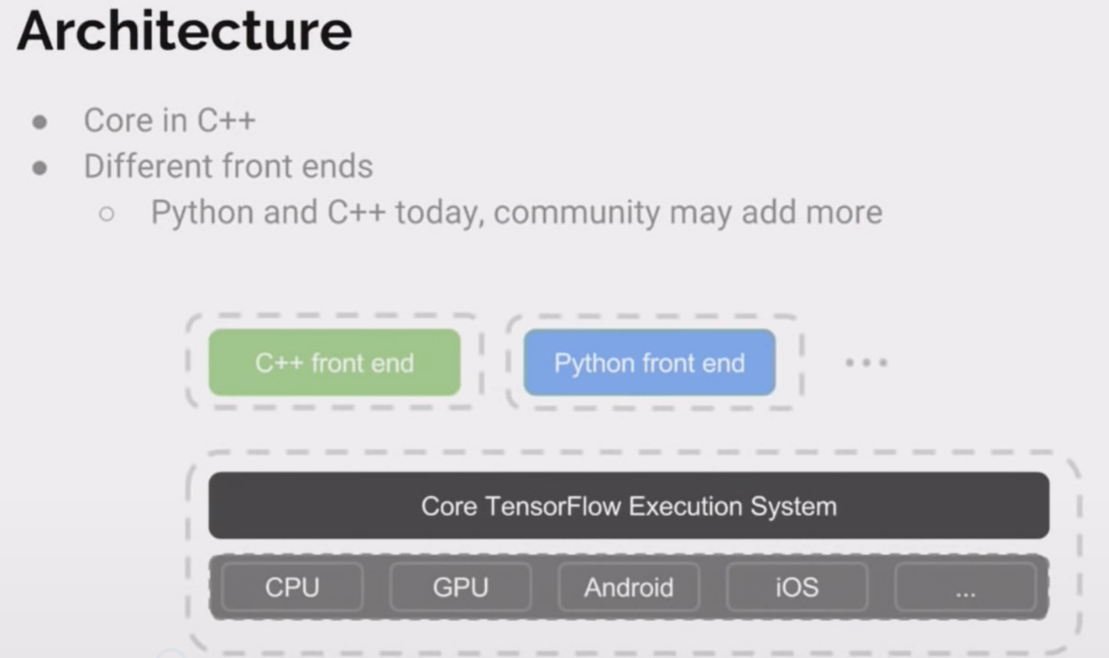
<https://github.com/ubarredo/LearnPythonTheHardWay>

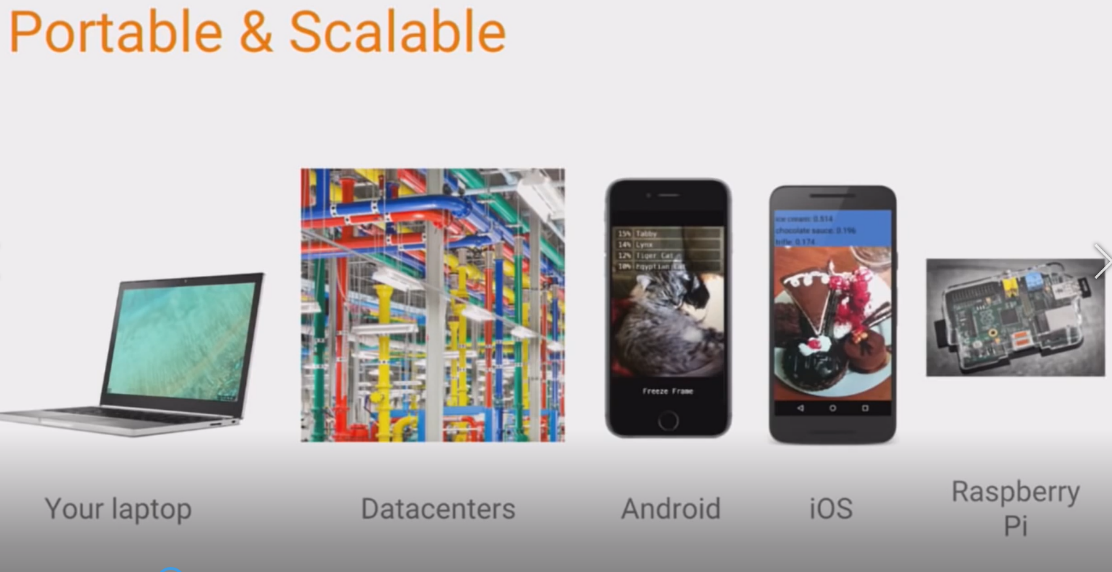
Python is a remarkably powerful dynamic programming language that is used in a wide variety of application domains. Python is often compared to Tcl, Perl, Ruby, Scheme or Java. Some of its key distinguishing features include:

* very clear, readable syntax
* strong introspection capabilities
* intuitive object orientation
* natural expression of procedural code
* full modularity, supporting hierarchical packages
* exception-based error handling
* very high level dynamic data types
* extensive standard libraries and third party modules for virtually every task
* extensions and modules easily written in C, C++ (or Java for Jython, or .NET languages for IronPython)
* embeddable within applications as a scripting interface

The open-source [Anaconda Distribution](https://docs.anaconda.com/anaconda/) is the easiest way to perform Python/R data science and machine learning on Linux, Windows, and Mac OS X. With over 11 million users worldwide, it is the industry standard for developing, testing, and training on a single machine, enabling *individual data scientists* to:

* Quickly download 1,500+ Python/R data science packages
* Manage libraries, dependencies, and environments with [Conda](https://conda.io/docs/)
* Develop and train machine learning and deep learning models with [scikit-learn](https://scikit-learn.org/stable/), [TensorFlow](https://www.tensorflow.org/), and [Theano](https://pypi.org/project/Theano/)
* Analyze data with scalability and performance with [Dask](https://dask.org/), [NumPy](http://www.numpy.org/), [pandas](https://pandas.pydata.org/), and [Numba](http://numba.pydata.org/)
* Visualize results with [Matplotlib](https://matplotlib.org/), [Bokeh](https://bokeh.pydata.org/en/latest/), [Datashader](http://datashader.org/), and [Holoviews](http://holoviews.org/)





Anaconda installation in linux

bash anaconda-1.6.0-linux-x86\_65.sh

/click yes for licence agreement, then click enter, start to install the packages

ls -la

which python

which ipythnon

/ipython notebookcreate a new python3 notebook/

Anaconda navigator /we can open jupyter notebook, ipython,spyder from navigator.

Spyder is a powerful scientific environment written in Python, for Python, and designed by and for scientists, engineers and data analysts. It offers a unique combination of the advanced editing, analysis, debugging, and profiling functionality of a comprehensive development tool with the data exploration, interactive execution, deep inspection, and beautiful visualization capabilities of a scientific package.

If you installed git, you can open git bash terminal and type the following command.

Peiwu@LAPTOP-7V8SU842 MINGW64 ~

$ git config --global user.name "Peiwu"

Peiwu@LAPTOP-7V8SU842 MINGW64 ~

$ git config --global user.email "pwqin@sz.tsinghua.edu.cn"

Peiwu@LAPTOP-7V8SU842 MINGW64 ~

$ git config --global core.editor "Atom --wait"

mkdir class\_work

cd class\_work

pwd

Absolute and relative directory, .. and . directory

ls

ls -a /list all items including current and upper directory

git –help add /find help about add

git init

git help add /open a webpage

/we can see .git/ now after git init

ls .git /see the internal file organization of git

ls .git/objects/\*

atom our\_paper.txt /type ‘this is our paper’

cd .git /enter git directory

git add our\_paper.txt

git status /we can see that new file our\_paper.txt was added.

ls .git/objects/\* /we see a new file that has a lone name

ls .git/objects/ /9e is the name of the new data: our\_paper.txt

ls .git/objects/9e /we can see the name of file in 9e folder

git commit /make the operation effective

/copy a folder from github and will learn pandas module from this tutorial

**Using IPython, an enhanced Python interpreter**

The stuff above works in the default python interpreter that you can get in the command line, but there are things that you'd like to be able to do, such as enter more than one line of code at a time, or perhaps look back through your history of commands. The enhanced **IPython** interpreter provides these perks and more, and it's built into Canopy.

20+30

Out[1]: 50

20-23

Out[2]: -3

13/5

Out[3]: 2.6

13.5/4.2

Out[4]: 3.214285714285714

12//5

Out[5]: 2

31\*4

Out[6]: 124

3\*\*3

Out[7]: 27

2\*2\*2

Out[8]: 8

2\*\*8 /8 bit image

Out[9]: 256

2\*\*16 /16 bit image

Out[10]: 65536

3\*4-5+6/2

Out[11]: 10.0

myVariableName=30 /define variable, first letter is small, following word start with capital letter

myVariableName /python is letter sensitive

Out[13]: 30

myVariableName+20

Out[14]: 50

myVariableName\*\*2

Out[15]: 900

In [9]: i=42

In [10]: f=3.14159

In [11]: sum=i+f

In [12]: diff=i-f

In [13]: prof=i\*f

In [14]: quo=i/f

In [15]: pow=i\*\*f

The Python input（） function is used to read a string from standard input such as keyboard. This way a programmer is able to include user inserted data into a program. Let's start with a simple example using python script to ask for an user name.

value=int(input('Enter the integer value:'))

Enter the integer value:30

/prompt for input

value

Out[17]: 30

In [4]: inp=input("Enter hours: ")

Enter hours: 10

In [5]: hours=float(inp)

In [6]: inp2=input("Enter rate: ")

Enter rate: 20

In [8]: rate=float(inp2)

In [9]: if hours<=40:

...: pay=hours\*rate

...: else:

...: pay=hours\*40 + (rate\*1.5\*(hours-40))

...: print(pay)

200.0

In [38]: print('variable can be interpolated as strings here %s and here %s' %(i,f))

variable can be interpolated as strings here 42 and here 3.14159

In [4]: print('resolu\\tio\\n')

resolu\tio\n

Regular expressions use the backslash character ('\') to indicate special forms or to allow special characters to be used without invoking their special meaning.

In Python strings, **the backslash "\"** is a **special character**, also called the "**escape**" character. It is used in representing certain whitespace characters: "\t" is a tab, "\n" is a newline, and "\r" is a carriage return.

Carriage return moves the position of the [cursor](https://en.wikipedia.org/wiki/Cursor_(computers)) to the first position on the same line.

In [2]: print('something that\r is not working')

is not working

In [3]: print('something that\t is not working')

something that is not working

In [24]: name='Peiwu Qin'

In [27]: last\_name=name[6:]

In [28]: last\_name

Out[28]: 'Qin'

In [29]: integer\_text='5'

In [8]: type(integer\_text)

Out[8]: str

In [9]: integer\_text = 5

In [10]: type(integer\_text)

Out[10]: int

In [30]: decimal\_text='3.14'

In [31]: print(integer\_text+decimal\_text)

53.14

In [32]: integer=int(integer\_text)

In [33]: integer

Out[33]: 5

In [34]: decimal=float(decimal\_text)

In [35]: decimal

Out[35]: 3.14

In [40]: n=int(input("Number?"))

Number?5

if n < 0:

print("the absolute value of", n, " is", -n)

In [43]: n=int(input("Number?"))

Number?-5

In [44]: if n < 0:

...: print("the absolute value of", n, " is", -n)

...:

the absolute value of -5 is 5

pow(2,3)

Out[18]: 8

Power is the builtin function.

dir(\_\_builtins\_\_)

Out[19]:

['ArithmeticError',

'AssertionError',

'AttributeError',

'BaseException',

….]

help(pow) /search for help.im

Help on built-in function pow in module builtins:

pow(x, y, z=None, /)

Equivalent to x\*\*y (with two arguments) or x\*\*y

% z (with three arguments)

Some types, such as ints, are able to use a more

efficient algorithm when

invoked using the three argument form.

In [22]: import math

In [23]: math. /click tab to show all the functions in this module

acos() atan() copysign() e

acosh() atan2() cos() erf()

asin() atanh() cosh() erfc() >

asinh() ceil() degrees() exp()

a='I am a single quoted string don\'t'

a

Out[24]: "I am a single quoted string don't"

len(a)

Out[25]: 33

b='Peiwu'

len(b)

Out[27]: 5

b='Peiwu '

len(b)

Out[29]: 7

In [1]: a='Hellow'

In [2]: b=5

In [3]: print(a\*10)

HellowHellowHellowHellowHellowHellowHellowHellowHellowHellow

In [4]: print(a\*b)

HellowHellowHellowHellowHellow

In [6]: print(a+b)

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-6-85111a3ab660> in <module>

----> 1 print(a+b)

TypeError: can only concatenate str (not "int") to str

In [7]: print(a+str(b))

/list introduction

names=[]

In [9]: names

Out[9]: []

In [10]: names=["yuanxi","yifeng","wei"]

In [11]: names

Out[11]: ['yuanxi', 'yifeng', 'wei']

/position of element in list, index starts from 0

In [12]: names[0]

Out[12]: 'yuanxi'

In [13]: names[1]

Out[13]: 'yifeng'

In [14]: names[-1]

Out[14]: 'wei'

In [15]: names[-2]

Out[15]: 'yifeng'

In [16]: names.append('yihang')

In [17]: names

Out[17]: ['yuanxi', 'yifeng', 'wei', 'yihang']

In [18]: age=[23,22,24,25]

In [19]: names.extend(age)

In [20]: names

Out[20]: ['yuanxi', 'yifeng', 'wei', 'yihang', 23, 22, 24, 25]

/list can contain string and values

In [21]: names.remove('wei')

In [22]: names

Out[22]: ['yuanxi', 'yifeng', 'yihang', 23, 22, 24, 25]

In [23]: print(names)

['yuanxi', 'yifeng', 'yihang', 23, 22, 24, 25]

In [24]: print(names+age)

['yuanxi', 'yifeng', 'yihang', 23, 22, 24, 25, 23, 22, 24, 25]

In [25]: len(names)

Out[25]: 7

In [30]: names

Out[30]: ['yuanxi', 'yifeng', 'yihang', 23, 22, 24, 25]

In [31]: names[2:4]

Out[31]: ['yihang', 23]

In [32]: names[2:]

Out[32]: ['yihang', 23, 22, 24, 25]

In [33]: names[:5]

Out[33]: ['yuanxi', 'yifeng', 'yihang', 23, 22]

In [34]: names[0:7:2]

Out[34]: ['yuanxi', 'yihang', 22, 25]

In [35]: names[-5:]

Out[35]: ['yihang', 23, 22, 24, 25]

In [36]: names[:]

Out[36]: ['yuanxi', 'yifeng', 'yihang', 23, 22, 24, 25]

In [37]: names[0:7:3]

Out[37]: ['yuanxi', 23, 25]

In [38]: names[::3]

Out[38]: ['yuanxi', 23, 25]

In [39]: names[::-3]

Out[39]: [25, 23, 'yuanxi'] /print backward

In [40]: names.pop

Out[40]: <function list.pop(index=-1, /)>

In [41]: names.pop() /last element was removed

Out[41]: 'wei'

In [26]: mylist=[0,1,2,3,4,5,6,7,8,9]

In [27]: mylist

Out[27]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [28]: mylist[4:8]

Out[28]: [4, 5, 6, 7]

In [29]: mylist[5:9]

Out[29]: [5, 6, 7, 8]

In [44]: noLi=[0]

In [45]: noLi

Out[45]: [0]

In [46]: noLi=4\*[0]

In [47]: noLi

Out[47]: [0, 0, 0, 0]

In [51]: noLi[2]=mice\_brain

In [54]: noLi[1]=rat\_brain

In [55]: noLi[3]=human\_brain

In [56]: noLi

Out[56]: [0, 20, 10, 500]

In [57]: noLi.sort()

In [58]: noLi

Out[58]: [0, 10, 20, 500]

In [59]: noLi.reverse()

In [60]: noLi

Out[60]: [500, 20, 10, 0]

In [61]: another\_noLi=sorted(noLi)

In [62]: another\_noLi

Out[62]: [0, 10, 20, 500]

In [63]: for x in another\_noLi:

...: print(x)

...:

0

10

20

500

In [64]: print('superheros' in noLi)

False

In [65]: print('500' in noLi)

False

In [66]: print(500 in noLi)

True

In [67]: trpA=('protein','TIM Barrel')

In [68]: trpA

Out[68]: ('protein', 'TIM Barrel')

In [69]: type(trpA)

Out[69]: tuple

In [70]: for i in trpA:

...: print(i)

...:

protein

TIM Barrel

In [71]: trpA[0]

Out[71]: 'protein'

In [72]: trpA[1]

Out[72]: 'TIM Barrel'

In [73]: trpA[1] ='RNA'

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-73-9288b0512645> in <module>

----> 1 trpA[1] ='RNA'

TypeError: 'tuple' object does not support item assignment

In [74]: trpA=list(trpA)

In [75]: trpA[1] ='RNA'

In [76]: trpA

Out[76]: ['protein', 'RNA']

In [77]: a=3

In [78]: b=5

In [79]: print('a=%s,b=%d'%(a,b))

In [80]: names = {'aaron':'hardin', 'mike':'lawson', 'peter':'combs'}

...: drinks = {'mike':'coffee', 'aisha':'caffeine', 'peter':'soda'}

...: wines = {'red':'cabernet','white':'pinot grigio',\

...: 'sparkling':'blanc de noirs', 'sticky':'muscato'}

...:

In [81]: print(names)

{'aaron': 'hardin', 'mike': 'lawson', 'peter': 'combs'}

In [82]: print(drinks)

{'mike': 'coffee', 'aisha': 'caffeine', 'peter': 'soda'}

In [83]: print(wines)

{'red': 'cabernet', 'white': 'pinot grigio', 'sparkling': 'blanc de noirs', 'sticky': 'mus

cato'}

In [84]: print(names['aaron'])

hardin

In [86]: print(drinks['peter'])

soda

In [87]: print(wines['sparkling'])

blanc de noirs

In [88]: drinks['debbie']='diet coke'

In [89]: drinks

Out[89]: {'mike': 'coffee', 'aisha': 'caffeine', 'peter': 'soda', 'debbie': 'diet coke'}

In [90]: del names['aaron']

In [91]: names

Out[91]: {'mike': 'lawson', 'peter': 'combs'}

In [92]: names['mike']='smith'

In [93]: names

Out[93]: {'mike': 'smith', 'peter': 'combs'}

In [94]: keys=wines.keys()

In [95]: values=wines.values()

In [96]: keys

Out[96]: dict\_keys(['red', 'white', 'sparkling', 'sticky'])

In [97]: values

Out[97]: dict\_values(['cabernet', 'pinot grigio', 'blanc de noirs', 'muscato'])

In [99]: wines

Out[99]:

{'red': 'cabernet',

'white': 'pinot grigio',

'sparkling': 'blanc de noirs',

'sticky': 'muscato'}

In [100]: for x in keys:

...: print('the category is',x,'and the variable is',wines[x])

...:

the category is red and the variable is cabernet

the category is white and the variable is pinot grigio

the category is sparkling and the variable is blanc de noirs

the category is sticky and the variable is muscato

In [101]: for x in sorted(keys):

...: print('the category is',x,'and the variable is',wines[x])

...:

the category is red and the variable is cabernet

the category is sparkling and the variable is blanc de noirs

the category is sticky and the variable is muscato

the category is white and the variable is pinot grigio

def my\_first\_func():

...: ... # my\_func can 'see' the global variable

...: ... print('I see "my\_var" = ', my\_var, ' from "my\_first\_func"')

my\_first\_func()

I see "my\_var" = 3 from "my\_first\_func"

In [25]: def hellow(name):

...: greeting = "Hellow %s!"%(name)

...: return greeting

...:

...:

In [27]: hellow('peiwu')

Out[27]: 'Hellow peiwu!'

def add(a, b):

return a + b

c=add(1,2)

c

Out[83]: 3

type(add)

Out[84]: function

def run\_a\_func(func, arg1, arg2):

result = add(arg1, arg2)

print('Result was', result)

run\_a\_func(add, 1, 2)

Result was 3

To define a function, you use the keyword **def**. Then comes the function name, in this case **hello**, with parentheses containing any input **arguments**the function might need. In this case, we need a name to form a proper greeting, so we're giving the **hello()** function a variable **argument** called **name**. After that, the function does its thing, executing the indented block of code immediately below. In this case, it creates a greeting "Hello <name>!". The last thing that it does is **return** that greeting to the rest of the program.

Technically speaking, a function does not *need* to explicitly return something, although it's uncommon that you'll write any that don't---if you do, it will probably be to write stuff to a file. If you don't **return** something explicitly, Python will nevertheless return the special object **None**. None is logically false (for if statements), and **print**ing None will result in the text "None" being printed (although None is not the string "None"). It's easy to forget to **return** a value, so this is an easy first thing to check in case your functions don't work as expected.

Note that the variable names are different on the inside and the outside of the function: I gave it **function Input**, although it takes **name**, and it returns **greeting**, although that return value is fed into **function Output**. I did this on purpose, as I want to emphasize that the function only knows to expect something, which it internally refers to as **name**, and then to give something else back. In fact, there is some insulation against the outside world.

In [31]: def whichFood(balance):

...: if balance < 10:

...: return 'ramen'

...: elif balance < 100:

...: return 'good ramen' /日本拉面

...: elif balance < 200 :

...: return 'better ramen'

...: else:

...: return 'ramen that is truly profound in its goodness'

...:

In [32]: whichFood(15)

Out[32]: 'good ramen'

In [33]: whichFood(9)

Out[33]: 'ramen'

In [34]: whichFood(107)

Out[34]: 'better ramen'

In [35]: whichFood(207)

Out[35]: 'ramen that is truly profound in its goodness'

Finally, we've shown examples with one input variable and one return value, but functions can accept zero input variables, one input variable, or multiple input variables, and functions don't necessarily need to return variables back to the program, but they are also capable of returning multiple variables. They can even have other functions nested inside them!  
Here are a few more examples of the syntax used with functions:

In [36]: def useless():

...: print('what was the point of that') #no argument

...:

In [37]: useless

Out[37]: <function \_\_main\_\_.useless()>

In [38]: useless()

what was the point of that

**Summary So Far...**

**Lists are:**

1) ordered collections of arbitrary variables.  
2) accessible by slicing.  
3) can be grown or shrunk in place.  
4) mutable (can be changed in place).  
5) defined with list = [X,Y,...]

**Tuples are:**

1) ordered collections of arbitrary variables  
2) accesible by slicing  
3) cannot be grown, shrunk, or changed in place  
4) defined with tuple = (X,Y,...)

**Dictionaries are:**

1）unordered collection of arbitrary variables

2) accessible by keys.  
3) can be grown or shrunk in place.  
4) mutable.  
5) defined with dict = {X:Y, A:B, ...}

In [102]: li = ['Red Leicester', 'Gruyere', 'Camembert', 'Parmesan', 'Mozarella', 'Chedda

...: r']

In [103]: for x in li:

...: print(x)

...:

Red Leicester

Gruyere

Camembert

Parmesan

Mozarella

Cheddar

In [104]: li = ['Red Leicester', 'Gruyere', 'Camembert', 'Parmesan', 'Mozarella', 'Chedda

...: r']

In [105]: for x in li:

...: print("Cleese: Do you have any %s?" % x)

...: if x == 'Camembert':

...: print("Chapman: Oooh, the cat's eaten it!")

...: else:

...: print("Chapman: No.")

...: print("Cleese: Well I'm sorry, but I'm going to have to shoot you.")

...:

...:

Cleese: Do you have any Red Leicester?

Chapman: No.

Cleese: Do you have any Gruyere?

Chapman: No.

Cleese: Do you have any Camembert?

Chapman: Oooh, the cat's eaten it!

Cleese: Do you have any Parmesan?

Chapman: No.

Cleese: Do you have any Mozarella?

Chapman: No.

Cleese: Do you have any Cheddar?

Chapman: No.

Cleese: Well I'm sorry, but I'm going to have to shoot you.

In [106]: for x in range(4):

...: print(x)

...:

0

1

2

3

In [107]: x

Out[107]: 3

In [1]: li2=[1,22,48,101,42]

In [2]: for x in range(len(li2)):

...: li2[x]=li2[x]+25

...:

In [3]: li2

Out[3]: [26, 47, 73, 126, 67]

In [2]: x = 5

...: while x > 0:

...: print(x)

...: x = x - 1

...: print

...:

5

4

3

2

1

Out[2]: <function print>

In [3]: y = 10

...: x = y-1

...: while x > 1:

...: if y % x == 0:

...: print(x, 'is a factor of', y)

...: break

...: x = x-1

...: else:

...: print(y, 'is prime')

...:

...:

5 is a factor of 10

In [4]: a = range(10)

...: b = []

...: for i in a:

...: b.append(i + 1)

...: a = b

...: print(a)

...:

...:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [6]: a = range(10)

...: # note that the entire statement below is contained in \*\*[ ]\*\*

...: # This is the defining syntax of a \*\*list comprehension\*\*.

...: print([ i + 1 for i in a ])

...:

...:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [7]: a = range(10) /create dictionary

...:

...: # list of 10 to 20

...: b = range(20)[10:]

...:

...: d = dict(zip(a,b))

...:

...: print(d)

...:

...:

{0: 10, 1: 11, 2: 12, 3: 13, 4: 14, 5: 15, 6: 16, 7: 17, 8: 18, 9: 19}

In [10]: 3+2==5

Out[10]: True

In [11]: 3>4

Out[11]: False

In [12]: 3-1!=3

Out[12]: True

In [13]: 3-1!=2

Out[13]: False

**Sets**

You will use dictionaries and lists almost exclusively in your coding. However, there is a remaining data structure that you should know about to make your life a little easier: **Sets**. **Sets** are unordered and unique bags of variables.  
A set is another nifty data structure. Like a set in mathematics, it has a bunch of elements with no repeats. To build a set, you pass in a list, and it will automatically remove duplicates.

In [10]: myset=set([1,2,1,3,5,8])

In [11]: myset

Out[11]: {1, 2, 3, 5, 8}

In [12]: myset.add(10)

In [13]: myset.add(15)

In [14]: myset

Out[14]: {1, 2, 3, 5, 8, 10, 15}

In [15]: UCs = set(["UC Berkeley", "UCLA", "UCSF", "UCSB", "UC Riverside", "UC

...: Davis",^M

...: "UCSD", "UC Santa Cruz", "UC Irvine", "UC Merced"])^M

...:

In [16]: bay\_area\_schools = set(["UC Berkeley", "Stanford", "UCSF", "San Jose S

...: tate",

...: "Santa Clara University", "University of San Francisco" ])

...:

In [17]: bay\_area\_UCs=UCs.intersection(bay\_area\_schools)

In [18]: bay\_area\_UCs

Out[18]: {'UC Berkeley', 'UCSF'}

In [19]: non\_bay\_area\_UCs=bay\_area\_schools.difference(UCs)

In [20]: non\_bay\_area\_UCs

Out[20]:

{'San Jose State',

'Santa Clara University',

'Stanford',

'University of San Francisco'}

In [21]: bay\_area\_schools.remove('stanford')

---------------------------------------------------------------------------

KeyError Traceback (most recent call last)

<ipython-input-21-f25933247e2b> in <module>

----> 1 bay\_area\_schools.remove('stanford')

KeyError: 'stanford'

In [22]: bay\_area\_schools.remove('Stanford')

In [23]: bay\_area\_schools

Out[23]:

{'San Jose State',

'Santa Clara University',

'UC Berkeley',

'UCSF',

'University of San Francisco'}

In [24]: bay\_area\_schools.discard('San Jose State')

In [25]: bay\_area\_schools

Out[25]:

{'Santa Clara University',

'UC Berkeley',

'UCSF',

'University of San Francisco'}

**Lists of lists**

Let's create a few lists. Being self-centered, let's make the list of our lives:

In [1]: coffee = ['yalis','strada','brewedawakening']

In [2]: lab = ['stanley','barker','LSA','Koshland','VLSB']

In [5]: importantplaces=[coffee,lab]

In [6]: importantplaces

Out[6]:

[['yalis', 'strada', 'brewedawakening'],

['stanley', 'barker', 'LSA', 'Koshland', 'VLSB']]

In [7]: aplace=importantplaces[0]

In [8]: aplace

Out[8]: ['yalis', 'strada', 'brewedawakening']

In [9]: aplace=importantplaces[0][0]

In [10]: aplace

Out[10]: 'yalis'

In [11]: home = ['rockridge','the mission','lake merritt','chinatown']

In [13]: importantplaces.append(home)

In [14]: importantplaces

Out[14]:

[['yalis', 'strada', 'brewedawakening'],

['stanley', 'barker', 'LSA', 'Koshland', 'VLSB'],

['rockridge', 'the mission', 'lake merritt', 'gourmet ghetto']]

In [15]: importantplaces[2].append('bushroad park')

In [16]: importantplaces

Out[16]:

[['yalis', 'strada', 'brewedawakening'],

['stanley', 'barker', 'LSA', 'Koshland', 'VLSB'],

['rockridge',

'the mission',

'lake merritt',

'gourmet ghetto',

'bushroad park']]

**Lists of dictionaries, dictionaries of lists**

Can we mix data structures? Of course! The main thing to keep in mind (besides what your data structure actually looks like) is that if you're building something up from scratch, you need to match the square brackets and curly braces:  
**Fancy loops for fancy data structures**

You are also allowed to loop over a dictionary. Be aware that it loops over the keys of the dictionary (i.e. the ones you put in brackets to get the values), and that you shouldn't count on any particular ordering to those keys:

In [17]: knights = {'gallahad': 'the pure', 'robin': 'the brave'}

In [18]: for x in knights:

...: print(x)

...: print(knights[x])

...:

gallahad

the pure

robin

the brave

In [19]: List\_of\_list=[[1,2,3],[7,8,9]]

In [21]: List\_of\_list

Out[21]: [[1, 2, 3], [7, 8, 9]]

In [22]: for x in List\_of\_list:

...: print(x)

...: for y in x:

...: print(y)

...:

[1, 2, 3]

1

2

3

[7, 8, 9]

7

8

9

**Modules**

In all of the examples above, we defined our functions right above the code that we hoped to execute. If you have many functions, you can see how this would get messy in a hurry. Furthermore, part of the benefit of functions is that you can call them multiple times within a program to execute the same operations without tiresomely writing them all out again. But wouldn't it be nice to share functions across programs, too? For example, working with genomic data means lots of time getting sequence out of FASTA files, and shuttling that sequence from program to program. Many of the programs we work with overlap to a significant degree, as they need to parse FASTA files, calculate evolutionary rates, and interface with our lab servers, for example -- all of which means that many of them share functions. And if the same function exists in two or more different programs, we hit the same problems that we hit before: complex debugging, decreased readability, and, of course, too much typing.  
**Modules** solve these problems. In short, they're collections of functions and variables (and often objects, which we'll get to towards the end of the course) that are kept together in a single file that can be read and imported by any number of programs.

In [25]: import sys

In [26]: x=float(sys.argv[0][23])

In [27]: x

Out[27]: 3.0

In [28]: sys.argv

Out[28]: ['C:\\Users\\Peiwu\\Anaconda3\\Scripts\\ipython']

In [29]: import math

In [30]: logX = math.log(10)

In [31]: x=5

In [32]: logX = math.log(x)

In [33]: logX

Out[33]: 1.6094379124341003

**Making a module**

Any file of python code with a *.py* extension can be imported as a module from your script. When you invoke an import operation from a program, all the statements in the imported module are executed immediately. The program also gains access to names assigned in the file (names can be functions, variables, classes, *etc.*), which can be invoked in the program using the syntax **module.name**. Go ahead and make your first module by pasting the following code into your text editor and saving as *greeting\_module.py*:

**print**('The top of the greeting\_module has been read.')

**def** hello(name):

greeting = "Hello %s!" % name

**return** greeting

**def** ahoy(name):

greeting = "Ahoy-hoy %s!" % name

**return** greeting

x = 5

**print**('The bottom of the greeting\_module has been read.')

In [2]: print('The top of the greeting\_module has been read.')

...: def hello(name):

...:

...: greeting = "Hello %s!" % name

...:

...: return greeting

...: def ahoy(name):

...:

...: greeting = "Ahoy-hoy %s!" % name

...:

...: return greeting

...: x = 5

...: print('The bottom of the greeting\_module has been read.')

The top of the greeting\_module has been read.

The bottom of the greeting\_module has been read.

Now make a new program called *test.py* with the following code and include your first name as an argument in the Terminal command line when you execute it:

*#!/usr/bin/env python*

**import** greeting\_module

hi = greeting\_module.hello('Peter')

**print** hi

**print** greeting\_module.x

*# What happens if you try 'print x' here?*

*# Remember how to access argv?*

**import** sys

**print** greeting\_module.hello(sys.argv[1])

*# This will take your Terminal argument as input for the greeting*

*# module's hello function*

In [11]: import greeting\_module

In [14]: hi=greeting\_module.hello('Peter')

In [15]: hi

Out[15]: 'Hello Peter!'

In [16]: print(greeting\_module.x)

5

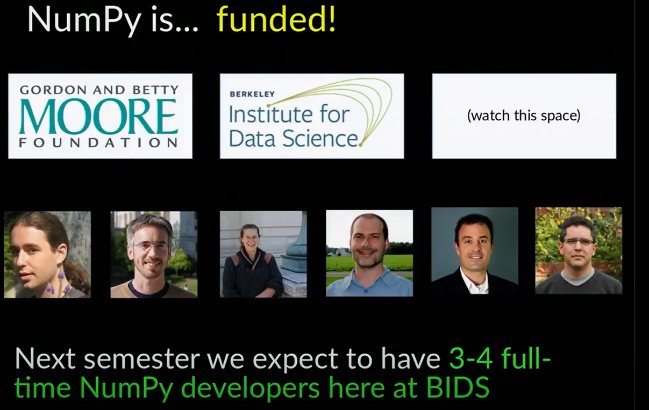
In [22]: print(greeting\_module.hello(sys.argv[0]))

Hello C:\Users\Peiwu\Anaconda3\Scripts\ipython!

In [23]: help()

**NumPy Basics**

1. Numerical Python is a powerful library of functions, methods, and data types we can used to analyze our data. Unforunately for those of us whose heads continue to spin in a crash-course of syntax, it also uses a different set of rules. I hope you'll understand why when you see the power and speed NumPy's data types afford us. Let's start off creating some empty arrays, which look sorta like lists, and are in fact vectors.  
   They differ in a few fundamental ways from lists:  
   Arrays cannot be of mixed types. They can be all **integers, floats, strings, logical** (or **boolean**) values, or other *immutable* values. But they cannot be some characters, some numbers, or any other olio of data types. They also cannot contain mutable types such as lists. So, we can have a list of lists, but not an array of lists. We can, however, have an array of arrays (sortof). Which brings us to:
2. Arrays can be multidimensional, but they must be rectangular. You can have a list of lists, where the first interior list is 3 elements long, the second 5, and the third 12, but your multidimensional array must be expressible as "a m by n (by j by k by...) array". I have never encountered a situation where Python says there are too many dimensions (but I've never had need beyond, maybe, 4 dimensions).
3. We can perform vector operations on them, which can be algebraic functions (like a dot product), or simple replacements of values in slice of the array.



In [43]: a=[0]\*10

In [44]: a

Out[44]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

In [45]: type(a)

Out[45]: list

In [46]: a=np.asarray(a)

In [47]: type(a)

Out[47]: numpy.ndarray

In [48]: a

Out[48]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In [49]: a=np.array([0]\*40)

In [50]: a

Out[50]:

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In [51]: a=np.zeros(40)

In [52]: a

Out[52]:

array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,

0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,

0., 0., 0., 0., 0., 0.])

In [53]: type(a)

Out[53]: numpy.ndarray

In [54]: type(a[0])

Out[54]: numpy.float64

In [55]: a=np.zeros(40,int)

In [56]: type(a[0])

Out[56]: numpy.int32

In [58]: a

Out[58]:

array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12.,

13., 14., 15., 16., 17., 18., 19., 20., 21., 22., 23., 24., 25.,

26., 27., 28., 29., 30., 31., 32., 33., 34., 35., 36., 37., 38.,

39.])

In [59]: type(a[0])

Out[59]: numpy.float64

In [60]: a=np.arange(40,50,dtype=float)

In [61]: a

Out[61]: array([40., 41., 42., 43., 44., 45., 46., 47., 48., 49.])

In [62]: a=np.arange(40,50,2,dtype=float)

In [63]: a

Out[63]: array([40., 42., 44., 46., 48.])

In [64]: a=np.zeros((10,10))

In [65]: a

Out[65]:

array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])

In [66]: a[5][5]=3

In [67]: a[6,6]=42

In [68]: a

Out[68]:

array([[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 3., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0., 42., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])

In [69]: a[1,:]=1

In [70]: a

Out[70]:

array([[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],

[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 3., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0., 42., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])

In [71]: a[:,0]=7

In [72]: a

Out[72]:

array([[ 7., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 7., 1., 1., 1., 1., 1., 1., 1., 1., 1.],

[ 7., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 7., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 7., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 7., 0., 0., 0., 0., 3., 0., 0., 0., 0.],

[ 7., 0., 0., 0., 0., 0., 42., 0., 0., 0.],

[ 7., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 7., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

[ 7., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])

In [73]: a=np.arange(10)

In [74]: a

Out[74]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [75]: a[2:5]

Out[75]: array([2, 3, 4])

In [76]: a[-1]

Out[76]: 9

In [77]: a[-5:]

Out[77]: array([5, 6, 7, 8, 9])

In [78]: a[:-5]

Out[78]: array([0, 1, 2, 3, 4])

In [79]: a+10

Out[79]: array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])

In [80]: a/2

Out[80]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])

In [81]: a\*3

Out[81]: array([ 0, 3, 6, 9, 12, 15, 18, 21, 24, 27])

In [82]: b=np.arange(10,20)

In [83]: a

Out[83]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [84]: b

Out[84]: array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])

In [85]: a\*b

Out[85]: array([ 0, 11, 24, 39, 56, 75, 96, 119, 144, 171])

In [86]: sum(a\*b)

Out[86]: 735

In [87]: sum(a)

Out[87]: 45

In [88]: sum(b)

Out[88]: 145

In [89]: a=np.zeros(10,dtype=bool)

In [90]: a

Out[90]:

array([False, False, False, False, False, False, False, False, False,

False])

In [91]: a[2:5]=True

In [92]: a

Out[92]:

array([False, False, True, True, True, False, False, False, False,

False])

In [93]: b=(a==False)

In [94]: b

Out[94]:

array([ True, True, False, False, False, True, True, True, True,

True])

In [95]: b=~a

In [96]: b

Out[96]:

array([ True, True, False, False, False, True, True, True, True,

True])

In [97]: a=np.arange(99)

In [98]: a

Out[98]:

array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,

17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,

34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,

51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,

68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,

85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98])

In [99]: np.mean(a)

Out[99]: 49.0

In [100]: np.std(a)

Out[100]: 28.577380332470412

In [101]: np.var(a)

Out[101]: 816.6666666666666

In [102]: np.

abs add\_docstring() all()

absolute add\_newdoc() allclose()

absolute\_import add\_newdoc\_ufunc() ALLOW\_THREADS >

add alen() alltrue()

In [102]: np.random.uniform(0,100,10) /range

Out[102]:

array([20.11609882, 62.10387441, 87.40308968, 24.83945678, 43.61384474,

37.21771511, 12.23670937, 78.96828438, 13.06338803, 1.34170328])

In [103]: np.random.uniform(0,100,(3,3))

Out[103]:

array([[34.75177903, 71.69657559, 52.99291718],

[36.97809545, 22.11297873, 35.78190942],

[46.17273351, 61.88300205, 19.85289732]])

In [107]: np.exp(10)

Out[107]: 22026.465794806718

In [108]: np.exp(1)

Out[108]: 2.718281828459045

In [109]: np.exp(0)

Out[109]: 1.0

In [110]: np.mean(np.random.exponential(np.exp(1),1000000))

Out[110]: 2.7114930012805183

In [155]: import numpy as no

In [156]: import numpy as np

In [157]: t=np.linspace(0,10,100)

In [158]: t

Out[158]:

array([ 0. , 0.1010101 , 0.2020202 , 0.3030303 , 0.4040404 ,

0.50505051, 0.60606061, 0.70707071, 0.80808081, 0.90909091,

1.01010101, 1.11111111, 1.21212121, 1.31313131, 1.41414141,

1.51515152, 1.61616162, 1.71717172, 1.81818182, 1.91919192,

2.02020202, 2.12121212, 2.22222222, 2.32323232, 2.42424242,

2.52525253, 2.62626263, 2.72727273, 2.82828283, 2.92929293,

3.03030303, 3.13131313, 3.23232323, 3.33333333, 3.43434343,

3.53535354, 3.63636364, 3.73737374, 3.83838384, 3.93939394,

4.04040404, 4.14141414, 4.24242424, 4.34343434, 4.44444444,

4.54545455, 4.64646465, 4.74747475, 4.84848485, 4.94949495,

5.05050505, 5.15151515, 5.25252525, 5.35353535, 5.45454545,

5.55555556, 5.65656566, 5.75757576, 5.85858586, 5.95959596,

6.06060606, 6.16161616, 6.26262626, 6.36363636, 6.46464646,

6.56565657, 6.66666667, 6.76767677, 6.86868687, 6.96969697,

7.07070707, 7.17171717, 7.27272727, 7.37373737, 7.47474747,

7.57575758, 7.67676768, 7.77777778, 7.87878788, 7.97979798,

8.08080808, 8.18181818, 8.28282828, 8.38383838, 8.48484848,

8.58585859, 8.68686869, 8.78787879, 8.88888889, 8.98989899,

9.09090909, 9.19191919, 9.29292929, 9.39393939, 9.49494949,

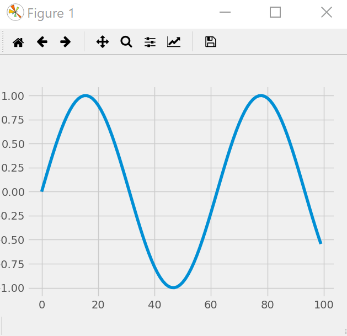
9.5959596 , 9.6969697 , 9.7979798 , 9.8989899 , 10. ])

In [159]: y=np.sin(t)

In [161]: plt.plot(y)

Out[161]: [<matplotlib.lines.Line2D at 0x29cde3792e8>]

In [162]: plt.show()

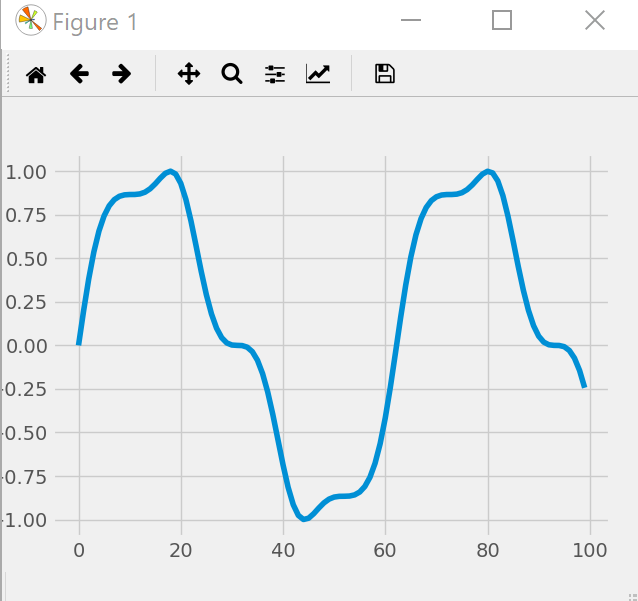


In [165]: y=np.sin(t+(1/3)\*np.sin(3\*t))

In [166]: plt.plot(y)

Out[166]: [<matplotlib.lines.Line2D at 0x29cde3f1630>]

In [167]: plt.show()



In [178]: y=np.sin(t)+(1/3)\*np.sin(3\*t)+(1/5)\*np.sin(5\*t)

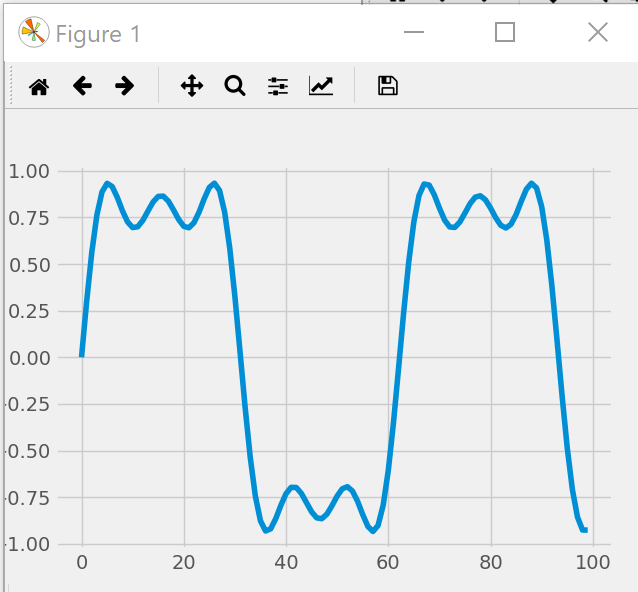
...:

...:

In [179]: plt.plot(y)

Out[179]: [<matplotlib.lines.Line2D at 0x29cdf00ddd8>]

In [180]: plt.show()



使用nditer可以完成的最基本的任务是访问数组的每个元素。 使用标准Python迭代器接口逐个提供每个元素.

>>> a = np.arange(6).reshape(2,3)

>>> for x in np.nditer(a):

... print x,

...

0 1 2 3 4 5

In [144]: a=np.arange(6).reshape(2,3)

In [145]: a

Out[145]:

array([[0, 1, 2],

[3, 4, 5]])

In [146]: for x in np.nditer(a):

...: print(x)

...:

0

1

2

3

4

5

Broadcasting

In [154]: a=np.array([[0,0,0],[10,10,10],[20,20,20],[30,30,30]])

...:

In [155]: a

Out[155]:

array([[ 0, 0, 0],

[10, 10, 10],

[20, 20, 20],

[30, 30, 30]])

In [156]: b=np.array([1,2,3])

In [157]: c=a+b

In [158]: c

Out[158]:

array([[ 1, 2, 3],

[11, 12, 13],

[21, 22, 23],

[31, 32, 33]])

**SciPy and Fitting**

SciPy (pronounced "Sigh Pie") is a collection of libraries that builds on NumPy, and has lots of convenient, fast functions for working with large amounts of scientific data. It's slightly smaller than NumPy, with only 900-odd pages of documentation. That includes sections on integrating C or Fortran code into Python, which is way outside the scope of this course, but if you ever do get to the point where you need a super-efficient implementation of something, you're covered. Especially in the one-off nature of academic science, you're often better served spending less time writing code that takes longer to run, compared to spending lots and lots of time writing code that runs slightly faster.  
The [stats module](http://docs.scipy.org/doc/scipy/reference/tutorial/stats.html) of SciPy has functions for even more statistical distributions, statistical tests, and other assorted functions that a good statistician might need. As an example, let's see how we might use the [linregress function](http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.linregress.html" \l "scipy.stats.linregress" \t "_blank), which does a linear regression on some data. Linear regression is the process of finding a line that minimizes the sum of the square of the vertical distances from each point to the line.  
First, we'll set up some noisy data:

**import** numpy **as** np

slope = .5

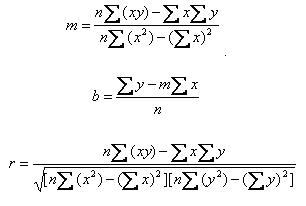
intercept = -10

x = np.arange(0,100)

y = slope\*x + intercept

noise = 5 \* np.random.randn(len(x))

y = y + noise

This isn't a math class, so we're going to start with the equations for slope, intercept, and correlation coefficient of the best-fit line as given:  


n = len(x)

m = (n \* sum(x \* y) - sum(x) \* sum(y)) / (n \* sum(x\*\*2) - (sum(x))\*\*2)

b = (sum(y) - m \* sum(x))/n

r = (n \* sum(x \* y) - sum(x) \* sum(y)) / np.sqrt((n\*sum(x\*\*2) - sum(x)\*\*2)

\* (n \* sum(y\*\*2) - sum(y)\*\*2))

**print(**m, b, r)

*0.486677343735 -9.05040165994 0.928979337505*  
This gives us pretty much the right result, but it was kind of a pain to type in. If only the libraries had some sort of function that could do linear regression for us...

**from** scipy **import** stats

r\_slope, r\_int, r\_rval, r\_pval, r\_stderr = stats.linregress(x, y)

**print**("Regression Slope: ", r\_slope)

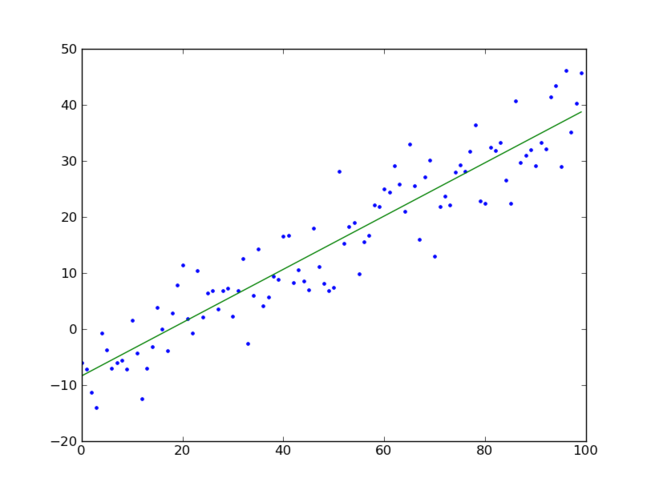
**print**("Regression Intercept: ", r\_int)

**print**("Regression correlation: ", r\_rval)

**print**("R^2:, ", r\_rval\*\*2)

**print**("p(slope is 0): ", r\_pval)

*Regression Slope: 0.486677343735*  
  
*Regression Intercept: -9.05040165994*  
  
*Regression correlation: 0.928979337505*  
  
*R^2:, 0.86300260951*  
  
*p(slope is 0): 4.31945319634e-44*



>>> corrcoef(x, y)

array([[ 1. , 0.92897934],

[ 0.92897934, 1. ]])

**import** random

list1 = [random.randrange(0, 100) **for** i **in** range(int(1e7))]

list2 = [random.randrange(0, 100) **for** i **in** range(int(1e7))]

list3 = [a + b **for** a, b **in** zip(list1, list2)]

In [13]: len(list3)

Out[13]: 10000000

In [14]: list3[:5]

Out[14]: [171, 102, 133, 88, 43]

In [15]: list1[:5]

Out[15]: [92, 57, 89, 76, 33]

In [16]: list2[:5]

Out[16]: [79, 45, 44, 12, 10]

**import** numpy **as** np

list1 = np.random.randint(0,101, 1e7)

list2 = np.random.randint(0,101, 1e7)

list3 = list1 + list2