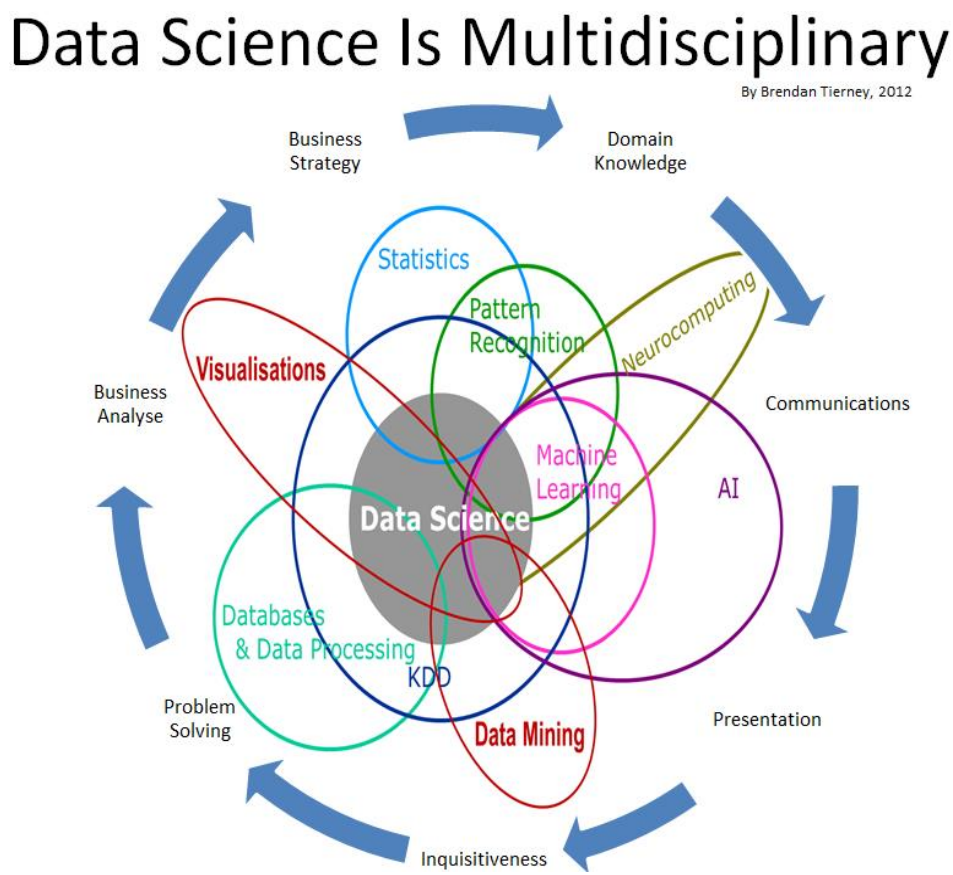
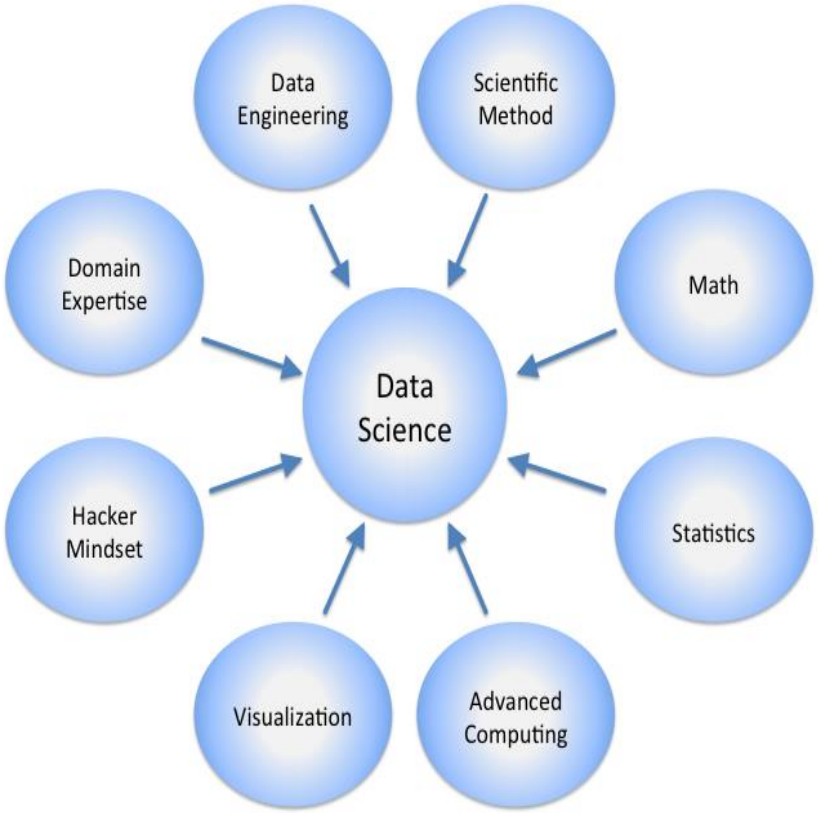
PYTHON PROGRAMMING 7

Anasua Sarkar
Computer Science & Engineering Department
Jadavpur University
anasua.sarkar@jadavpuruniversity.in

Data Science – A Definition

Data Science is the science which uses computer science, statistics and machine learning, visualization and human-computer interactions to collect, clean, integrate, analyze, visualize, interact with data to create data products.





Python : Introduction

- Most recent popular (scripting/extension) language
  - although origin ~1991
- heritage: teaching language (ABC)
  - Tcl: shell
  - perl: string (regex) processing
- object-oriented
- It includes modules for creating graphical user interfaces, connecting to relational databases, generating pseudorandom numbers, arithmetic with arbitrary-precision decimals, manipulating regular expressions, and unit testing.
- Large organizations that use Python include Wikipedia, Google, Yahoo!, CERN, NASA, Facebook, Amazon, Instagram and Spotify. The social news networking site Reddit is written entirely in Python.

What's Python?

- Python is a general-purpose, interpreted high-level programming language.
- Its syntax is clear and emphasize readability.
- Python has a large and comprehensive standard library.
- Python supports multiple programming paradigms, primarily but not limited to object-oriented, imperative and, to a lesser extent, functional programming styles.
- It features a fully dynamic type system and automatic memory management

@Yang Li

Python structure

- modules: Python source files
  - import, top-level via from, reload
- statements
  - control flow
  - create objects
  - indentation matters – instead of {}
- objects
  - everything is an object
  - automatically reclaimed when no longer needed

Comprehensions

- https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Comprehensions.html
- https://realpython.com/list-comprehension-python/

Collections

- https://docs.python.org/3/library/collections.html

Lambda forms

- anonymous functions
- may not work in older versions

```
def make_incrementor(n):
  return lambda x: x + n

f = make_incrementor(42)
f(0)
f(1)
```

Functional programming tools

- filter(*function, sequence*)
  - filter does the work of a list-comprehension if

```
def f(x): return x%2 != 0 and x%3 == 0
filter(f, range(2,25))
```

```
def is_even(x):
    """True if x is even, False if x is odd"""
    return x % 2 == 0
```

```
x_evens = [x for x in xs if is_even(x)]
x_evens = filter(is_even, xs)
list_evener = partial(filter, is_even)
x_evens = list_evener(xs)
```

```
# [2, 4]
# same as above
# *function* that filters a list
# again [2, 4]
```

- map(*function, sequence*)
  - call function for each item
  - return list of return values

```
def multiply(x, y): return x * y

products = map(multiply, [1, 2], [4, 5]) # [1 * 4, 2 * 5] = [4, 10]
```

- reduce(*function, sequence*)
  - return a single value
  - call binary function on the first two items
  - then that result with the third and so on
  - iterate

```
x_product = reduce(multiply, xs)
list_product = partial(reduce, multiply)
x_product = list_product(xs)
```

```
# = 1 * 2 * 3 * 4 = 24
# *function* that reduces a list
# again = 24
```

File I/O

- open() returns a *file object*, and is most commonly used with two arguments: open(filename, mode).

```
f = file("foo", "r")
line = f.readline()
print line,
f.close()
# Can use sys.stdin as input;
# Can use sys.stdout as output.
```

```
>>> with open('workfile') as f:
...     read_data = f.read()
>>> f.closed
True
```

- *mode* can be 'r' when the file will only be read, 'w' for only writing (an existing file with the same name will be erased), and 'a' opens the file for appending; any data written to the file is automatically added to the end. 'r+' opens the file for both reading and writing.
- Normally, files are opened in *text mode*. 'b' appended to the mode opens the file in *binary mode.*

Files

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

```
list(f)

f.readlines()
```

- Creating file object
  - Syntax: file_object = open(filename, mode)
    - `Input = open("d:\inventory.dat", "r")`
    - `Output = open("d:\report.dat", "w")`
- Manual close
  - Syntax: close(file_object)
    - `close(input)`
- Reading an entire file
  - Syntax: string = file_object.read()
    - `content = input.read()`
- To read a file's contents, call f.read(size), which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode).
  - Syntax: list_of_strings = file_object.readlines()
    - `lines = input.readline()`
    - It is only omitted on the last line of the file if the file doesn't end in a newline.

Files

```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)     # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

- Reading one line at time
  - Syntax: list_of_strings = file_object.readline()
    - `line = input.readline()`
- Writing a string
  - Syntax: file_object.write(string)

`output.write("Price is %(total)d" % vars())`

- Writing a list of strings
  - Syntax: file_object.writelines(list_of_string)
    - `output.write(price_list)`
- This is very simple!
  - Compare it with `java.io`
- f.tell() returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.
- To change the file object's position, use f.seek(offset, from_what).

Saving structured data with json

- JSON (JavaScript Object Notation)
- The standard module called json can take Python data hierarchies, and convert them to string representations; this process is called *serializing*.
- Reconstructing the data from the string representation is called *deserializing*.
- Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

- Another variant of the dumps() function, called dump(), simply serializes the object to a *text file*. So if f is a *text file* object opened for writing,

```
json.dump(x, f)
```

- To decode the object again, if f is a *text file* object which has been opened for reading

```
x = json.load(f)
```

Files: Input

input = open('data', 'r')	Open the file for input
S = input.read()	Read whole file into one String
S = input.read(N)	Reads N bytes (N >= 1)
L = input.readlines()	Returns a list of line strings

Files: Output

output = open(‘data’, ‘w’)	Open the file for writing
output.write(S)	Writes the string S to file
output.writelines(L)	Writes each of the strings in list L to file
output.close()	Manual close

`open()` and `file()`

- These are identical:

```
f = open(filename, "r")
f = file(filename, "r")
```

- The **`open()`** version is older
- The **`file()`** version is the recommended way to open a file now
  - uses object constructor syntax (next lecture)

Collections

- https://docs.python.org/3/library/collections.html

Control flow: if

```
x = int(raw_input("Please enter #:"))
if x < 0:
   x = 0
   print 'Negative changed to zero'
elif x == 0:
   print 'Zero'
elif x == 1:
   print 'Single'
else:
   print 'More'
```

- no case statement

Control flow: for

```
a = ['cat', 'window', 'defenestrate']
for x in a:
  print x, len(x)
```

- no arithmetic progression, but
  - `range(10)` → `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
  -
```
for i in range(len(a)):
    print i, a[i]
```
- do not modify the sequence being iterated over

Loops: break, continue, else

- `break` and `continue` like C
- `else` after loop exhaustion

```
for n in range(2,10):
  for x in range(2,n):
    if n % x == 0:
      print n, 'equals', x, '*', n/x
      break
  else:
    # loop fell through without finding a factor
    print n, 'is prime'
```

Control flow (6)

- Common **while** loop idiom:

```
f = open(filename, "r")
while True:
    line = f.readline()
    if not line:
        break
    # do something with line
```

Do nothing

- pass does nothing
- syntactic filler

```
while 1:
    pass
```

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

- This is commonly used for creating minimal classes

```
>>> class MyEmptyClass:
...     pass
...
```

- Another place pass can be used is as a place-holder for a function or conditional body.

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

Looping Techniques

- When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

- When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

- To loop over two or more sequences at the same time, the entries can be paired with the zip() function.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

More looping techniques

- To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed() function.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

- To loop over a sequence in sorted order, use the sorted() function which returns a new sorted list while leaving the source unaltered.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

Defining functions

```
def foo(x):
    y = 10 * x + 2
    return y
```

- Def keyword must be followed by the function name and the parenthesized list of formal parameters.
- All variables are local unless specified as **`global`**
- Arguments passed by value
- Arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object).
- When a function calls another function, a new local symbol table is created for that call.
- The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*.

Executing functions

```
def foo(x):
    y = 10 * x + 2
    return y
print foo(10)  # 102
```

- The *execution* of a function introduces a new symbol table used for the local variables of the function.
- More precisely, all variable assignments in a function store the value in the local symbol table;
- whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names.
- Thus, global variables cannot be directly assigned a value within a function (unless named in a global statement), although they may be referenced.

Defining functions

```
def fib(n):
  """Print a Fibonacci series up to n."""
  a, b = 0, 1
  while b < n:
    print b,
    a, b = b, a+b

>>> fib(2000)
```

- First line is *docstring*
- first look for variables in local, then global
- need global to assign global variables

```
>>> fib(0)
>>> print(fib(0))
None
```

- In fact, even functions without a return statement do return a value, albeit a rather boring one. This value is called None (it's a built-in name).

Functions: default argument values

```
def ask_ok(prompt, retries=4, complaint='Yes or no,
 please!'):
  while 1:
    ok = raw_input(prompt)
    if ok in ('y', 'ye', 'yes'): return 1
    if ok in ('n', 'no'): return 0
    retries = retries - 1
    if retries < 0: raise IOError, 'refusenik error'
    print complaint
```

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

```
[1]
[1, 2]
[1, 2, 3]
```

```
>>> ask_ok('Really?')
```

- giving only the mandatory argument: ask_ok('Do you really want to quit?')
- giving one of the optional arguments: ask_ok('OK to overwrite the file?', 2)
- or even giving all arguments: ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')

Keyword arguments

- last arguments can be given as keywords

```
def parrot(voltage, state='a stiff', action='voom',
  type='Norwegian blue'):
   print "-- This parrot wouldn't", action,
   print "if you put", voltage, "Volts through it."
   print "Lovely plumage, the ", type
   print "-- It's", state, "!"


parrot(1000)
parrot(action='VOOOM', voltage=100000)
```

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Arbitrary Number of Arguments

- These arguments will be wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

Unpacking Argument Lists

- The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments.
- If they are not available separately, write the function call with the *-operator to unpack the arguments out of a list or tuple.

```
>>> list(range(3, 6))            # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))            # call with arguments unpacked from a list
[3, 4, 5]
```

- In the same fashion, dictionaries can deliver keyword arguments with the **-operator.

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

Lambda forms

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

- anonymous functions
- may not work in older versions
- Small anonymous functions can be created with the lambda keyword. This function returns the sum of its two arguments: lambda a, b: a+b. Lambda functions can be used wherever function objects are required.
- They are syntactically restricted to a single expression.
- Like nested function definitions, lambda functions can reference variables from the containing scope.
- Another use is to pass a small function as an argument.

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Function Annotations

- Function annotations are completely optional metadata information about the types used by user-defined functions.

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

Command-line arguments

```
import sys
print len(sys.argv) # NOT argc
# Print all arguments:
print sys.argv
# Print all arguments but the program
# or module name:
print sys.argv[1:]  # "array slice"
```

Modules

- A Python module is the Python source file, which can consist of statements, classes, functions, and variables.
  - import module1, module2, module3
  - import module_name as new_name
- import module:

```
import fibo
```

- Use modules via "name space":

```
>>> fibo.fib(1000)
>>> fibo.__name__
'fibo'
```

- can give it a local name:

```
>>> fib = fibo.fib
>>> fib(500)
```

  - `from module1 import sum1`
  - from module-name import *

```
def sum1(a,b):
 c = a+b
 return c

def mul1(a,b):
 c = a*b
 return c
```

module1.py

```
import module1
x = 12
y = 34
print "Sum is ", module1.sum1(x,y)
print "Multiple is ", module1.mul1(x,y)
```

Modules

- Access other code by importing modules

```
import math
print math.sqrt(2.0)
```

- or:

```
from math import sqrt
print sqrt(2.0)
```

- **Python offers the built-in function dir(), which can be used to identify the functions.**
- import module1
- print dir(module1)

Modules

- or:

```
from math import *
print sqrt(2.0)
```

- Can import multiple modules on one line:

```
import sys, string, math
```

- Only one "**`from x import y`**" per line

`import`
: Lets a client (importer) fetch a module as a whole

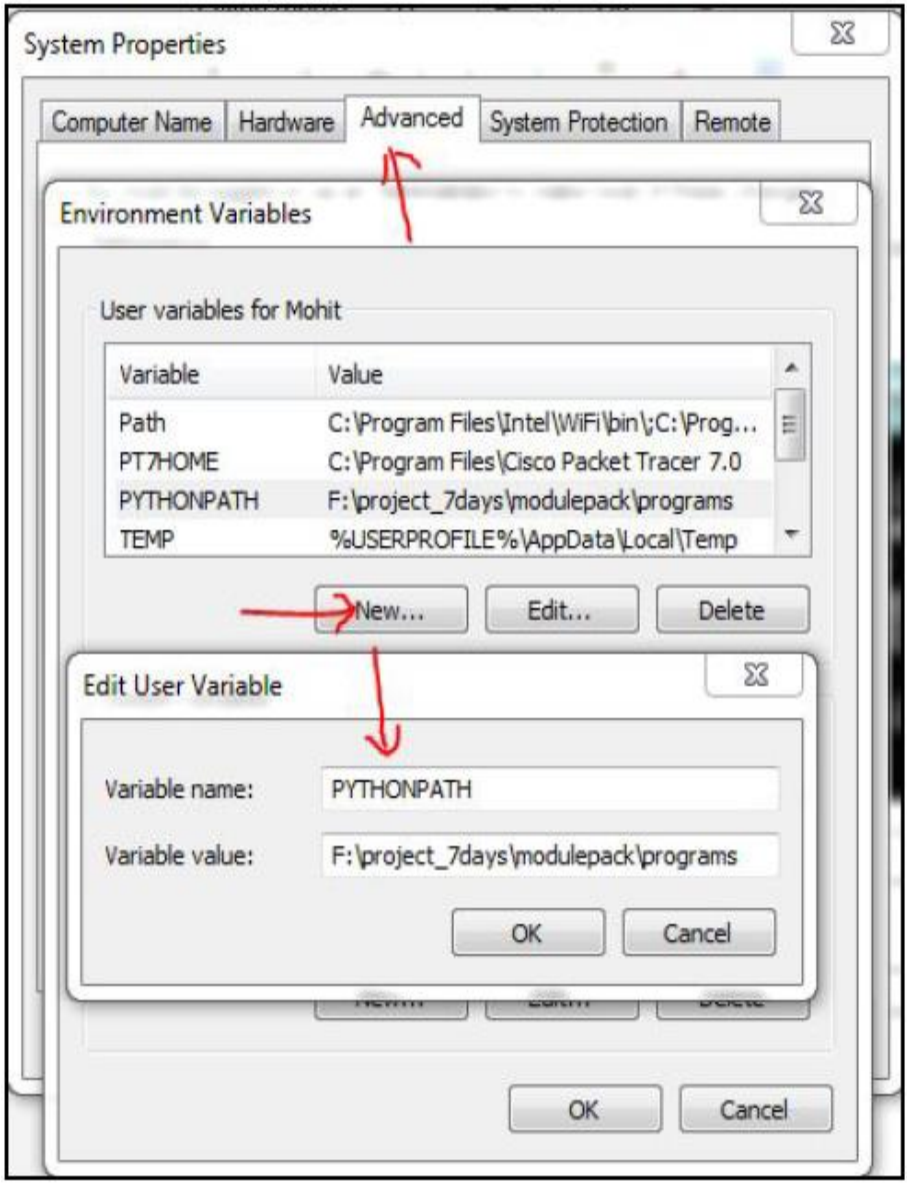
`from`
: Allows clients to fetch particular names from a module

`imp.reload`
: Provides a way to reload a module's code without stopping Python

Module search path

- current directory
- list of directories specified in PYTHONPATH environment variable
- uses installation-default if not defined, e.g., .:/usr/local/lib/python
- uses sys.path

```
>>> import sys
>>> sys.path
['',    'C:\\PROGRA~1\\Python2.2',
 'C:\\Program
 Files\\Python2.2\\DLLs',
 'C:\\Program
 Files\\Python2.2\\lib',
 'C:\\Program
 Files\\Python2.2\\lib\\lib-tk',
 'C:\\Program  Files\\Python2.2',
 'C:\\Program
 Files\\Python2.2\\lib\\site-
 packages']
```



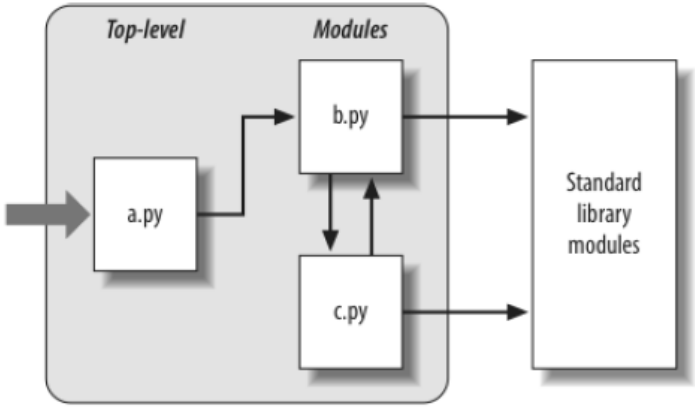
Python package

- collection of modules
- package is a directory that contains Python modules and one additional file: __init__.py.
- The __init__.py file can be an empty file, but it can also be used to
- import the module.

```
import sys
sys.path.append("F:project_7daysmodulepackprograms")
from sound_conversion import rectomp3
from sound_conversion.rectowma import rec2wma
print rectomp3.rec2mp3()
print rec2wma()
```

Modules

- Modules are functions and variables defined in separate files
- Items are imported using from or import
  - from module import function
  - function()
  - import module
  - module.function()
- Modules are namespaces
  - Can be used to organize variable na
    - atom.position = atom.position - molecule.posit
    - After finding a source code file that matches an import statement by traversing the module search path, Python next compiles it to byte code, if necessary.
    - it finds a .pyc byte code file that is not older than the corresponding .py source file, it skips the source-to–byte code compile step.
    - In addition, if Python finds only a byte code file on the search path and no source, it simply loads the byte code directly.



name, _main_

- In effect, a module's __name__ variable serves as a usage mode flag, allowing its code to be leveraged as both an importable library and a top-level script.

Here's another module-related trick that lets you both import a file as a module and run it as a standalone program. Each module has a built-in attribute called __name__, which Python sets automatically as follows:

- If the file is being run as a top-level program file, __name__ is set to the string "__main__" when it starts.
- If the file is being imported instead, __name__ is set to the module's name as known by its clients.

```
def tester():
    print("It's Christmas in Heaven...")

if __name__ == '__main__':          # Only when run
    tester()                        # Not when imported
```

```
% python
>>> import runme
>>> runme.tester()
It's Christmas in Heaven...
```

```
def sum1(a,b):
 c = a+b
 return c

if __name__ == '__main__' :
 print "Sum is ", sum1(3,6)
```

Why use modules?

- Code reuse
  - Routines can be called multiple times within a program
  - Routines can be used from multiple programs
- Namespace partitioning
  - Group data together with functions used for that data
- Implementing shared services or data
  - Can provide global data structure that is accessed by multiple subprograms
  - Modules provide an easy way to organize components into a system by serving as self-contained packages of variables known as namespaces.
  - All the names defined at the top level of a module file become attributes of the imported module object.

Modules

- function definition + executable statements
- executed only when module is imported
- modules have private symbol tables
- avoids name clash for global variables
- accessible as *module.globalname*
- can import into name space:

```
>>> from fibo import fib, fib2
>>> fib(500)
```

- can import all names defined by module:

```
>>> from fibo import *
```

Modules

- Long programs should be divided into different files
  - It makes them easier to maintain
- Example: mandelbrot.py

```
# Mandelbrot module
def inMandelbrotSet(point):
  """
  True iff point is in the Mandelbrot Set
  """
  X, t = 0 + 0j, 0
  while (t < 30):
      if abs(X) >= 2: return 0
      X, t = X ** 2 + point, t + 1
  return 1
```

Using Modules

- Importing a module
  - Syntax: import module_name

```
import mandelbrot

p = 1+0.5j

if mandelbrot.inMandelbrotSet(p):
     print "%f+%fj is in the set" % (p.real, p.imag)
else:
     print "%f+%fj is NOT in the set" % (p.real, p.imag)
```

Using Modules

- Importing functions within a modules
  - Syntax: from module_name import function_name

```
from mandelbrot import inMandelbrotSet
p = 1+0.5j
if inMandelbrotSet(p):
     print "%f+%fj is in the set" % (p.real, p.imag)
else:
     print "%f+%fj is NOT in the set" % (p.real, p.imag)
```

- Importing all the functions within a module
  - Syntax: from module_name import *

Module search path

- current directory
- list of directories specified in PYTHONPATH environment variable
- uses installation-default if not defined, e.g., .:/usr/local/lib/python
- uses sys.path

```
>>> import sys
>>> sys.path
['', 'C:\\PROGRA~1\\Python2.2', 'C:\\Program
  Files\\Python2.2\\DLLs', 'C:\\Program Files\\Python2.2\\lib',
  'C:\\Program Files\\Python2.2\\lib\\lib-tk', 'C:\\Program
  Files\\Python2.2', 'C:\\Program Files\\Python2.2\\lib\\site-
  packages']
```

Compiled Python files

- include byte-compiled version of module if there exists fibo.pyc in same directory as fibo.py
- only if creation time of fibo.pyc matches fibo.py
- automatically write compiled file, if possible
- platform independent
- doesn't run any faster, but *loads* faster
- can have only .pyc file → hide source

Standard modules

- system-dependent list
- always sys module

```
>>> import sys
>>> sys.p1
'>>> '
>>> sys.p2
'... '
>>> sys.path.append('/some/directory')
```

Module listing

- use `dir()` for each module

```
>>> dir(fibo)
['___name___', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__', '__st
din__', '__stdout__', '_getframe', 'argv', 'builtin_module_names', 'byteorder',
'copyright', 'displayhook', 'dllhandle', 'exc_info', 'exc_type', 'excepthook', '
exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getrecursionlimit', '
getrefcount', 'hexversion', 'last_type', 'last_value', 'maxint', 'maxunicode', '
modules', 'path', 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setpr
ofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'version',
 'version_info', 'warnoptions', 'winver']
```

Standard Modules

- Python has a very comprehensive set of standard modules (a.k.a. libraries)
  - See Python library reference
    - http://www.python.org/doc/current/lib/lib.html

string

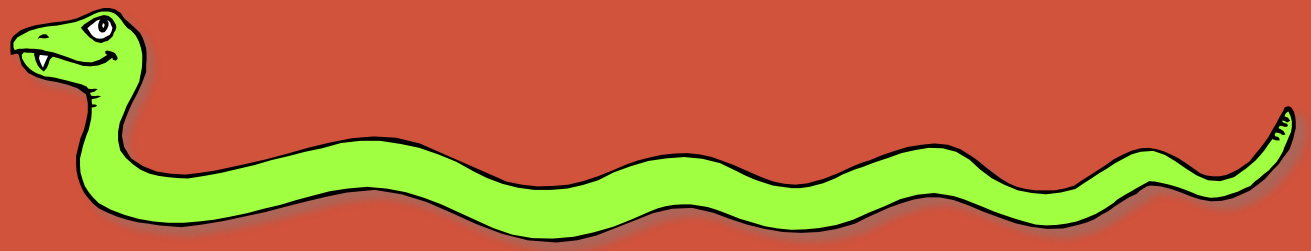
- Reference
  - http://www.python.org/doc/current/lib/module-string.html
- Some very useful functions
  - find(s, sub[, start[,end]])
  - split(s[, sep[, maxsplit]])
  - strip(s)
  - replace(str, old, new[, maxsplit])

zip and Argument Unpacking

- Often we will need to zip two or more lists together. zip transforms multiple lists into a single list of tuples of corresponding elements:

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
zip(list1, list2)        # is [('a', 1), ('b', 2), ('c', 3)]
```

PYTHON
REGULAR EXPRESSIONS




WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

“Some people, when confronted with a problem, think ‘I know, I'll use regular expressions.’ Now they have two problems.”

-- Jamie Zawinski
http://www.jwz.org/

Regular expressions

- The `re` module provides Perl-style REs
- References:
  - HOWTO
    - http://py-howto.sourceforge.net/regex/regex.html
  - Module reference
    - http://www.python.org/doc/current/lib/module-re.html

Regular Expressions

- A regular expression (RE) is:
  - A single character
  - The empty string, ε
  - The concatenation of two regular expressions
    - *Notation:* RE_1 RE_2 (*i.e.* RE_1 followed by RE_2)
  - The union of two regular expressions
    - *Notation:* RE_1 | RE_2
  - The closure of a regular expression
    - *Notation:* RE*
    - * is known as the *Kleene star*
    - * represents the concatenation of 0 or more strings

Defining Regular Expression

- Regular expression are created using compile(re)
  - *E.g.*

```
import re
regex_object = re.compile('key')
```

- Basic syntax
  - Concatenation: sequence
  - Union: |
  - Closure: *

What is a regular expression?

```
"[a-zA-Z_\-]+@(([a-zA-Z_\-])+\.)+[a-zA-Z]{2,4}"
```

- **regular expression** ("regex"): a description of a pattern of text
  - can test whether a string matches the expression's pattern
  - can use a regex to search/replace characters in a string
  - regular expressions are extremely powerful but tough to read
    - (the above regular expression matches basic email addresses)

- regular expressions occur in many places:
  - shell commands (`grep`)
  - many text editors (TextPad) allow regexes in search/replace
  - Java `Scanner`, `String` `split` (CSE 143 grammar solver)

Basic regexes

```
"abc"
```

- the simplest regexes simply match a particular substring
- this is really a pattern, not a string!
- the above regular expression matches any line containing "abc"
  - *YES*: `"abc"`, `"abcdef"`, `"defabc"`, `".=.abc.=."`, ...
  - *NO*: `"fedcba"`, `"ab c"`, `"AbC"`, `"Bash"`,...

Wildcards and anchors

`.` (a dot) matches any character except `\n`

- `".oo.y"` matches `"Doocy"`, `"goofy"`, `"LooPy"`, ...
- use `\.` to literally match a dot `.` character

`^` matches the beginning of a line; `$` the end

- `"^fi$"` matches lines that consist entirely of `fi`

`\<` demands that pattern is the beginning of a *word*;
`\>` demands that pattern is the end of a word

- `"\<for\>"` matches lines that contain the word `"for"`

- *Exercise* : Find lines in `ideas.txt` that refer to the C language.
- *Exercise* : Find act/scene numbers in `hamlet.txt`.

Special characters

| means OR

- `"abc|def|g"` matches lines with "abc", "def", or "g"
- precedence of `^(Subject|Date)` vs. `^Subject|Date:`
- There's no AND symbol.

( ) are for grouping

- `"(Homer|Marge) Simpson"` matches lines containing `"Homer Simpson"` or `"Marge Simpson"`

\ starts an escape sequence

- many characters must be escaped to match them: `/ \ $ . [ ] ( ) ^ * + ?`
- `"\.\\n"` matches lines containing `".\n"`

Quantifiers: * + ?

`*` means 0 or more occurrences

- "`abc*`" matches "ab", "abc", "abcc", "abccc", ...
- "`a(bc)*`" matches "a", "abc", "abcbc", "abcbcbc", ...
- "`a.*a`" matches "aa", "aba", "a8qa", "a!?_a", ...

`+` means 1 or more occurrences

- "`a(bc)+`" matches "abc", "abcbc", "abcbcbc", ...
- "`Goo+gle`" matches "Google", "Gooogle", "Goooogle", ...

`?` means 0 or 1 occurrences

- "`Martina?`" matches lines with "Martin" or "Martina"
- "`Dan(iel)?`" matches lines with "Dan" or "Daniel"

- *Exercise* : Find all `^^ or ^_^` type smileys in `chat.txt`.

More quantifiers

`{min,max}` means between ***min*** and ***max*** occurrences

- `"a(bc){2,4}"` matches "abcbc", "abcbcbc", or "abcbcbcbc"

- ***min*** or ***max*** may be omitted to specify any number
  - `"{2,}"` means 2 or more
  - `"{,6}"` means up to 6
  - `"{3}"` means exactly 3

Character sets

`[ ]` group characters into a character set;
will match any single character from the set

- `"[bcd]art"` matches strings containing "bart", "cart", and "dart"
- equivalent to `"(b|c|d)art"` but shorter

- inside `[ ]`, most modifier keys act as normal characters
  - `"what[.!*?]*"` matches "what", "what.", "what!", "what?**!", ...

- *Exercise* : Match letter grades in 143.txt such as A, B+, or D- .

Character ranges

- inside a character set, specify a range of characters with -
  - "`[a-z]`" matches any lowercase letter
  - "`[a-zA-Z0-9]`" matches any lower- or uppercase letter or digit

- an initial `^` inside a character set negates it
  - "`[^abcd]`" matches any character other than a, b, c, or d

- inside a character set, `-` must be escaped to be matched
  - "`[+\-]?[0-9]+`" matches optional + or -, followed by ≥ one digit

- *Exercise* : Match phone #s in `faculty.html`, e.g. (206) 685-2181 .

Regular Expressions

- Regular expressions are a powerful string manipulation tool
- All modern languages have similar library packages for regular expressions
- Use regular expressions to:
  - Search a string (`search` and `match`)
  - Replace parts of a string (`sub`)
  - Break strings into smaller pieces (`split`)

Regular Expression Quick Guide

^ Matches the beginning of a line
$ Matches the end of the line
. Matches any character
\s Matches whitespace
\S Matches any non-whitespace character
* Repeats a character zero or more times
*? Repeats a character zero or more times (non-greedy)
+ Repeats a chracter one or more times
+? Repeats a character one or more times (non-greedy)
[aeiou] Matches a single character in the listed set
[^XYZ] Matches a single character not in the listed set
[a-z0-9] The set of characters can include a range
(Indicates where string extraction is to start
) Indicates where string extraction is to end

The Regular Expression Module

- Before you can use regular expressions in your program, you must import the library using "import re"
- You can use re.search() to see if a string matches a regular expression similar to using the find() method for strings
- You can use re.findall() extract portions of a string that match your regular expression similar to a combination of find() and slicing: var[5:10]

Python's Regular Expression Syntax

- Most characters match themselves

  The regular expression "test" matches the string `'test'`, and only that string

- [x] matches any *one* of a list of characters

  "[abc]" matches `'a'`,`'b'`,or `'c'`

- [^x] matches any *one* character that is not included in *x*

  "[^abc]" matches any single character *except* `'a'`,`'b'`,or `'c'`

Python's Regular Expression Syntax

- "." matches any single character
- Parentheses can be used for grouping

  "(abc)+" matches `'abc'`, `'abcabc'`, `'abcabcabc'`, etc.
- *x*|*y* matches *x* or *y*

  "this|that" matches `'this'` and `'that'`, but not `'thisthat'`.

Python'sRegular Expression Syntax

- *x** matches zero or more *x*'s

  "a*" matches `''`, `'a'`, `'aa'`, etc.

- *x*+ matches one or more *x*'s

  "a+" matches `'a'`,`'aa'`,`'aaa'`, etc.

- *x*? matches zero or one *x*'s

  "a?" matches `''` or `'a'`

- *x{m, n}* matches *i x*'s, where $m \leq i \leq n$

  "a{2,3}" matches `'aa'` or `'aaa'`

Regular Expression Syntax

- “\d” matches any digit; “\D” any non-digit
- “\s” matches any whitespace character; “\S” any non-whitespace character
- “\w” matches any alphanumeric character; “\W” any non-alphanumeric character
- “^” matches the beginning of the string; “$” the end of the string
- “\b” matches a word boundary; “\B” matches a character that is not a word boundary

Search and Match

- The two basic functions are **re.search** and **re.match**
  - Search looks for a pattern anywhere in a string
  - Match looks for a match staring at the beginning
- Both return *None* (logical false) if the pattern isn't found and a "match object" instance if it is

```
>>> import re
>>> pat = "a*b"
>>> re.search(pat,"fooaaabcde")
<_sre.SRE_Match object at 0x809c0>
>>> re.match(pat,"fooaaabcde")
>>>
```

Q: What's a match object?

- A: an instance of the match class with the details of the match result

```
>>> r1 = re.search("a*b","fooaaabcde")
>>> r1.group()  # group returns string matched
'aaab'
>>> r1.start()  # index of the match start
3
>>> r1.end()    # index of the match end
7
>>> r1.span()   # tuple of (start, end)
(3, 7)
```

Using re.search() like find()

```
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if line.find('From:') >= 0:
        print line
```

```
import re

hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line) :
        print line
```

Using re.search() like startswith()

```
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if line.startswith('From:') :
        print line
```

```
import re

hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line) :
        print line
```

We fine-tune what is matched by adding special characters to the strin

What got matched?

- Here's a pattern to match simple email addresses

  \w+@(\w+\.)+(com|org|net|edu)

```
>>> pat1 = "\w+@(\w+\.)+(com|org|net|edu)"
>>> r1 = re.match(pat,"finin@cs.umbc.edu")
>>> r1.group()
'finin@cs.umbc.edu'
```

- We might want to extract the pattern parts, like the email name and host

What got matched?

- We can put parentheses around groups we want to be able to reference

```
>>> pat2 = "(\w+)@((\w+\.)+(com|org|net|edu))"
>>> r2 = re.match(pat2,"finin@cs.umbc.edu")
>>> r2.group(1)
'finin'
>>> r2.group(2)
'cs.umbc.edu'
>>> r2.groups()
r2.groups()
('finin', 'cs.umbc.edu', 'umbc.', 'edu')
```

- Note that the ‘groups’ are numbered in a preorder traversal of the forest

What got matched?

- We can ‘label’ the groups as well…

```
>>> pat3
 ="(?P<name>\w+)@(?P<host>(\w+\.)+(com|org
 |net|edu))"
>>> r3 =
 re.match(pat3,"finin@cs.umbc.edu")
>>> r3.group('name')
'finin'
>>> r3.group('host')
'cs.umbc.edu’
```

- And reference the matching parts by the labels

More re functions

- re.split() is like split but can use patterns

```
>>> re.split("\W+", “This... is a test,
  short and sweet, of split().”)
['This', 'is', 'a', 'test', 'short’,
  'and', 'sweet', 'of', 'split’, ‘’]
```

- re.sub substitutes one string for a pattern

```
>>> re.sub('(blue|white|red)', 'black', 'blue
 socks and red shoes')
'black socks and black shoes’
```

- re.findall() finds al matches

```
>>> re.findall("\d+”,"12 dogs,11 cats, 1 egg")
['12', '11', ’1’]
```

Regular Expression Matching

- Matching at the beginning of strings
  - Syntax: math_object = re.match(string)
  - *E.g.*

```
>>> import re
>>> regex = re.compile('key')
>>> match = regex.match("I have a key")
>>> print match
None
>>> match = regex.match("keys open doors")
>>> print match
<_sre.SRE_Match object at 0x009B6068>
```

Regular Expression Matching

- Matching within strings
  - Syntax: math_object = re.search(string)
  - *E.g.*

```
>>> regex = re.compile('11*')
>>> match = regex.match("I have $111 dollars")
>>> print match
None
>>> match = regex.search("I have $111 dollars")
>>> print match
<_sre.SRE_Match object at 0x00A036F0>
>>> match.group()
'111'
```

Regular Expression Repetition

- Greedy vs Nongreedy matching
  - Greedy matching gets the longest results possible
  - Nongreedy matching gets the shortest possible
  - Let's say $robot = 'The12thRobotIs2ndInLine'

    - $robot =~ /\w*\d+/; (greedy)
      - Matches The12thRobotIs2
      - Maximizes the length of \w
    - $robot =~ /\w*?\d+/; (nongreedy)
      - Matches The12
      - Minimizes the length of \w

Regular Expression Repetition

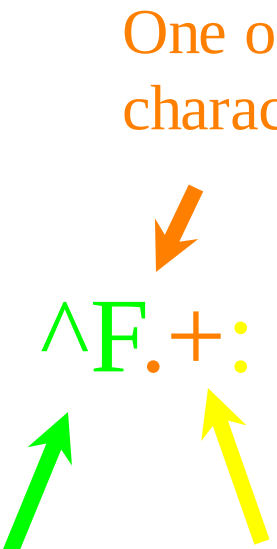
- **Greedy vs Nongreedy matching**
  - Suppose $txt = ‘something is so cool’;
    - $txt =~ /something/;
      - Matches ‘something’

    - $txt =~ /so(mething)?/;
      - Matches ‘something’ and the second ‘so’
    - $txt =~ /so(mething)??/;
      - Matches only ‘so’ and the second ‘so’
      - Doesn’t really make sense to do this

Warning: Greedy Matching

- The repeat characters (* and +) push outward in both directions (greedy) to match the largest possible string

```
>>> import re
>>> x = 'From: Using the : character'
>>> y = re.findall('^F.+:', x)
>>> print y
['From: Using the :']
```

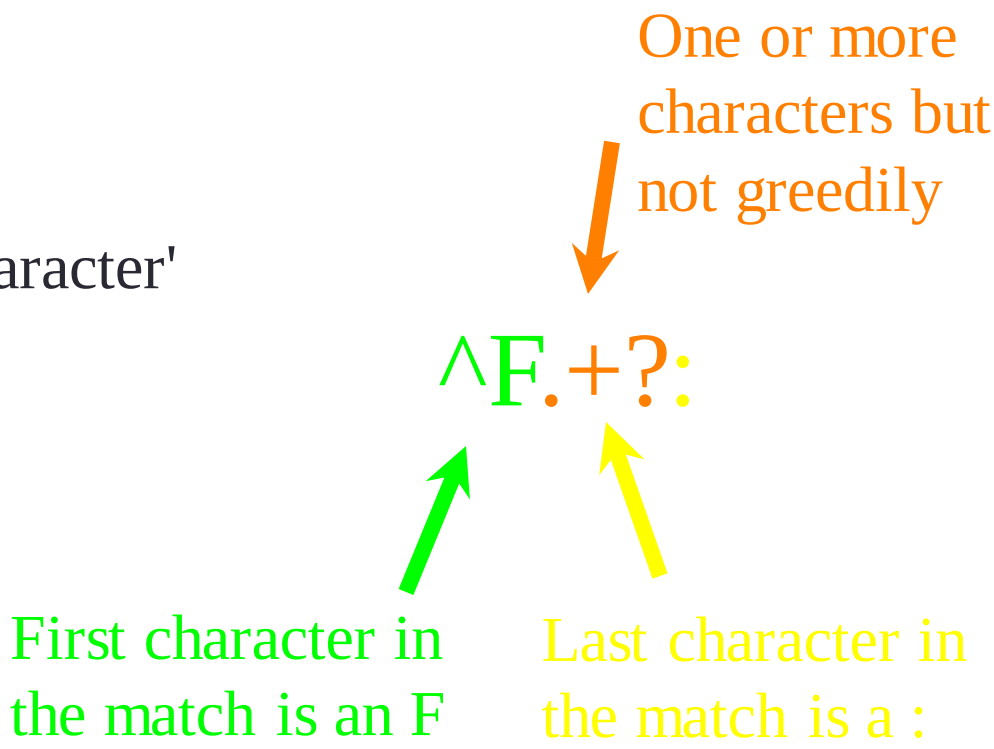


Why not 'From:'?

Non-Greedy Matching

- Not all regular expression repeat codes are greedy! If you add a ? character - the + and * chill out a bit...

```
>>> import re
>>> x = 'From: Using the : character'
>>> y = re.findall('^F.+?:', x)
>>> print y
['From:']
```



Matching Object Methods

- Return matched string
  - Syntax: `string = match_object.group()`
- Return starting position of the match
  - Syntax: `m = match_object.start()`
- Return ending position of the match
  - Syntax: `n = match_object.end()`
- Return a tuple with start and end positions
  - Syntax: `m, n = match_object.span()`

Compiling regular expressions

- If you plan to use a re pattern more than once, compile it to a re object
- Python produces a special data structure that speeds up matching

```
>>> capt3 = re.compile(pat3)
>>> cpat3
<_sre.SRE_Pattern object at 0x2d9c0>
>>> r3 = cpat3.search("finin@cs.umbc.edu")
>>> r3
<_sre.SRE_Match object at 0x895a0>
>>> r3.group()
'finin@cs.umbc.edu'
```

Extended Syntax

- Ranges
  - Any character within `[]` (*e.g.* `[abc]`)
  - Any character not within `[]` (*e.g.* `[^abc]`)
- Predefined ranges
  - `\d` Matches any decimal digit; this is equivalent to the set `[0-9]`.
  - `\D` Matches any non-digit character; equivalent to the set `[^0-9]`.
  - `\s` Matches any whitespace character; this is equivalent to the set `[\t\n\r\f\v]`.
  - `\S` Matches any non-whitespace character; this is equivalent to the set [^ \t\n\r\f\v].
- One or more closure `+` (*e.g.* `a+`)

Grouping

- Groups are defined using parentheses
  - *E.g.*

```
>>> regex = re.compile('(ab*)(c+)(def)')
>>> match = regex.match("abbbdefhhggg")
>>> print match
None
>>> match = regex.match("abbbcdefhhggg")
>>> print match
<_sre.SRE_Match object at 0x00A35A10>
>>> match.groups()
('abbb', 'c', 'def')
```

Grouping for Backreferences

- **Backreferences**
  - With all these wildcards and possible matches, we usually need to know what the expression finally ended up matching.
    - Backreferences let you see what was matched
    - Can be used after the expression has evaluated or even inside the expression itself
    - Handled very differently in different languages
    - Numbered from left to right, starting at 1

Grouping for Backreferences

- **PHP/Python backreferences**
  - Allows the use of specifically named backreferences
    - Groups also maintain their numbers
- **.NET backreferences**
  - Allows named backreferences
    - If you try to access named groups by number, stuff breaks

- **Check the web for info on how to use backreferences in these and other languages.**

Pattern object methods

Pattern objects have methods that parallel the re functions (e.g., match, search, split, findall, sub), e.g.:

```
>>> p1 = re.compile("\w+@\w+\.+com|org|net|edu")
>>> p1.match("steve@apple.com").group(0)
'steve@apple.com'
>>> p1.search(”Email steve@apple.com today.").group(0)
'steve@apple.com’
>>> p1.findall("Email steve@apple.com and bill@msft.com now.")
['steve@apple.com', 'bill@msft.com’]
>>> p2 = re.compile("[.?!]+\s+")
>>> p2.split("Tired? Go to bed!   Now!! ")
['Tired', 'Go to bed', 'Now', ’ ']
```

email address

sentence boundary

Example: pig latin

- Rules
  - If word starts with consonant(s)
    - Move them to the end, append “ay”
  - Else word starts with vowel(s)
    - Keep as is, but add “zay”
  - How might we do this?

http://cs.umbc.edu/courses/331/current/code/python/pig.py

The pattern

```
([bcdfghjklmnpqrstvwxyz]+)(\w+)
```

piglatin.py

```
import re
pat = ‘([bcdfghjklmnpqrstvwxyz]+)(\w+)’
cpat = re.compile(pat)

def piglatin(string):
  return " ".join( [piglatin1(w) for w in string.split()] )
```

piglatin.py

```
def piglatin1(word):
    """Returns the pig latin form of a word. e.g.:
    piglatin1("dog”) => "ogday". """
    match = cpat.match(word)
    if match:
        consonants = match.group(1)
        rest = match.group(2)
        return rest + consonents + “ay”
    else:
        return word + "zay"
```

Exceptions

- Try/except/raise

```
>>> while 1:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was not valid.  Try again"
...
```

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Handling exceptions

```
while 1:
  try:
    x = int(raw_input("Please enter a number: "))
    break
  except ValueError:
    print "Not a valid number"
```

- First, execute `try` clause
- if no exception, skip `except` clause
- if exception, skip rest of `try` clause and use except clause
- if no matching exception, attempt outer `try` statement

Handling exceptions

- try.py

```
import sys
for arg in sys.argv[1:]:
    try:
    f = open(arg, 'r')
    except IOError:
    print 'cannot open', arg
 else:
    print arg, 'lines:', len(f.readlines())
    f.close
```

- e.g., as `python try.py *.py`

Exceptions

- syntax (parsing) errors

```
while 1 print 'Hello World'
File "<stdin>", line 1
   while 1 print 'Hello World'
                 ^
SyntaxError: invalid syntax
```

- exceptions
  - run-time errors
  - e.g., `ZeroDivisionError, NameError, TypeError`

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

Catching Exceptions

#python code a.py

```
x = 0
try:
    print 1/x
except ZeroDivisionError, message:
    print "Can’t divide by zero:"
    print message
```

>>>python a.py

Can't divide by zero:

integer division or modulo by zero

Try-Finally: Cleanup

```
f = open(file)
try:
    process_file(f)
finally:
    f.close()     # always executed
print "OK"       # executed on success only
```

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Raising Exceptions

- raise IndexError
- raise IndexError("k out of range")
- raise IndexError, "k out of range”
- try:
  something
  except: # catch everything
  print "Oops"
  raise # reraise

Reading Assignment

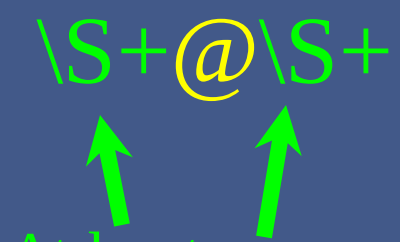
- Guido van Rossum and Fred L. Drake, Jr. (ed.), *Python tutorial*, PythonLabs, 2001.
  - Read chapters 6 to 9
  - http://www.python.org/doc/current/tut/tut.html
  - Write some simple programs
- Glance at the RE HOWTO and reference
  - You will need REs in the next assignment
  - HOWTO
    - http://py-howto.sourceforge.net/regex/regex.html
  - Module reference
    - http://www.python.org/doc/current/lib/module-re.html

Fine Tuning String Extraction

- You can refine the match for re.findall() and separately determine which portion of the match that is to be extracted using parenthesis

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
>>> y = re.findall('\S+@\S+',x)
>>> print y
['stephen.marquard@uct.ac.za']
>>> y = re.findall('^From:.*? (\S+@\S+)',x)
>>> print y['stephen.marquard@uct.ac.za']
```



Fine Tuning String Extraction

- Parenthesis are not part of the match - but they tell where to start and stop what string to extract

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
>>> y = re.findall('\S+@\S+',x)
>>> print y
['stephen.marquard@uct.ac.za']
>>> y = re.findall('^From (\S+@\S+)',x)
>>> print y
['stephen.marquard@uct.ac.za']
```

^From (\S+@\S+)


```
>>> data = 'From stephen.marquard@uct.ac.za  Sat Jan  5 09:14:16 2
>>> atpos = data.find('@')
>>> print atpos
21
>>> sppos = data.find(' ',atpos)
>>> print sppos
31
>>> host = data[atpos+1 : sppos]
>>> print host
uct.ac.za
```

Extracting a host name - using find and string slicing.

The Double Split Version

- Sometimes we split a line one way and then grab one of the pieces of the line and split that piece again

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

The Double Split Version

- Sometimes we split a line one way and then grab one of the pieces of the line and split that piece again

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
words = line.split()
email = words[1]
pieces = email.split('@')
print pieces[1]
```

stephen.marquard@uct.ac.za

['stephen.marquard', 'uct.ac.za']

'uct.ac.za'

The Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
lin = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16
2008'
y = re.findall('@([^ ]*)',lin)
print y['uct.ac.za']
```

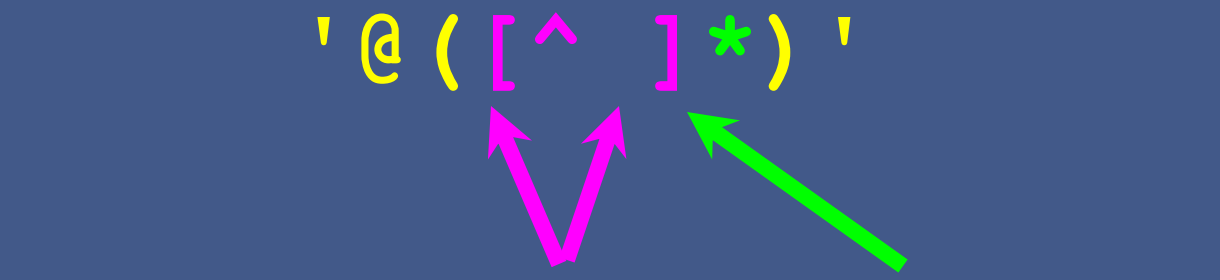
```
'@([^ ]*)'
```

Look through the string until you find an at-sign

The Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
lin = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16
2008'
y = re.findall('@([^ ]*)',lin)
print y['uct.ac.za']
```

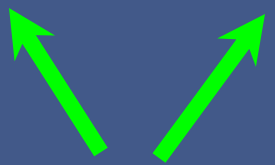


The Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
lin = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16
2008'
y = re.findall('@([^ ]*)',lin)
print y['uct.ac.za']
```

```
'@([^ ]*)'
```



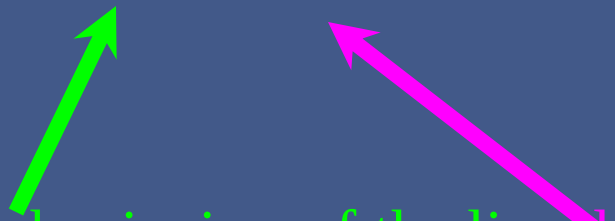
Extract the non-blank characters

Even Cooler Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
lin = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16
2008'
y = re.findall('^From .*@([^ ]*)',lin)
print y['uct.ac.za']
```

'^From .*@([^]*)'



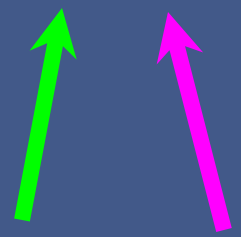
Starting at the beginning of the line, look for the string 'From '

Even Cooler Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
lin = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16
2008'
y = re.findall('^From .*@([^ ]*)',lin)
print y['uct.ac.za']
```

'^From .*@([^]*)'



Skip a bunch of characters, looking for an at-sign

Even Cooler Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
lin = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16
2008'
y = re.findall('^From .*@([^ ]*)',lin)
print y['uct.ac.za']
```

'^From .*@([^]*)'

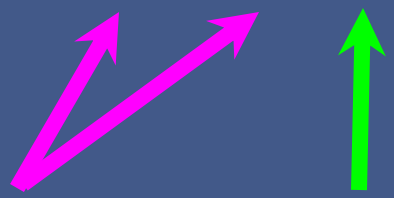
Start 'extracting'

Even Cooler Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
lin = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16
2008'
y = re.findall('^From .*@([^ ]*)',lin)
print y['uct.ac.za']
```

'^From .*@([^]*)'



Match non-blank characte Match many of the

Even Cooler Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
lin = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16
2008'
y = re.findall('^From .*@([^ ]*)',lin)
print y['uct.ac.za']
```

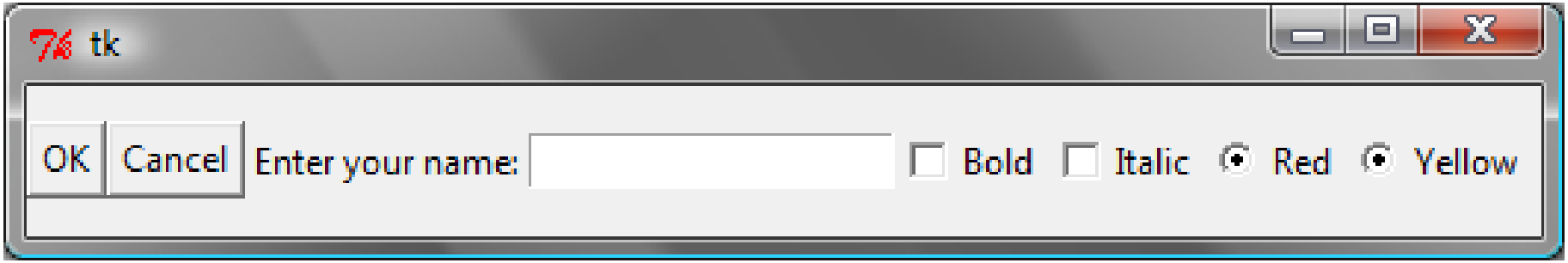
'^From .*@([^]*)'

Stop 'extracting'

OOP

Motivations of OOP

After learning the preceding chapters, you are capable of solving many programming problems using selections, loops, and functions. However, these Python features are not sufficient for developing graphical user interfaces and large scale software systems. Suppose you want to develop a graphical user interface as shown below. How do you program it?

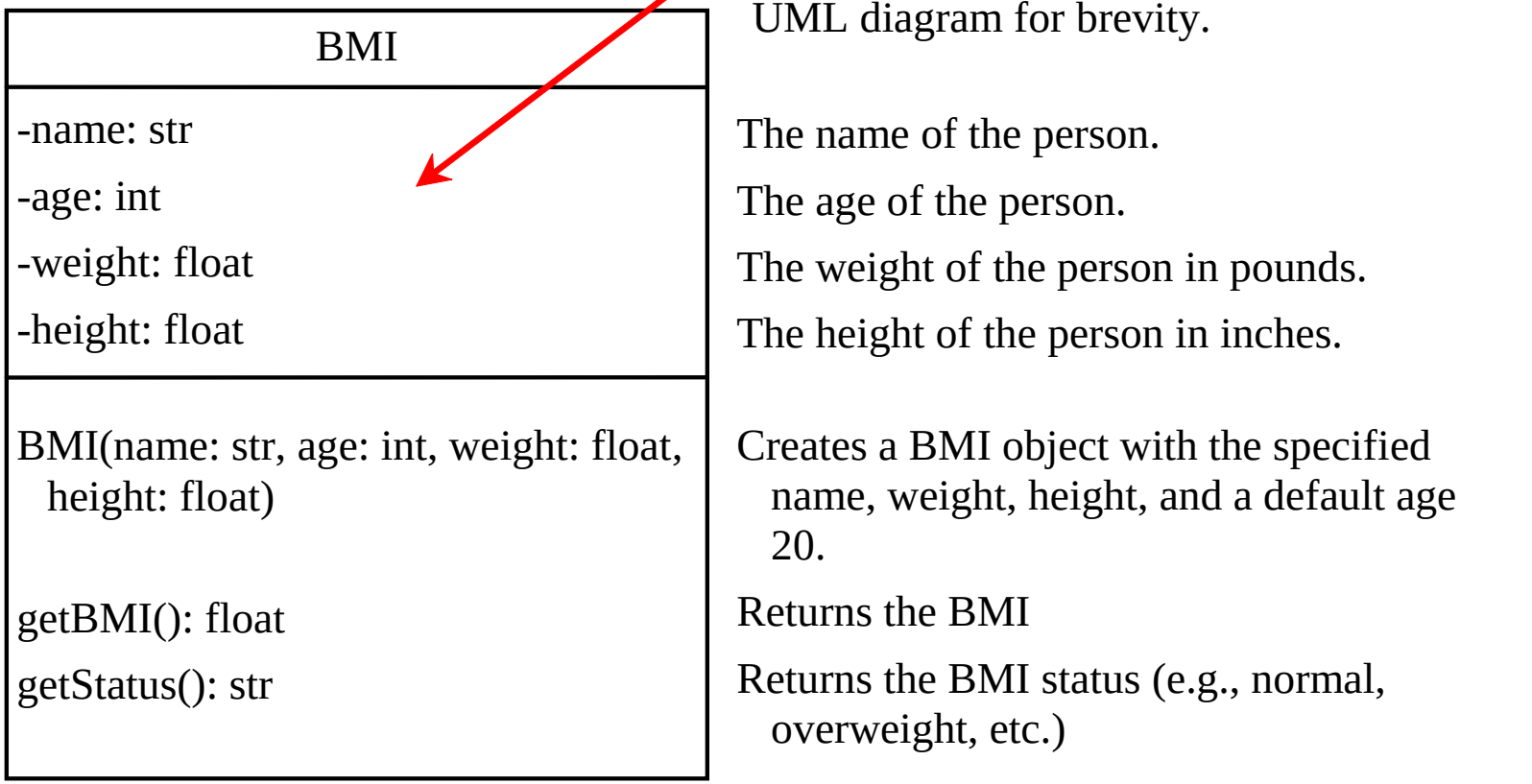


@07slide.ppt

OO Programming Concepts

Object-oriented programming (OOP) involves programming using objects. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behaviors. The *state* of an object consists of a set of *data fields* (also known as *properties*) with their current values. The *behavior* of an object is defined by a set of methods.

The BMI Class



The get methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The name of the person.

The age of the person.

The weight of the person in pounds.

The height of the person in inches.

Creates a BMI object with the specified name, weight, height, and a default age 20.

Returns the BMI

Returns the BMI status (e.g., normal, overweight, etc.)

BMI UseBMIClass Run

OOP Terminology

- class -- a template for building objects
- instance -- an object created from the template (an instance of the class)
- method -- a function that is part of the object and acts on instances directly
- constructor -- special "method" that creates new instances

Classes

- Defined using `class` and indentation
  - E.g.

```
class MyClass:
        "A simple example class"
        i = 12345
        def f(self):
            return 'hello world'
```

- *Methods* are functions defined within the class declaration or using the *dot* notation
- *Attributes* are variables defined within the the class declaration or using the *dot* notation

Class Constructor

- `__init__` method
  - E.g.

```
class MyClass:
    def __init__(self):
            self.data = []
```

- Creation of instances is straightforward
  - E.g.

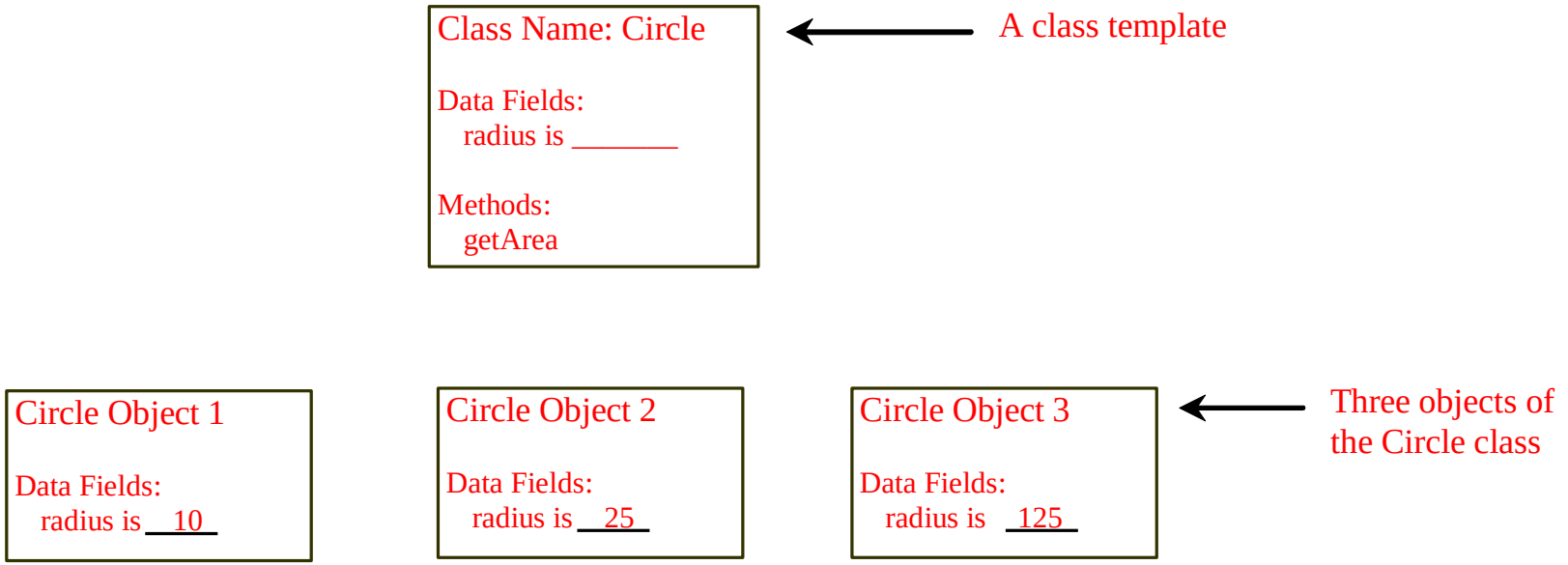
```
x = MyClass()
```

Class Examples

- Example

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Objects



An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.

A Example of Python Class

```
class Person:

    def __init__(self,name):
        self.name = name

    def Sayhello(self):
        print 'Hello, my name is', self.name

    def __del__(self):
        print '%s says bye.' % self.name

A = Person('Yang Li')
```

This example includes
class definition, **constructor function**, **destructor function**, **attributes and methods definition** and **object definition.**
These definitions and uses will be introduced specifically in the following.

@Yang Li

Defining a class

```
class Thingy:
    """This class stores an arbitrary object."""
    def __init__(self, value):
        """Initialize a Thingy."""
        self.value = value
    def showme(self):
        """Print this object to stdout."""
        print "value = %s" % self.value
```

constructor

method

Classes

- mixture of C++ and Modula-3
- multiple base classes
- derived class can override any methods of its base class(es)
- method can call the method of a base class with the same name
- objects have private data
- C++ terms:
  - all class members are public
  - all member functions are virtual
  - no constructors or destructors (not needed)

Classes

- classes (and data types) are objects
- built-in types cannot be used as base classes by user
- arithmetic operators, subscripting can be redefined for class instances (like C++, unlike Java)

Special Class Attributes in Python

- Except for self-defined class attributes in Python, class has some special attributes. They are provided by object module.

Attributes Name	Description
__dict__	Dict variable of class name space
__doc__	Document reference string of class
__name__	Class name
__module__	Module name consisting of class
__bases__	The tuple including all the superclasses

@Yang Li

Class objects

- class instantiation:

```
>>> x = MyClass()
>>> x.f()
'hello world'
```

- creates new instance of class
  - note `x = MyClass` vs. `x = MyClass()`
- `___init__()` special method for initialization of object

```
def __init__(self,realpart,imagpart):
  self.r = realpart
  self.i = imagpart
```

Constructor: __init__()

- The___init___method is run as soon as an object of a class is instantiated. Its aim is to initialize the object.

From the code , we can see that after instantiate object, it automatically invokes __init__()

As a result, it runs
self.name = 'Yang Li',
and
print self.name

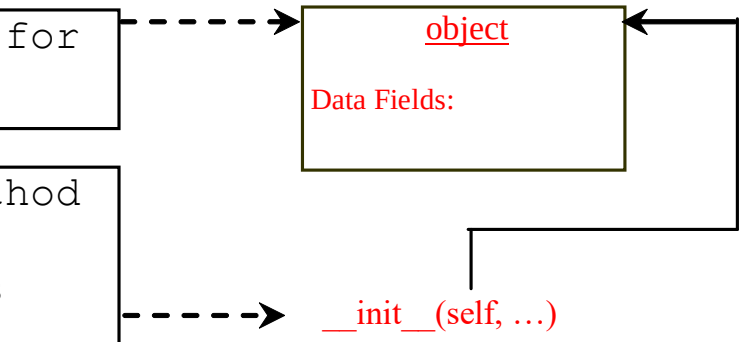
```
>>> class Person:
    def __init__(self,name):
        self.name = name
        print self.name


>>> A = Person('Yang Li')
Yang Li
>>> A.name
'Yang Li'
```

Constructing Objects

Once a class is defined, you can create objects from the class by using the following syntax, called a *constructor*:

```
className(arguments)
```

1. It creates an object in the memory for the class.
2. It invokes the class's `__init__` method to initialize the object. The `self` parameter in the `__init__` method is automatically set to reference the object that was just created.



"Special" methods

- All start and end with `__` (two underscores)
- Most are used to emulate functionality of built-in types in user-defined classes
- *e.g.* operator overloading
  - `__add__`, `__sub__`, `__mult__`, ...
  - see python docs for more information

Another example

- bag.py

```
class Bag:
  def __init__(self):
    self.data = []
  def add(self, x):
    self.data.append(x)
  def addtwice(self,x):
    self.add(x)
    self.add(x)
```

Another example, cont'd.

- invoke:

```
>>> from bag import *
>>> l = Bag()
>>> l.add('first')
>>> l.add('second')
>>> l.data
['first', 'second']
```

Inheritance

```
class Person:
    def speak(self):
        print 'I can speak'

class Man(Person):
    def wear(self):
        print 'I wear shirt'

class Woman(Person):
    def wear(self):
        print 'I wear Skirt'

man = Man()
man.wear()
man.speak()

>>>
I wear shirt
I can speak
```

Inheritance in Python is simple,
Just like JAVA, subclass can invoke
Attributes and methods in superclass.

From the example, **Class Man** inherits **Class Person**, and invoke **speak()** method In **Class Person**

Inherit Syntax:

class subclass(superclass):
…
…

In Python, it supports multiple inheritance,
In the next slide, it will be introduced.

@Yang Li

An Example of Multiple Inheritance

C multiple-inherit A and B, but since A is in the left of B, so C inherit A and invoke A.A() according to the left-to-right sequence.

To implement C.B(), class A does not have B() method, so C inherit B for the second priority. So C.B() actually invokes B() in class B.

```
class A:
    def A(self):
        print 'I am A'

class B:
    def A(self):
        print 'I am a'
    def B(self):
        print 'I am B'

class C(A,B):
    def C(self):
        print 'I am C'

C = C();
C.A()
C.B()
C.C()
```

@Yang Li

"Self"

- "Self" in Python is like the pointer "this" in C++. In Python, functions in class access data via "self".

```
class Person:
    def __init__(self,name):
        self.name = name
    def PrintName(self):
        print self.name

P = Person('Yang Li')
print P.name
P.PrintName()
```

- "Self" in Python works as a variable of function but it won't invoke data.

@Yang Li

The datetime Class

```
from datetime import datetime
d = datetime.now()
print("Current year is " + str(d.year))
print("Current month is " + str(d.month))
print("Current day of month is " + str(d.day))
print("Current hour is " + str(d.hour))
print("Current minute is " + str(d.minute))
print("Current second is " + str(d.second))
```

Data Field Encapsulation

CircleWithPrivateDataRadius

```
>>> from CircleWithPrivateRadius import Circle
>>> c = Circle(5)
>>> c.__radius
AttributeError: 'Circle' object has no attribute
'__radius'
>>> c.getRadius()
 5
```

Encapsulation – Accessibility (1)

- In Python, there is no keywords like ‘public’, ‘protected’ and ‘private’ to define the accessibility. In other words, In Python, it acquiesce that all attributes are public.

- But there is a method in Python to define Private:

Add “_” in front of the variable and function name can hide them when accessing them from out of class.

@Yang Li

An Example of Private

```
class Person:
    def __init__(self):

        self.A = 'Yang Li'
        self.__B = 'Yingying Gu'

    def PrintName(self):
        print self.A
        print self.__B

P = Person()

>>> P.A
'Yang Li'
>>> P.__B

Traceback (most recent call last):
  File "<pyshell#61>", line 1, in <module>
    P.__B
AttributeError: Person instance has no attribute '__B'
>>> P.PrintName()
Yang Li
Yingying Gu
```

Public variable

Private variable

Invoke private variable in class

Access public variable out of class, succeed

Access private variable our of class, fail

Access public function but this function access Private variable __B successfully since they are in the same class.

@Yang Li

An example of Accessing Private

```
class C:
    def accessible(self):
        print 'you can see me'
    def __inaccessible(self):
        print 'you can not see me'
```

Define public function

Define private function

```
>>> C().accessible()
you can see me
>>> C().inaccessible()
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    C().inaccessible()
AttributeError: C instance has no attribute 'inaccessible'
>>> C()._C__inaccessible()
you can not see me
```

Access public function

Can't access private function

Access private function via changed name

@Yang Li

Compare Accessibility of Python and Java

- Java is a static and strong type definition language. Java has strict definition of accessibility type with keywords.
- While Python is a dynamic and weak type definition language. Python acquiesces all the accessibility types are public except for supporting a method to realize private accessibility virtually.
- Someone think Python violates the requirement of encapsulation. But its aim is to make language simple.

@Yang Li

Traditional Polymorphism Example

```
class Animal:
    def Name(self):
        pass
    def Sleep(self):
        print 'sleep'
    def MakeNoise(self):
        pass

class Dog(Animal):
    def Name(self):
        print 'I am a dog!'
    def MakeNoise(self):
        print 'Woof!'

class Cat(Animal):
    def Name(self):
        print 'I am a cat!'
    def MakeNoise(self):
        print 'Meow'

class Lion(Animal):
    def Name(self):
        print 'I am a lion!'
    def MakeNoise(self):
        print 'Roar'

class TestAnimals:
    def PrintName(self,animal):
        animal.Name()
    def GotoSleep(self,animal):
        animal.Sleep()
    def MakeNoise(self,animal):
        animal.MakeNoise()
```

```
TestAnimals = TestAnimals()
dog = Dog()
cat = Cat()
lion = Lion()

TestAnimals.PrintName(dog)
TestAnimals.GotoSleep(dog)
TestAnimals.MakeNoise(dog)
TestAnimals.PrintName(cat)
TestAnimals.GotoSleep(cat)
TestAnimals.MakeNoise(cat)
TestAnimals.PrintName(lion)
TestAnimals.GotoSleep(lion)
TestAnimals.MakeNoise(lion)
```

```
>>>
I am a dog!
sleep
Woof!
I am a cat!
sleep
Meow
I am a lion!
sleep
Roar
```

@Yang Li

Everywhere is polymorphism in Python (2)

- So, in Python, many operators have the property of polymorphism. Like the following example:

```
>>> 1+2
3
>>> 'key'+'board'
'keyboard'
>>> [1,2,3]+[4,5,6,7]
[1, 2, 3, 4, 5, 6, 7]
>>> (1,2,3)+(4,5,6)
(1, 2, 3, 4, 5, 6)
>>> {A:a, B:b}+{C:c, D:d}
```

- Looks stupid, but the key is that variables can support any objects which support 'add' operation. Not only integer but also string, list, tuple and dictionary can realize their relative 'add' operation.

@Yang Li

OOP in Python

- As a OOP language, Python has its special advantages but also has its disadvantages.
- Python can support operator overloading and multiple inheritance etc. advanced definition that some OOP languages don't have.
- The advantages for Python to use design pattern is that it supports dynamic type binding. In other words, an object is rarely only one instance of a class, it can be dynamically changed at runtime.
- But Python might ignore a basic regulation of OOP: data and methods hiding. It doesn't have the keywords of 'private', 'public' and 'protected' to better support encapsulation. But I think it aims to guarantee syntax simple. As the inventor of Python, Guido Van Rossum said: "abundant syntax bring more burden than help".

@Yang Li

Language comparison

		Tcl	Perl	Python	JavaScript	Visual Basic
Speed	development	✓	✓	✓	✓	✓
	regexp	✓	✓	✓		
breadth	extensible	✓		✓		✓
	embeddable	✓		✓		
	easy GUI	✓		✓ (Tk)		✓
	net/web	✓	✓	✓	✓	✓
enterprise	cross-platform	✓	✓	✓	✓	
	I18N	✓		✓	✓	✓
	thread-safe	✓		✓		✓
	database access	✓	✓	✓	✓	✓

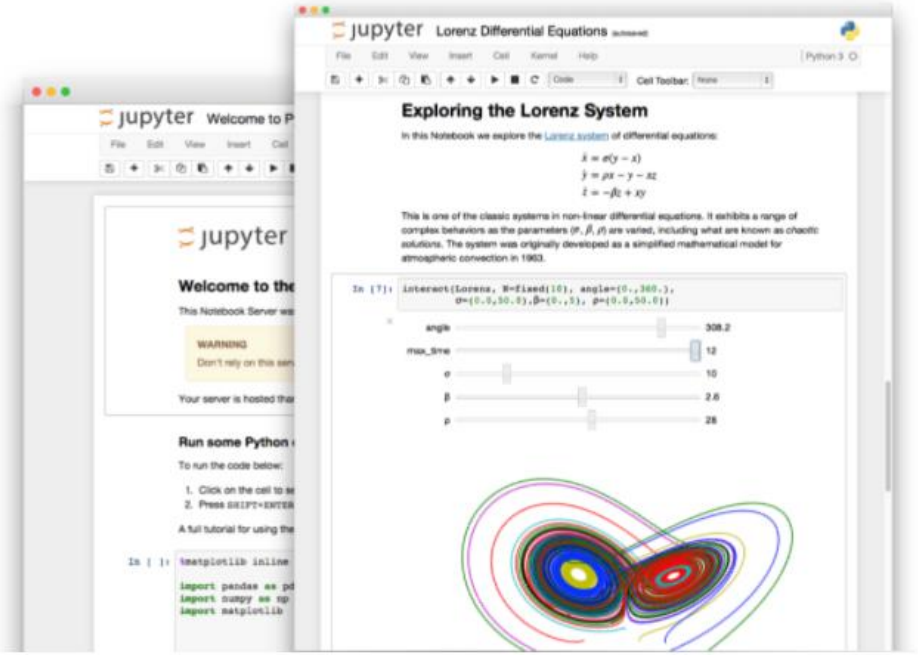
Python: Pros & Cons

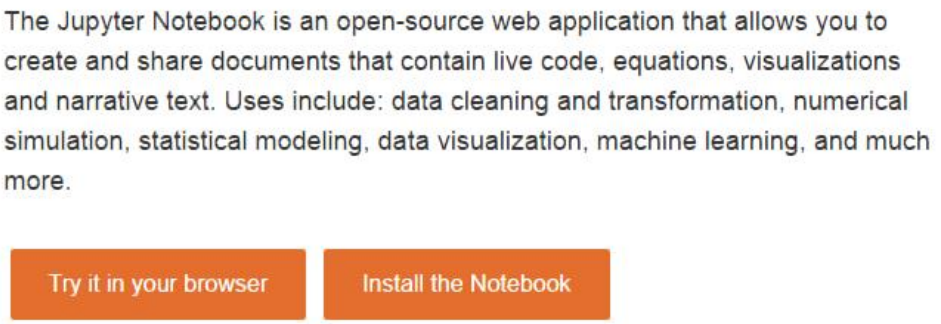
- Pros
  - Free availability (like Perl, Python is open source).
  - Stability (Python is in release 2.6 at this point and, as I noted earlier, is older than Java).
  - Very easy to learn and use
  - Good support for objects, modules, and other reusability mechanisms.
  - Easy integration with and extensibility using C and Java.
- Cons
  - Smaller pool of Python developers compared to other languages, such as Java
  - Lack of true multiprocessor support
  - Absence of a commercial support point, even for an Open Source project (though this situation is changing)
  - Software performance slow, not suitable for high performance applications

DATA SCIENCE APPLICATIONS

Jupyter Notebook

- *"The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, machine learning and much more."*
- *–description from [Project Jupyter](#)*





What is PIP install in Python?

- PIP is a package management system used to install and manage software packages written in Python.
- It stands for “preferred installer program” or “Pip Installs Packages.”
- PIP for Python is a utility to manage PyPI package installations from the command line.
- As of January 2020, the Python Package Index (PyPI), the official repository for third-party Python software, contains over 287,000 packages with a wide range of functionality.

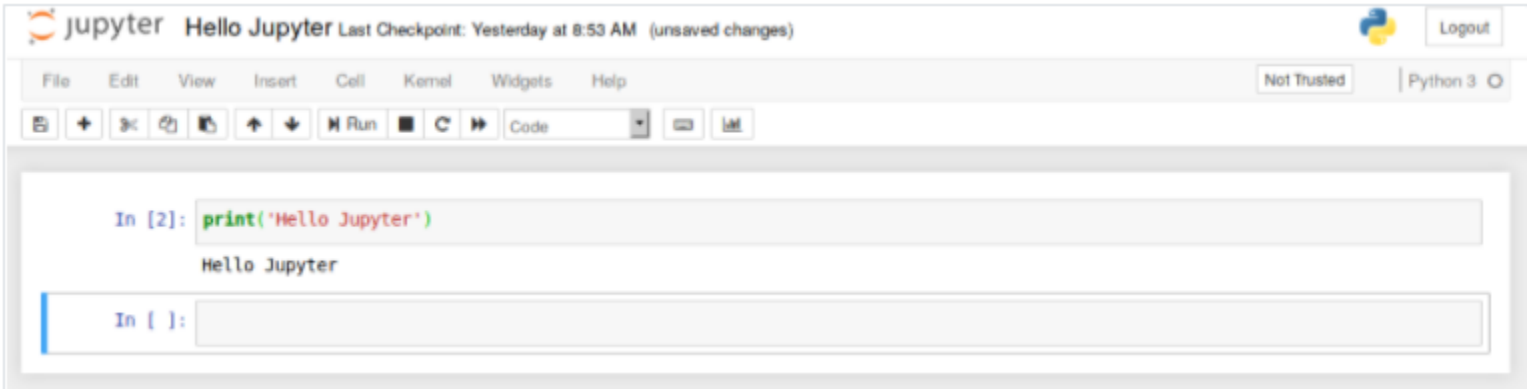
Shell

```
$ pip install jupyter
```

Shell

```
$ jupyter notebook
```

URL: http://localhost:8888/tree



Python Libraries for Data Science

Many popular Python toolboxes/libraries:

- NumPy
- SciPy
- Pandas
- SciKit-Learn

Visualization libraries

- matplotlib
- Seaborn

and many more …

Python Libraries for Data Science

NumPy:

- introduces objects for multidimensional arrays and matrices, as well as functions that allow to easily perform advanced mathematical and statistical operations on those objects
- provides vectorization of mathematical operations on arrays and matrices which significantly improves the performance
- many other python libraries are built on NumPy

Link:

http://www.numpy.org/

Python Libraries for Data Science

SciPy:

- collection of algorithms for linear algebra, differential equations, numerical integration, optimization, statistics and more
- part of SciPy Stack
- built on NumPy

Link:
https://www.scipy.org/scipylib/

Python Libraries for Data Science

Pandas:

- adds data structures and tools designed to work with table-like data (similar to Series and Data Frames in R)
- provides tools for data manipulation: reshaping, merging, sorting, slicing, aggregation etc.
- allows handling missing data

Link: http://pandas.pydata.org/

Python Libraries for Data Science

SciKit-Learn:

- provides machine learning algorithms: classification, regression, clustering, model validation etc.

- built on NumPy, SciPy and matplotlib

Link: http://scikit-learn.org/

Python Libraries for Data Science

matplotlib:

- python 2D plotting library which produces publication quality figures in a variety of hardcopy formats
- a set of functionalities similar to those of MATLAB
- line plots, scatter plots, barcharts, histograms, pie charts etc.
- relatively low-level; some effort needed to create advanced visualization

Link: https://matplotlib.org/

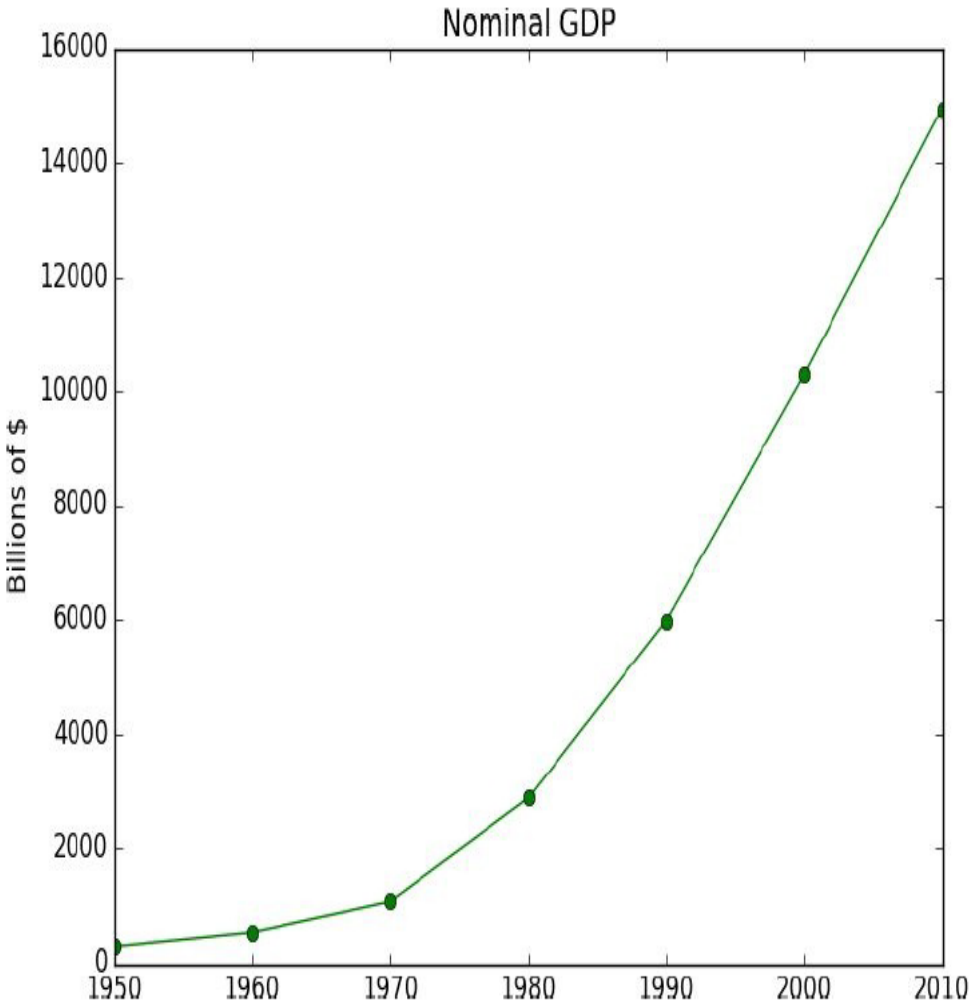
Visualizing Data

- There are two primary uses for data visualization:
- To *explore* data
- To *communicate* data

matplotlib

- matplotlib.pyplot module

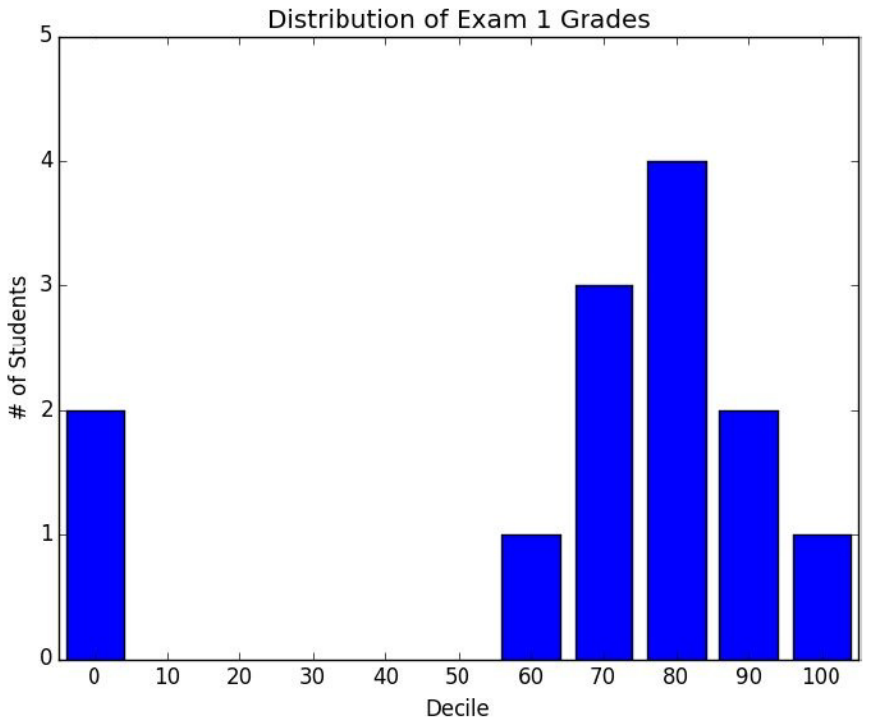
```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]

# create a line chart, years on x-axis, gdp on y-axis
plt.plot(years, gdp, color='green', marker='o', linestyle='solid'

# add a title
plt.title("Nominal GDP")

# add a label to the y-axis
plt.ylabel("Billions of $")
plt.show()
```



Using a bar chart for a histogram

- A bar chart is a good choice when you want to show how some quantity varies among some discrete set of items.
- A bar chart can also be a good choice for plotting histograms of bucketed numeric values, in order to visually explore how the values are distributed.

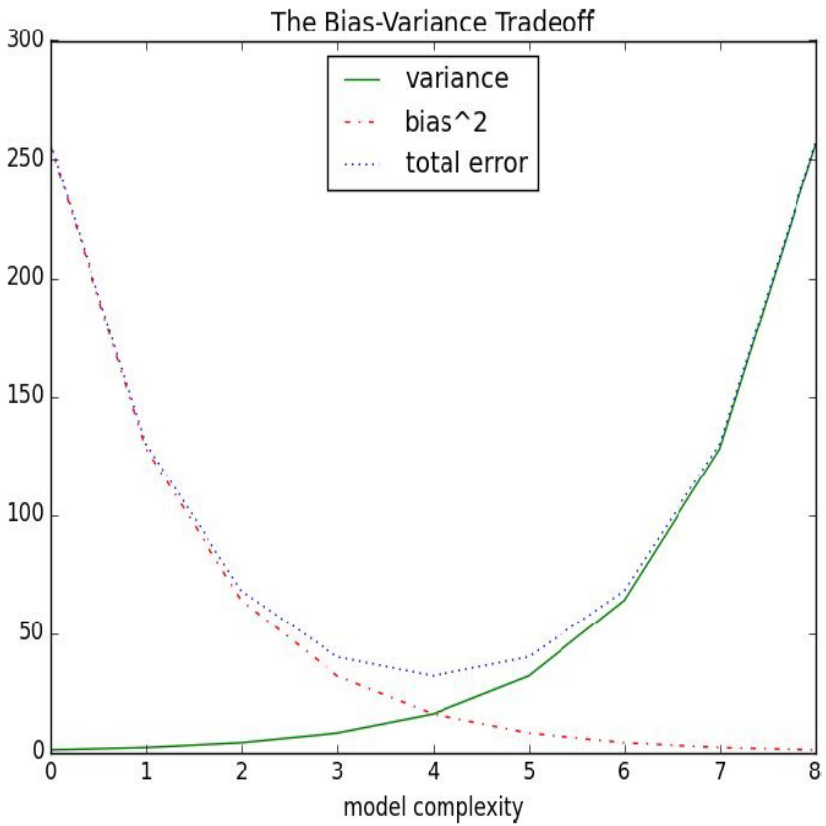
```
grades = [83,95,91,87,70,0,85,82,100,67,73,77,0]
decile = lambda grade: grade // 10 * 10
histogram = Counter(decile(grade) for grade in grades)

plt.bar([x - 4 for x in histogram.keys()], # shift each bar to the left by 4
        histogram.values(),               # give each bar its correct height
        8)                                # give each bar a width of 8

plt.axis([-5, 105, 0, 5])                 # x-axis from -5 to 105,
                                          # y-axis from 0 to 5

plt.xticks([10 * i for i in range(11)])   # x-axis labels at 0, 10, ..., 100
plt.xlabel("Decile")
plt.ylabel("# of Students")
plt.title("Distribution of Exam 1 Grades")
plt.show()
```



Line Charts

- As we saw already, we can make line charts using plt.plot(). These are a good choice for showing trends.

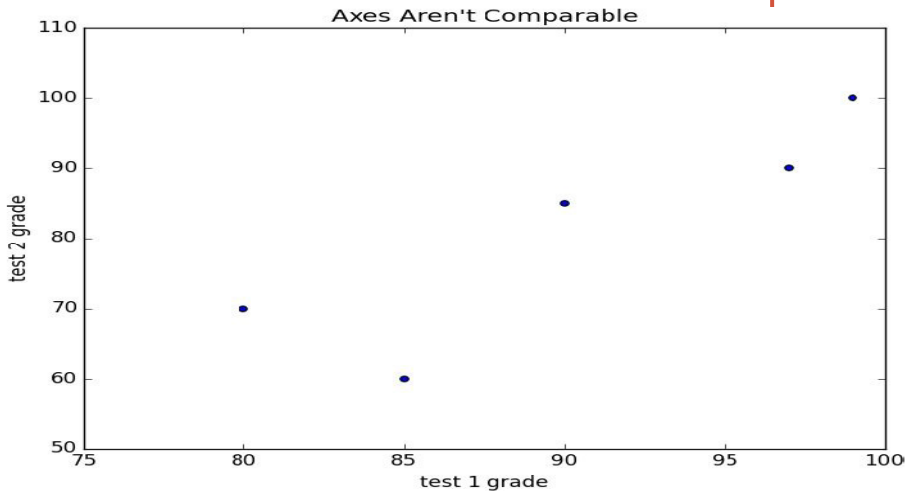
```
variance     = [1, 2, 4, 8, 16, 32, 64, 128, 256]
bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]
total_error  = [x + y for x, y in zip(variance, bias_squared)]
xs = [i for i, _ in enumerate(variance)]

# we can make multiple calls to plt.plot
# to show multiple series on the same chart
plt.plot(xs, variance,     'g-',  label='variance')    # green solid line
plt.plot(xs, bias_squared, 'r-.', label='bias^2')      # red dot-dashed line
plt.plot(xs, total_error,  'b:',  label='total error') # blue dotted line

# because we've assigned labels to each series
# we can get a legend for free
# loc=9 means "top center"
plt.legend(loc=9)
plt.xlabel("model complexity")
plt.title("The Bias-Variance Tradeoff")
plt.show()
```



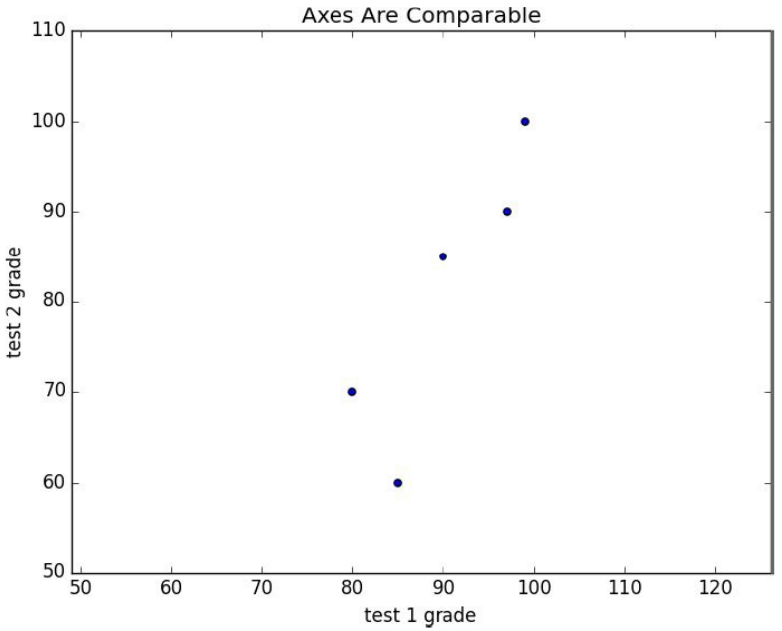
Scatterplots

- A scatterplot is the right choice for visualizing the relationship between two paired sets of data.
- If you're scattering comparable variables, you might get a misleading picture if you let matplotlib choose the scale.

```
test_1_grades = [ 99, 90, 85, 97, 80]
test_2_grades = [100, 85, 60, 90, 70]

plt.scatter(test_1_grades, test_2_grades)
plt.title("Axes Aren't Comparable")
plt.xlabel("test 1 grade")
plt.ylabel("test 2 grade")
plt.show()
```



- If we include a call to plt.axis("equal"), the plot more accurately shows that most of the variation occurs on test 2.



More tools

- **seaborn** is built on top of **matplotlib** and allows to easily produce prettier (and more complex) visualizations.
- **D3.js** is a JavaScript library for producing sophisticated interactive visualizations for the web.
- **Bokeh** is a newer library that brings D3-style visualizations into Python.
- **ggplot** is a Python port of the popular R library **ggplot2**, which is widely used for creating “publication quality” charts and graphics.

What is pandas ?

- ***Pandas*** is Python package for **data analysis**.
- It Provides built-in data structures which simplify the manipulation and analysis of data sets.
- Pandas is easy to use and powerful, but “with great power comes great responsibility”
- http://pandas.pydata.org/pandas-docs/stable/
- Pandas Basics
  - Series
  - DataFrame
  - Creating a DataFrame from a dict
  - Get a column, Get rows

Pandas: Essential Concepts

- A **Series** is a named Python list (dict with list as value).

```
{ 'grades' : [50,90,100,45] }
```

- A **DataFrame** is a collection of Series (dict of series):

```
{  { 'names' : ['bob','ken','art','joe']}
    { 'grades' : [50,90,100,45] }
  }
```

- Reading Data with Pandas
  - Pandas can read a variety of formats!

Loading Python Libraries

In [ ]:
```
#Import Python Libraries
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib as mpl
import seaborn as sns
```

Reading data using pandas

In [ ]:
```
#Read csv file
df =
pd.read_csv("http://rcs.bu.edu/examples/python/data_analysis/
Salaries.csv")
```

Note: The above command has many optional arguments to fine-tune the data import process.

There is a number of pandas commands to read other data formats:

```
pd.read_excel('myfile.xlsx',sheet_name='Sheet1',
index_col=None, na_values=['NA'])
pd.read_stata('myfile.dta')
pd.read_sas('myfile.sas7bdat')
pd.read_hdf('myfile.h5','df')
```

Exploring data frames

In [3]:
```
#List first 5 records
df.head()
```

Out[

	rank	discipline	phd	service	sex	salary
0	Prof	B	56	49	Male	186960
1	Prof	A	12	6	Male	93000
2	Prof	A	23	20	Male	110515
3	Prof	A	40	31	Male	131205
4	Prof	B	20	18	Male	104800

Data Frame data types

Pandas Type	Native Python Type	Description
object	string	The most general dtype. Will be assigned to your column if column has mixed types (numbers and strings).
int64	int	Numeric characters. 64 refers to the memory allocated to hold this character.
float64	float	Numeric characters with decimals. If a column contains numbers and NaNs(see below), pandas will default to float64, in case your missing value has a decimal.
datetime64, timedelta[ns]	N/A (but see the datetime module in Python's standard library)	Values meant to hold time data. Look into these for time series experiments.

Data Frames methods

Unlike attributes, python methods have *parenthesis*.
All attributes and methods can be listed with a *dir()* function:
`dir(df)`

df.method()	description
head([n]), tail([n])	first/last n rows
describe()	generate descriptive statistics (for numeric columns only)
max(), min()	return max/min values for all numeric columns
mean(), median()	return mean/median values for all numeric columns
std()	standard deviation
sample([n])	returns a random sample of the data frame
dropna()	drop all the records with missing values

Data Frames *groupby* method

Using "group by" method we can:

- Split the data into groups based on some criteria
- Calculate statistics (or apply a function) to each group
- Similar to dplyr() function in R

In [ ]:
```
#Group data using rank
df_rank = df.groupby(['rank'])
```

In [ ]:
```
#Calculate mean value for each numeric column per each group
df_rank.mean()
```

rank	phd	service	salary
AssocProf	15.076923	11.307692	91786.230769
AsstProf	5.052632	2.210526	81362.789474
Prof	27.065217	21.413043	123624.804348

Data Frame: filtering

To subset the data we can apply Boolean indexing. This indexing is commonly known as a filter. For example if we want to subset the rows in which the salary value is greater than $120K:

In [ ]:
```
#Calculate mean salary for each professor rank:
df_sub = df[ df['salary'] > 120000 ]
```

Any Boolean operator can be used to subset the data:

> greater; >= greater or equal;

< less; <= less or equal;

== equal; != not equal;

Data Frames: Slicing

When selecting one column, it is possible to use single set of brackets, but the resulting object will be a Series (not a DataFrame):

In [ ]:
```
#Select column salary:
df['salary']
```

When we need to select more than one column and/or make the output to be a DataFrame, we should use double brackets:

In [ ]:
```
#Select column salary:
df[['rank','salary']]
```

Data Frames: method loc

If we need to select a range of rows, using their labels we can use method loc:

In [ ]:
```
#Select rows by their labels:
df_sub.loc[10:20,['rank','sex','salary']]
```

Out[ ]:

	rank	sex	salary
10	Prof	Male	128250
11	Prof	Male	134778
13	Prof	Male	162200
14	Prof	Male	153750
15	Prof	Male	150480
19	Prof	Male	150500

Data Frames: method iloc

If we need to select a range of rows and/or columns, using their positions we can use method iloc:

In [ ]:
```
#Select rows by their labels:
df_sub.iloc[10:20,[0, 3, 4, 5]]
```

Out[ ]:

	rank	service	sex	salary
26	Prof	19	Male	148750
27	Prof	43	Male	155865
29	Prof	20	Male	123683
31	Prof	21	Male	155750
35	Prof	23	Male	126933
36	Prof	45	Male	146856
39	Prof	18	Female	129000
40	Prof	36	Female	137000
44	Prof	19	Female	151768
45	Prof	25	Female	140096

Data Frames: Sorting

We can sort the data by a value in the column. By default the sorting will occur in ascending order and a new data frame is return.

In [ ]:
```
# Create a new data frame from the original sorted by the column Salary
df_sorted = df.sort_values( by ='service')
df_sorted.head()
```

Out[ ]:

	rank	discipline	phd	service	sex	salary
55	AsstProf	A	2	0	Female	72500
23	AsstProf	A	2	0	Male	85000
43	AsstProf	B	5	0	Female	77000
17	AsstProf	B	4	0	Male	92000
12	AsstProf	B	1	0	Male	88000

Missing Values

Missing values are marked as NaN

In [ ]:
```
# Read a dataset with missing values
flights =
pd.read_csv("http://rcs.bu.edu/examples/python/data_analysis/flights.csv")
```

In [ ]:
```
# Select the rows that have at least one missing value
flights[flights.isnull().any(axis=1)].head()
```

Out[ ]:

	year	month	day	dep_time	dep_delay	arr_time	arr_delay	carrier	tailnum	flight	origin	dest	air_time	distance	hour	minute
330	2013	1	1	1807.0	29.0	2251.0	NaN	UA	N31412	1228	EWR	SAN	NaN	2425	18.0	7.0
403	2013	1	1	NaN	NaN	NaN	NaN	AA	N3EHAA	791	LGA	DFW	NaN	1389	NaN	NaN
404	2013	1	1	NaN	NaN	NaN	NaN	AA	N3EVAA	1925	LGA	MIA	NaN	1096	NaN	NaN
855	2013	1	2	2145.0	16.0	NaN	NaN	UA	N12221	1299	EWR	RSW	NaN	1068	21.0	45.0
858	2013	1	2	NaN	NaN	NaN	NaN	AA	NaN	133	JFK	LAX	NaN	2475	NaN	NaN

Aggregation Functions in Pandas

agg() method are useful when multiple statistics are computed per column:

```
In [ ]: flights[['dep_delay','arr_delay']].agg(['min','mean','max'])
```

Out[ ]:

	dep_delay	arr_delay
min	-16.000000	-62.000000
mean	9.384302	2.298675
max	351.000000	389.000000

Basic Descriptive Statistics

df.method()	description
describe	Basic statistics (count, mean, std, min, quantiles, max)
min, max	Minimum and maximum values
mean, median, mode	Arithmetic average, median and mode
var, std	Variance and standard deviation
sem	Standard error of mean
skew	Sample skewness
kurt	kurtosis

What is Numpy?

- Numpy, Scipy, and Matplotlib provide MATLAB-like functionality in python.
- Numpy Features:
  - Typed multidimentional arrays (matrices)
  - Fast numerical computations (matrix math)
  - High-level math functions
- Python does numerical computations slowly.
- 1000 x 1000 matrix multiply
  - Python triple loop takes > 10 min.
  - Numpy takes ~0.03 seconds

NumPy Overview

1. Arrays
2. Shaping and transposition
3. Mathematical Operations
4. Indexing and slicing
5. Broadcasting

Arrays

Structured lists of numbers.

- Vectors
- Matrices
- Images
- Tensors
- ConvNets

Arrays

Structured lists of numbers.

- **Vectors**
- **Matrices**
- Images
- Tensors
- ConvNets

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

Arrays

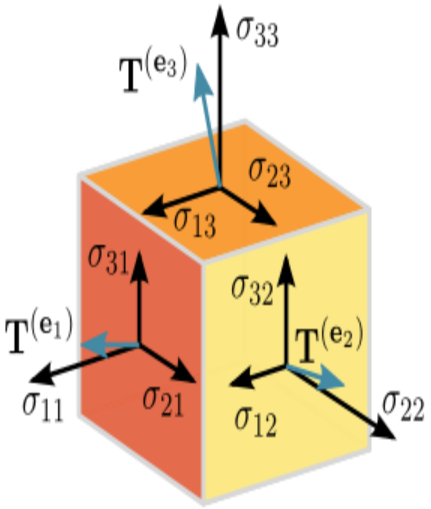
Structured lists of numbers.

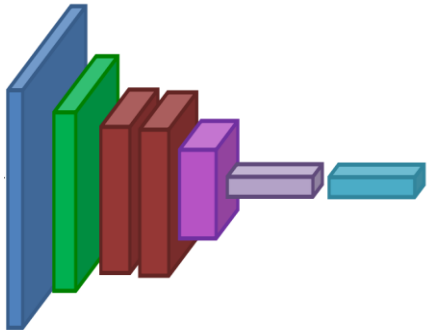
- Vectors
- Matrices
- **Images**
- Tensors
- ConvNets

Arrays

Structured lists of numbers.

- Vectors
- Matrices
- Images
- **Tensors**
- **ConvNets**





Arrays, Basic Properties

```
import numpy as np
a                                                    =
np.array([[1,2,3],[4,5,6]],dtype=np.float3
2)
print a.ndim, a.shape, a.dtype
```

1. Arrays can have any number of dimensions, including zero (a scalar).
2. Arrays are typed: np.uint8, np.int64, np.float32, np.float64
3. Arrays are dense. Each element of the array exists and has the same type.

Arrays, creation

- **np.ones, np.zeros**
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> np.ones((3,5),dtype=np.float32)
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]], dtype=float32)
```

```
>>> np.zeros((6,2),dtype=np.int8)
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=int8)
```

Arrays, creation

- np.ones, np.zeros
- **np.arange**
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> np.arange(1334,1338)
array([1334, 1335, 1336, 1337])
```

```
>>> np.random.random((10,3))
array([[ 0.61481644,  0.55453657,  0.04320502],
       [ 0.08973085,  0.25959573,  0.27566721],
       [ 0.84375899,  0.2949532 ,  0.29712833],
       [ 0.44564992,  0.37728361,  0.29471536],
       [ 0.71256698,  0.53193976,  0.63061914],
       [ 0.03738061,  0.96497761,  0.01481647],
       [ 0.09924332,  0.73128868,  0.22521644],
       [ 0.94249399,  0.72355378,  0.94034095],
       [ 0.35742243,  0.91085299,  0.15669063],
       [ 0.54259617,  0.85891392,  0.77224443]])
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> A = np.ones((2,3))
>>> B = np.zeros((4,3))
>>> np.concatenate([A,B])
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>>
```

```
>>> A = np.ones((4,1))
>>> B = np.zeros((4,2))
>>> np.concatenate([A,B], axis=1)
array([[ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.]])
```

Saving and loading arrays

```
np.savez('data.npz', a=a)
data = np.load('data.npz')
a = data['a']
```

1. NPZ files can hold multiple arrays
2. np.savez_compressed similar.

Saving and Loading Images

SciPy: `skimage.io.imread`, `skimage.io.imsave`

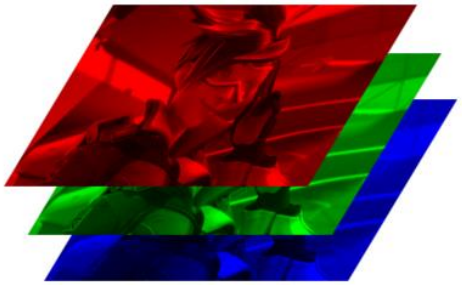
height x width x RGB

PIL / Pillow: `PIL.Image.open`, `Image.save`

width x height x RGB

OpenCV: `cv2.imread`, `cv2.imwrite`

height x width x BGR



Mathematical operators

- **Arithmetic operations are element-wise**
- Logical operator return a bool array
- In place operations modify the array

```
>>> a
array([1, 2, 3])
>>> b
array([ 4,  4, 10])
>>> a * b
array([ 4,  8, 30])
```

```
>>> a
array([[ 4, 15],
       [20, 75]])
>>> b
array([[ 2,  5],
       [ 5, 15]])
>>> a /= b
>>> a
array([[2, 3],
       [4, 5]])
```

```
>>> a
array([[ 0.93445601,  0.42984044,  0.12228461],
       [ 0.06239738,  0.76019703,  0.11123116],
       [ 0.14617578,  0.90159137,  0.89746818]])
>>> a > 0.5
array([[ True, False, False],
       [False,  True, False],
       [False,  True,  True]], dtype=bool)
```

Math, upcasting

Just as in Python and Java, the result of a math operator is cast to the more general or precise datatype.

uint64 + uint16 => uint64

float32 / int32 => float32

Warning: upcasting does not prevent overflow/underflow. You must manually cast first.

Use case: images often stored as uint8. You should convert to float32 or float64 before doing math.

Math, universal functions

Also called ufuncs

Element-wise

Examples:

- np.exp
- np.sqrt
- np.sin
- np.cos
- np.isnan

```
>>> a
array([[ 1,  4],
       [ 9, 16],
       [25, 36]])
>>> np.sqrt(a)
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
```

Indexing

```
x[0,0]    # top-left element
x[0,-1]   # first row, last column
x[0,:]    # first row (many entries)
x[:,0]    # first column (many entries)
```

Notes:

- Zero-indexing
- Multi-dimensional indices are comma-separated (i.e., a tuple)

Python Slicing

Syntax: start:stop:step

```
a = list(range(10))
a[:3] # indices 0, 1, 2
a[-3:] # indices 7, 8, 9
a[3:8:2] # indices 3, 5, 7
a[4:1:-1] # indices 4, 3, 2 (this one
is tricky)
```

Axes

```
a.sum() # sum all entries
a.sum(axis=0) # sum over rows
a.sum(axis=1) # sum over columns
a.sum(axis=1, keepdims=True)
```

1. Use the axis parameter to control which axis NumPy operates on
2. Typically, the axis specified will disappear, keepdims keeps all dimensions

Broadcasting

```
a = a + 1 # add one to every element
```

When operating on multiple arrays, broadcasting rules are used.

Each dimension must match, from right-to-left

1. Dimensions of size 1 will broadcast (as if the value was repeated).
2. Otherwise, the dimension must have the same shape.
3. Extra dimensions of size 1 are added to the left as needed.

WORKING WITH DATA

Getting Data

- **Finding APIs**
- If you need data from a specific site, look for a developers or API section of the site for details, and try searching the Web for "*python __ api*" to find a library.
- Twitter is a fantastic source of data to work with. And you can get access to its data through its API. To interact with the Twitter APIs one can use the *Twython library (pip install twython).*

```
from twython import Twython

twitter = Twython(CONSUMER_KEY, CONSUMER_SECRET)

# search for tweets containing the phrase "data science"
for status in twitter.search(q='"data science"')["statuses"]:
    user = status["user"]["screen_name"].encode('utf-8')
    text = status["text"].encode('utf-8')
    print user, ":", text
    print
```

```
haithemnyc: Data scientists with the technical savvy &amp;amp; analytical chops to
derive meaning from big data are in demand. http://t.co/HsF9Q0dShP

RPubsRecent: Data Science http://t.co/6hcHUz2PHM

spleonard1: Using #dplyr in #R to work through a procrastinated assignment for
@rdpeng in @coursera data science specialization.  So easy and Awesome.
```

Exploring Data

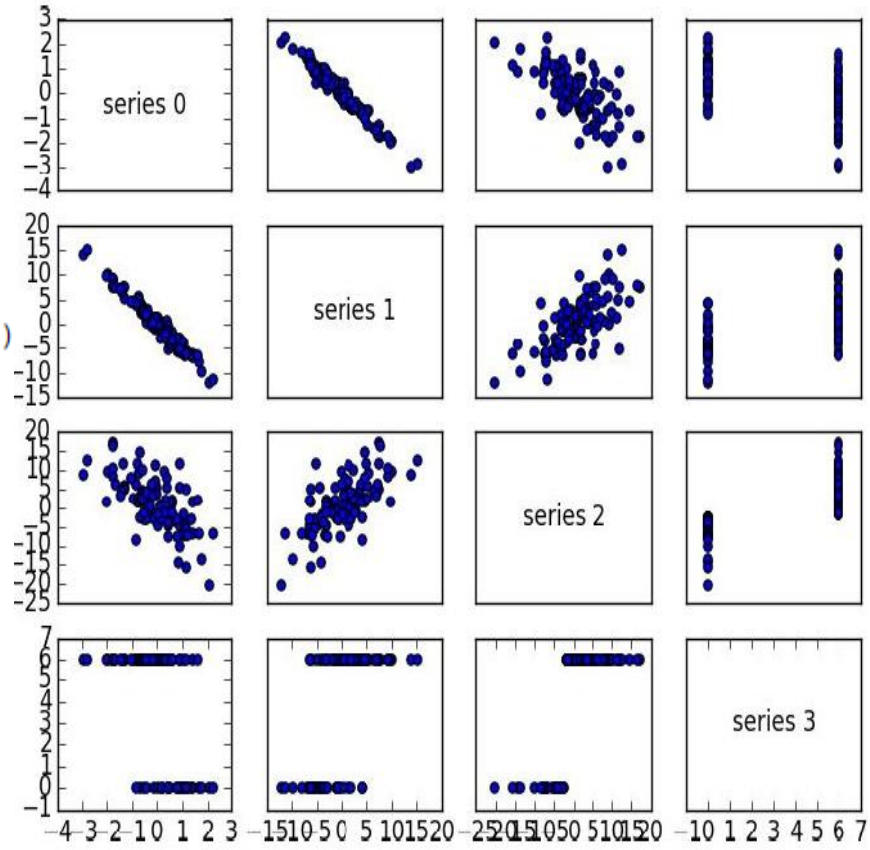
- With many dimensions, you'd like to know how all the dimensions relate to one another. A simple approach is to look at the correlation matrix, in which the entry in row i and column j is the correlation between the ith dimension and the jth dimension of the data.

```
def correlation_matrix(data):
    """returns the num_columns x num_columns matrix whose (i, j)th entry
    is the correlation between columns i and j of data"""

    _, num_columns = shape(data)

    def matrix_entry(i, j):
        return correlation(get_column(data, i), get_column(data, j))

    return make_matrix(num_columns, num_columns, matrix_entry)
```

Scatterplot Matrix

- A more visual approach (if you don't have too many dimensions) is to make a scatterplot matrix showing all the pairwise scatterplots. To do that one needs to use **plt.subplots(),** which allows to create subplots of the chart.

```
import matplotlib.pyplot as plt

_, num_columns = shape(data)
fig, ax = plt.subplots(num_columns, num_columns)

for i in range(num_columns):
    for j in range(num_columns):

        # scatter column_j on the x-axis vs column_i on the y-axis
        if i != j: ax[i][j].scatter(get_column(data, j), get_column(data, i))

        # unless i == j, in which case show the series name
        else: ax[i][j].annotate("series " + str(i), (0.5, 0.5),
                                xycoords='axes fraction',
                                ha="center", va="center")

        # then hide axis labels except left and bottom charts
        if i < num_columns - 1: ax[i][j].xaxis.set_visible(False)
        if j > 0: ax[i][j].yaxis.set_visible(False)

# fix the bottom right and top left axis labels, which are wrong because
# their charts only have text in them
ax[-1][-1].set_xlim(ax[0][-1].get_xlim())
ax[0][0].set_ylim(ax[0][1].get_ylim())

plt.show()
```



Cleaning and Munging

- Real-world data is dirty. Often you'll have to do some work on it before you can use it.
- We have to convert strings to floats or ints before we can use them.

```
6/20/2014,AAPL,90.91
6/20/2014,MSFT,41.68
6/20/3014,FB,64.5
6/19/2014,AAPL,91.86
6/19/2014,MSFT,n/a
6/19/2014,FB,64.34
```

```
import dateutil.parser
data = []

with open("comma_delimited_stock_prices.csv", "rb") as f:
    reader = csv.reader(f)
    for line in parse_rows_with(reader, [dateutil.parser.parse, None, float]):
        data.append(line)

for row in data:
    if any(x is None for x in row):
        print row
```

Rescaling

- Many techniques are sensitive to the *scale* of your data. For example, imagine that you have a data set consisting of the heights and weights of hundreds of data scientists, and that you are trying to identify *clusters* of body sizes.
- To start with, we'll need to compute the mean and the standard_deviation for each column and then use them to create a new data matrix.

```
def scale(data_matrix):
    """returns the means and standard deviations of each column"""
    num_rows, num_cols = shape(data_matrix)
    means = [mean(get_column(data_matrix,j))
             for j in range(num_cols)]
    stdevs = [standard_deviation(get_column(data_matrix,j))
              for j in range(num_cols)]
    return means, stdevs
```

```
def rescale(data_matrix):
    """rescales the input data so that each column
    has mean 0 and standard deviation 1
    leaves alone columns with no deviation"""
    means, stdevs = scale(data_matrix)

    def rescaled(i, j):
        if stdevs[j] > 0:
            return (data_matrix[i][j] - means[j]) / stdevs[j]
        else:
            return data_matrix[i][j]

    num_rows, num_cols = shape(data_matrix)
    return make_matrix(num_rows, num_cols, rescaled)
```

Dimensionality Reduction

- Sometimes the “actual” (or useful) dimensions of the data might not correspond to the dimensions we have.
- Some of the more popular methods include:
  - Principal Components Analysis
  - Singular Value Decomposition
  - Non-Negative Matrix Factorization

```
# evaluate logistic regression model on raw data
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.linear_model import LogisticRegression
# define dataset
X, y = make_classification(n_samples=1000, n_features=20,
n_informative=10, n_redundant=10, random_state=7)
# define the model
model = LogisticRegression()
# evaluate model
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
n_scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv,
n_jobs=-1)
# report performance
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

```
Accuracy: 0.824 (0.034)
```

MACHINE LEARNING

Models

- To creating and using models that are learned from data.
- Typically, our goal will be to use existing data to develop models that we can use to *predict* various outcomes for new data.
  - Predicting whether an email message is spam or not
  - Predicting whether a credit card transaction is fraudulent
  - Predicting which advertisement a shopper is most likely to click on
  - Predicting which football team is going to win the Super Bowl
- *Supervised models* - In which there is a set of data labeled with the correct answers to learn from,
- Unsupervised models - In which there are no such labels.

Defining Models

```
def split_data(data, prob):
    """split data into fractions [prob, 1 - prob]"""
    results = [], []
    for row in data:
        results[0 if random.random() < prob else 1].append(row)
    return results

def train_test_split(x, y, test_pct):
    data = zip(x, y)                                  # pair corresponding values
    train, test = split_data(data, 1 - test_pct)      # split the data set of pairs
    x_train, y_train = zip(*train)                    # magical un-zip trick
    x_test, y_test = zip(*test)
    return x_train, x_test, y_train, y_test

model = SomeKindOfModel()
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.33)
model.train(x_train, y_train)
performance = model.test(x_test, y_test)
```

Checking Correctness

- Given a set of labeled data and such a predictive model, every data point lies in one of four categories:
  - True positive: “This message is spam, and we correctly predicted spam.”
  - False positive (Type 1 Error): “This message is not spam, but we predicted spam.”
  - False negative (Type 2 Error): “This message is spam, but we predicted not spam.”
  - True negative: “This message is not spam, and we correctly predicted not spam.”

	Spam	not Spam
predict “Spam”	True Positive	False Positive
predict “Not Spam”	False Negative	True Negative

	leukemia	no leukemia	total
“Luke”	70	4,930	5,000
not “Luke”	13,930	981,070	995,000
total	14,000	986,000	1,000,000

```
def accuracy(tp, fp, fn, tn):
    correct = tp + tn
    total = tp + fp + fn + tn
    return correct / total

print accuracy(70, 4930, 13930, 981070)     # 0.98114
```

```
def precision(tp, fp, fn, tn):
    return tp / (tp + fp)

print precision(70, 4930, 13930, 981070)    # 0.014
```

```
def recall(tp, fp, fn, tn):
    return tp / (tp + fn)

print recall(70, 4930, 13930, 981070)       # 0.005
```

```
def f1_score(tp, fp, fn, tn):
    p = precision(tp, fp, fn, tn)
    r = recall(tp, fp, fn, tn)

    return 2 * p * r / (p + r)
```

Simple Linear Regression

- **def** predict(alpha, beta, x_i):

  return beta * x_i + alpha

- **def** error(alpha, beta, x_i, y_i):

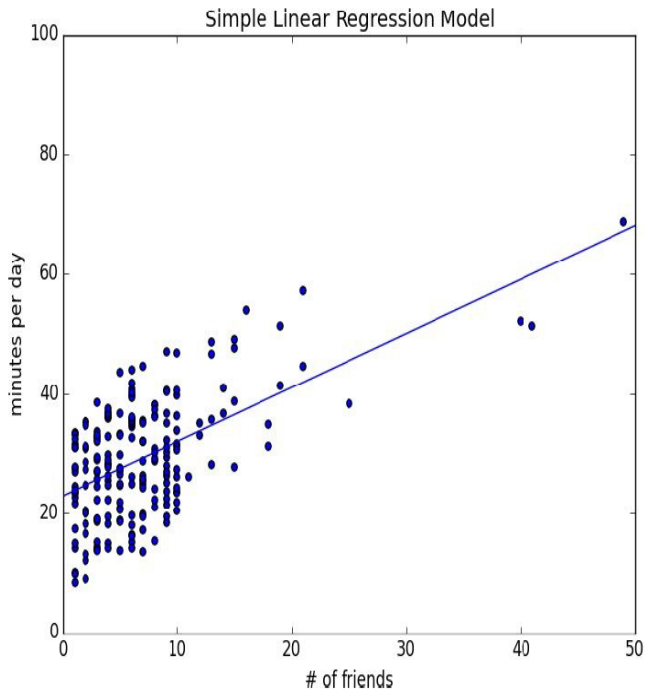
  *"""the error from predicting beta * x_i + alpha when the actual value is y_i"""*

  return y_i - predict(alpha, beta, x_i)

- **def** sum_of_squared_errors(alpha, beta, x, y):

  return sum(error(alpha, beta, x_i, y_i) ** 2

  for x_i, y_i **in** zip(x, y))



- **def** least_squares_fit(x, y):

  """given training values for x and y,find the least-squares values of alpha and beta"""

  beta = correlation(x, y) * standard_deviation(y) / standard_deviation(x)

  alpha = mean(y) - beta * mean(x)

  return alpha, beta

- alpha, beta = least_squares_fit(num_friends_good, daily_minutes_good)

References

- Python Homepage
  - http://www.python.org
- Python Tutorial
  - http://docs.python.org/tutorial/
- Python Documentation
  - http://www.python.org/doc
- Python Library References
  - http://docs.python.org/release/2.5.2/lib/lib.html
- Python Add-on Packages:
  - http://pypi.python.org/pypi
- Data Science from Scratch – First principles with Python, Joel Grus, O'Reilly.
- Python Data Science Handbook – Essential Tools for Working with Data, Jake VanderPlas, O'Reilly.