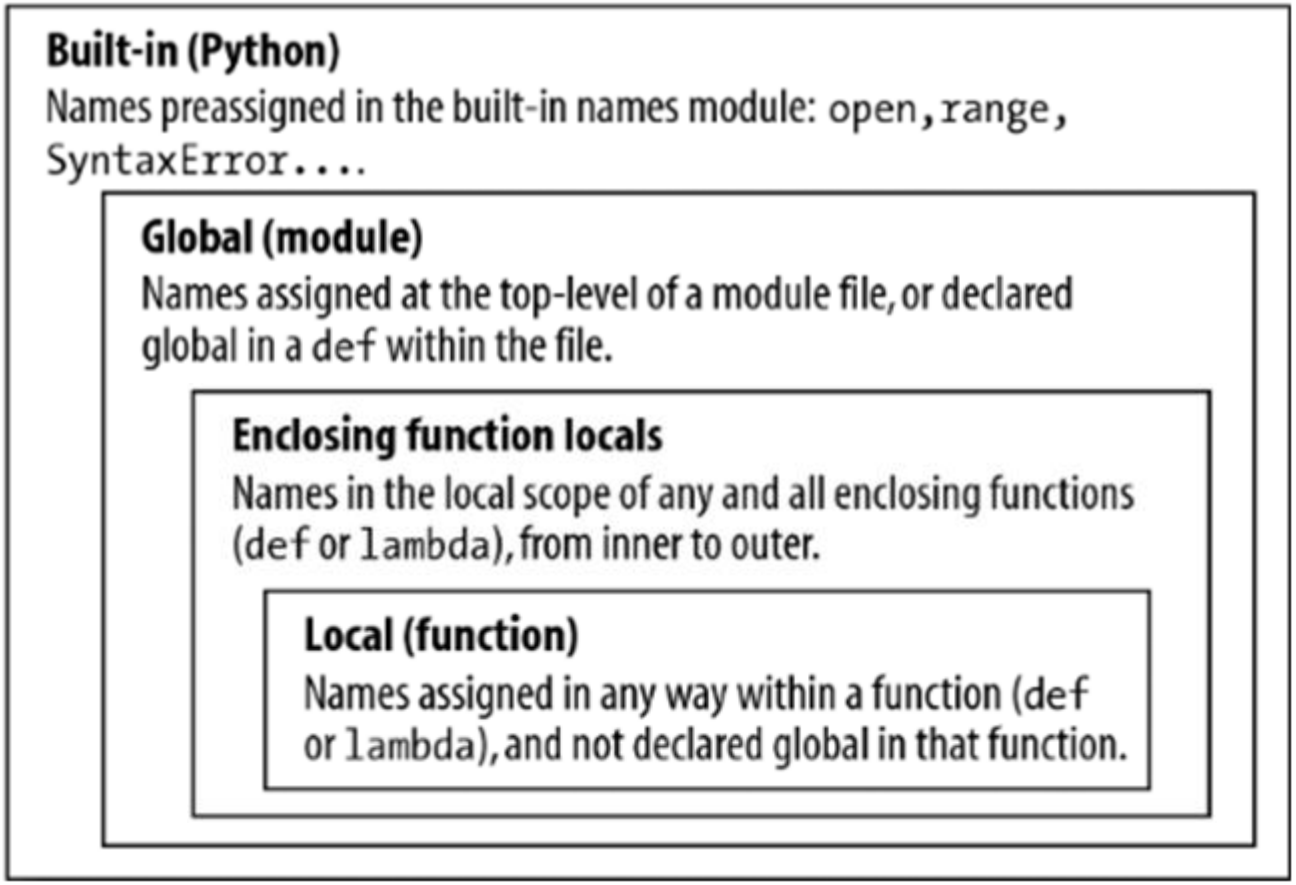
```
>>> print_my_args(a = 2, b = {1:"a","b":3})
{'a': 2, 'b': {1: 'a', 'b': 3}}
```

Anonymous / Lambda functions

- They are small one-line functions
- anonymous since they are not declared in the standard manner with the `def` keyword.

```
>>> diff = lambda arg1, arg2: arg1 - arg2;
>>> print diff(3,5)
-2
>>> diff(5,2)
3
>>>
```

Python scope rules



Python scope example

```
def scope_rules():
    print "---- Inside scope_rules ---- "
    global myconstant
    myconstant = 3.14
    circumference = 20
    anotherconstant = 34

    def area(r):
        print ("---- Inside area ---- ")
        print ("Area of circle = ", myconstant*r*r)
        circumference = 2*myconstant*r
        print ("Circumference of circle = ", circumference)
        print ("---- Exiting area ---- ", anotherconstant)

    area(3)
    print ("Circumference of circle = ", circumference)
    print ("---- Exiting scope_rules ---- ")
```

```
circumference = 36
print "Before calling scope_rules"
print "Circumference of circle = ", circumference
scope_rules()
print "After calling scope_rules"
print "Circumference of circle = ", circumference
```

Function exercise

- Read data from a text file (datafile.txt) into a list. Assume it contains one number in every line.
- Compute the mean and standard deviation for that list of numbers
- Communicate the following to the user:
  - The sample size
  - The mean of the sample
  - The standard deviation around the mean

To be done in class

- Start with one program that contains everything (name it `oneprogram.py`)
- Write the same program using functions (name it `withfunctions.py`)
- Change the `withfunctions.py` to read files using a GUI based interface (name it `withfunctionswith gui.py`)
- The objective of this exercise is to demonstrate the value of modularity

How to make and use modules

- The objective of this exercise is to show how to create and use modules.
- The idea is the same: modularity and clean code
- Create one or more modules with the functionality – think about what would make sense
- Write a main program to use these modules

From modules to packages

- Modules reflect the Unix philosophy:
  - Do one thing, do it well.
- Modules are often organized in packages.
- A package is a structured collection of modules that have the same purpose.
- One example of a package is `matplotlib`.

Module location

- Modules are mainly stored in files that are searched:
  - in your current working directory,
  - in PYTHONHOME, where Python has been installed,
  - in a path, i.e a colon (':') separated list of file paths, stored in the environment variable PYTHONPATH. You can check this path through the `sys.path` variable.
  - Try this sequence of commands

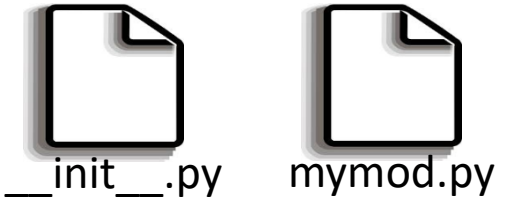
```
import sys
print sys.path
dir(sys)
```

Package

- A package is a collection of Python modules
- A module is a single Python file and a package is a directory
- It is a directory of Python modules containing an additional `__init__.py` file
- The `__init__.py` file distinguishes a package from a directory that just happens to contain a bunch of Python scripts.

Simple example


```
from modtest import mymod
print (mymod.myadd(2,3))
```




```
def myadd(a,b):
    return a+b

def mysub(a,b):
    return a-b
```

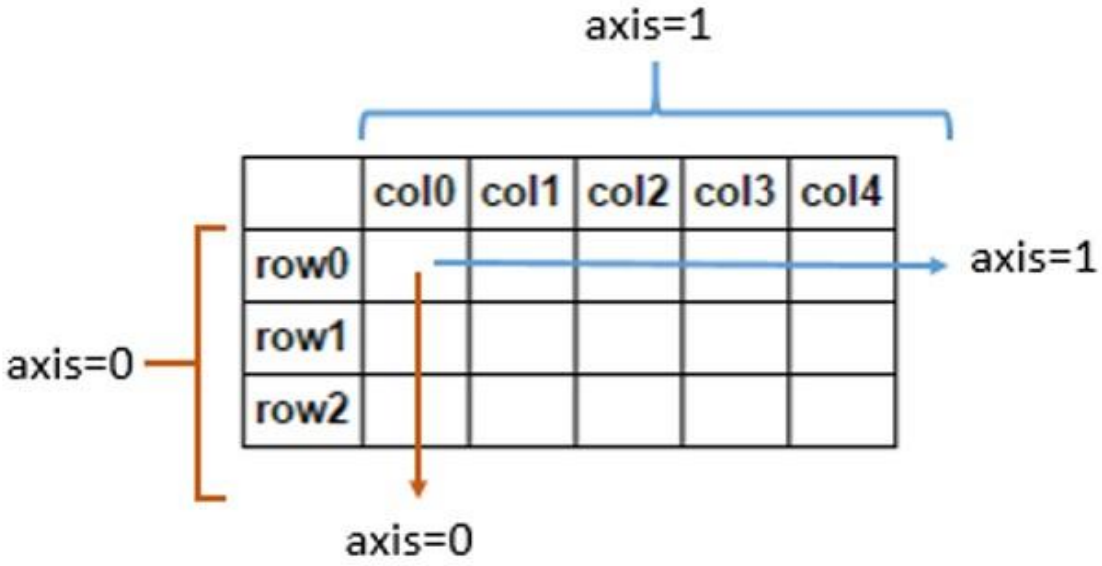
```
__all__ = ['modtest']
import modtest
```

Argparse

- Read up on this and we will do an exercise in class
- The official documentation for argparse is pretty good: https://docs.python.org/3/howto/argparse.html

Pandas dataframe

- Spreadsheet like data object
- Many ways to create it
- Dictionaries are one option



Create dataframe from dictionary

```
import pandas as pd

mydata = {
    'state': [],
    'year':[],
    'pop':[]
}
mydata = {
    'state': ['FL', 'FL', 'GA', 'GA', 'GA'],
    'year':[2010, 2011, 2008, 2010, 2011],
    'pop':[18.8, 19.1, 9.7, 9.8, 9.9]
}
mydf = pd.DataFrame(mydata)
print mydf
```

Getting data into a dataframe

- pandas supports several ways to handle data loading
- Text file data
  - `read_csv`
  - `read_table`
- Structured data (JSON, XML, HTML)
  - existing libraries work
- Excel (depends upon `xlrd` and `openpyxl` packages)
- Database
  - pandas.io.sql module (read_frame)