Zero-to-Hero: Python Programming
Yogesh Haribhau Kulkarni

Module 1: Introduction

Python: Quick Introduction

Guess
What are the differences with the programming languages you know?

```
x = 34 - 23
y = ‘‘Hello’’
z = 3.45
if z == 3.45 or y == ‘‘Hello’’:
  x = x + 1
  y = y + ‘‘ World’’
print(x)
print(y)
```

Why Python?

- Readability
- Ease of use
- “Fits in your head”
- Incremental sense of accomplishment, aka“gets things done”
- Good libraries
- Deployment, aka “Lookie what I did!”

Truths about Good Programmers

- Lazy (in a good way)
- Just want things to work
- Spoiled kids who just want to have fun
- And sometimes create Fortune 100 companies

One Truth About Python

- Power scales with the ability of the programmer
- Novices can do simple things
- Really bright people build tools
- Novices leverage these tools
- Lone sys-admins <3 perl
- Mavericks in small work-groups <3 Python

Brief History of Python

- Invented in the Netherlands, early 90s by Guido van Rossum
- Named after Monty Python (a British comedy group, the language has a playful approach)
- Open sourced from the beginning
- Considered a scripting language, but is much more
- Used by Google from the beginning
- Increasingly popular

Overview

Why Python?

- Readability.
- Ease of use.
- Fits in your head.
- Gets things done.
- Good libraries.
- Lookie what I did!.

Introduction

- Python is a simple, yet powerful interpreted language.
- Numerous libraries: NumPy, SciPy, Matplotlib ....
- Named after Monty Python.
- Open Source and Free
- Invented by Guido van Rossum.

Python’s Benevolent Dictator For Life

“Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another’s code; too little and expressiveness is endangered.”
- Guido van Rossum

(Reference: https://en.wikipedia.org/wiki/Guido_van_Rossum)

What is Python?

- Interpreted
- Object-oriented
- High-level
- Dynamic semantics
- Cross-platform
- Readability.

Compiled Languages

- Needs entire program
- Translates directly to machine codes
- Exe native and fast
- Usually statically typed
- Types known during compilation
- Change in type : recompilation
- Ideal for compute-heavy tasks
- E.g. C, C++, FORTRAN

Interpreted Languages

- Interpreted on the fly
- No need to compile: can execute right away
- Usually dynamically typed
- Non-syntax errors are detected only in run-time
- Slower than compiled languages
- Ideal for small tasks
- E.g. Python, Perl, PHP, Bash

Note: Python, at the beginning, loosely checks the program. Only at run time, line-by-line, it checks for errors. So, if the error statements are not in the running path, their error does not get reported. So, its a bit relaxed. Thats the objection for its use in production code, where strong type checking and error checking, upfront is essential.

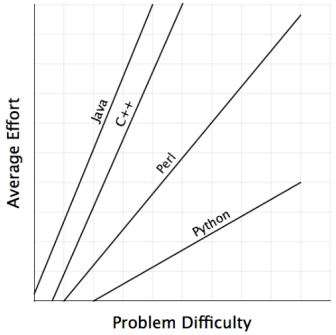
JIT-Compiled Languages

- Between compiled and interpreted
- Code is initially interpreted, hotspots compiled
- Deduce types during compilation, can change in run-time
- Slower than compiled
- E.g. Java, C#

"C" guys to take pride in

- Python interpreter written in "C"
- Source code at www.python.org
- So, Python Interpreter is compiled exe

Completely Controversial Observations about Languages



Python Help

- Home page – http://www.python.org
- Wiki – http://wiki.python.org/
- Packages – https://pypi.python.org/pypi
- Projects at http://sourceforge.net and github.org

Important features

- Built-in high level data types: `strings`, `lists`, `dictionaries`, etc.
- Usual control structures: `if`, `if-else`, `if-elif-else`, `while`, `for`
- Levels of organization: `functions`, `classes`, `modules`, `packages`
- Extensions in C and C++ possible

Python 2 *vs* Python 3

- Two major versions 2.*, 3.*
- Python 2.7: Latest release in 2.x series
- Python 3.5: More polished syntax, removed inconsistencies

Installations

Installing Python

- Pre-installed on most Unix, Linux and MAC OS X
- Windows: Python.org or Anaconda (preferred)
- Anaconda has necessary libraries
- 2.* and 3.* both
- 'pip' and 'conda'

Installing Python by Anaconda Distribution

- Need Python 3.5 (not 2.* or 3.6)
- By default in Anaconda 4.2.0
- Site: https://repo.continuum.io/archive/
- 64 Bit: Anaconda3-4.2.0-Windows-x86_64.exe
- 32 bit: Anaconda3-4.2.0-Windows-x86.exe
- About 300+MB

The Python shell, I

- Can run from "shell", IDE, Notebook
- Shell/Command Line (below):
- Start typing at >>>

```
$ python

Python 3.5.3 | packaged by conda-forge | (default,
    May 12 2017, 16:16:49) [MSC v.1900 64 bit
    (AMD64)] on win32
Type "help", "copyright", "credits" or "license"
    for more information.
>>>
```

The Python shell, I

- Typing anything??
- Need to learn syntax

```
>>> I come in peace, please take me to your leader
File "<stdin>" , line 1
I come in peace, please take me to your leader
^
SyntaxError : invalid syntax
>>>
```

The Python shell, II

- Expressions: evaluated, result printed:

```
>>> 2+2
4
```

- Line continuation

```
>>> "hello" + \
... " world!"
'hello world!'
```

- Prompt changes to '...' on continuation lines.

The Python shell, II

- Syntactically correct:

```
>>>print('I come in peace, please take me')
```

- `print` is a function, with defined syntax.
- If you don't obey:

```
>>>print 'I come in peace, please take me'
File "<stdin>", line 1
print 'I come in peace, please take me'
^
SyntaxError: Missing parentheses in call to
    'print'
```

The Python shell, II

- To exit:

```
>>> good-bye
Traceback (most recent call last):
File"<stdin>", line1, in <module>
NameError name 'good' is not defined
>>> if you don't mind, I need to leave
File "<stdin>", line 1
if you don't mind, I need to leave
          ^
SyntaxError: invalid syntax
```

- Better to:

```
>>> quit()
```

Exercises

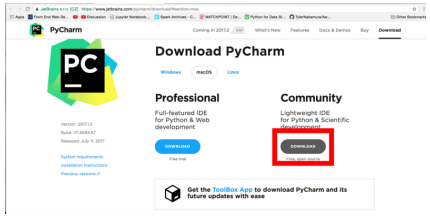
- Find more about JIT-compilation.
- Install any IDE, having debugging.
- Get familiar with command prompt.

Integrated Dev Envt

- Pycharm https://www.jetbrains.com/pycharm/
- IDLE comes with python.org distribution
- Spyder https://pypi.python.org/pypi/spyder

PyCharm

Download the community edition of Pycharm



PyCharm Project

Create a New Project

PyCharm Exercise

- Create a directory.
- Create New Project in Pychram.
- Give the newly created dir as path.
- Create New Python file. Write:

```
print(‘‘Hello World!!’’)
```

- Run.
- Result?.
- Get familiar with Pycharm UI.

Module 2: Basics

Syntax

Lines

- Statement separator : a semi-colon,
- Only needed for multiple statements on a line.
- Else, nothing
- Continuation: a back-slash
- But, not necessary in “context” : ( ),{ },[ ]

Blocks and indentation

- Blocks by indentation
- No begin/end brackets
- Indentation is 4 spaces and no hard tabs
- Reduces clutter
- `if`, `while`, `function`, `except`, etc have : at the end
- : must follow with an indent on next line.

Comments

- Everything after “#” ignored.

```
# compute the percentage of the hour that
    has elapsed
percentage = (minute * 100) / 60

percentage = (minute * 100) / 60 #
    percentage per hour
```

- Useless/redundant:

```
v = 5 # assign 5 to v
```

- Useful info:

```
v = 5 # velocity in meters/second.
```

Comments

No block comments

```
# A comment, this is so you can read program later.
# Anything after the # is ignored by python.
print("something.") # and the comment after is
    ignored

# Use a comment to "disable" a piece of code:
# print("This won't run.")
print("This will run.")
```

Data Types

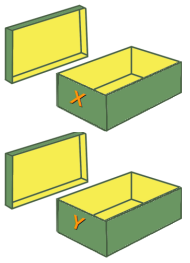
Variables

- A variable can be seen as a container (or some say a pigeonhole) to store certain values.
- While the program is running, variables are accessed and sometimes changed, i.e. a new value will be assigned to a variable.
- Every variable has and must have a unique data type.
- Declaring a variable means binding it to a data type.
- Defining a variable meas to have value assigned to it.

Variables in C/C++ or Java

```
int x;
int y;
```

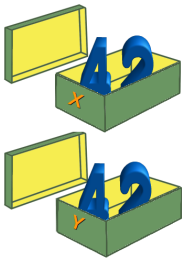
- Such declarations make sure that the program reserves memory for two variables with the names x and y.
- The variable names stand for the memory location.
- It's like the two empty shoeboxes, labelled with x and y.
- Like the two shoeboxes, the memory is empty as well.



Variables in C/C++ or Java

```
x = 42;
y = 42;
```

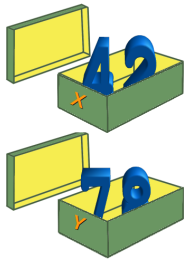
- Putting values into the variables can be realized with assignments.
- Two numbers are physically saved in the memory, which correspond to the two shoeboxes.



Variables in C/C++ or Java

```
y = 78;
```

- If we assign a new value to one of the variables, let's say the values 78 to y.
- We have exchanged the content of the memory location of y.



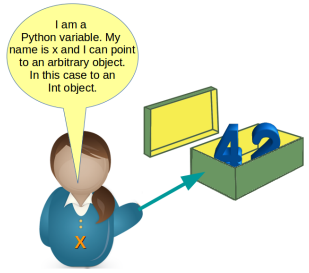
Variables in Python

- There is no declaration of variables required.
- Not only the value of a variable may change during program execution but the type as well.
- You can assign an integer value to a variable, use it as an integer for a while and then assign a string to the same variable.

```
i = 42      # data type is implicitly set to
    integer
i = 42 + 0.11   # data type is changed to float
i = "forty"   # and now it will be a string
```

All variables in Python are references

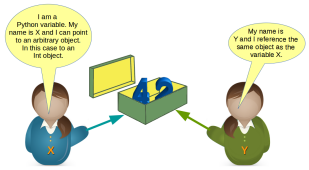
Python variables are references to objects, but the actual data is contained in the objects:



As variables are pointing to objects and objects can be of arbitrary data type, variables cannot have types associated with them.

All variables are references

```
>>> x = 42
>>> y = x
```

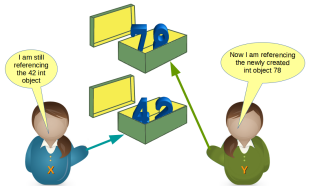


All variables are references

What will happen, when we execute

```
y = 78
```

Python will create a new integer object with the content 78 and then the variable y will reference this newly created object, as we can see in the following picture:

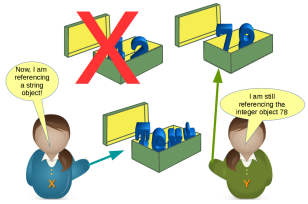


All variables are references

What will happen, when we execute

```
x = ‘‘Text’’
```

The previously integer object "42" will be orphaned after this assignment. It will be removed by Python, because no other variable is referencing it.



All variables are references

In Python, **all objects are ever passed by reference**. In particular, **variables always store a reference to an object**, never a copy!
Hence, you have to be careful when modifying objects:

```
>>> a = [1,2,3]
>>> b = a
>>> b.remove(2)
>>> print(a)
```

???

Q: How many items are in the `a` list now?

All variables are references

However

```
>>> a = 1
>>> b = a
>>> b += 1
>>> a
1
>>> b
2
```

How can you explain this?
The $b+=1$ operator could be replaced by $b = b + 1$, and the $b + 1$ expression yields a new value.

Dynamic Typing: Python

- Variables come into existence at first assignment.
- A variable can refer to an object of any type.
- Can switch type based on assignment
- All types (almost) treated the same way.
- Type errors only caught in runtime.

Variables

Definition-Declaration?

- Assignment: Definition + Declaration:

```
>>> n = 17

>>> pi = 3.1415926535897931

>>> message = 'And now for something
    different'
```

Variables Value

- Display value

```
>>> print(n)
17

>>> print(pi)
3.141592653589793

>>> print(message)
???
```

Variables Type

- Display type

```
>>> print(type(n))
<class 'int'>

>>> print(type(pi))
<class 'float'>

>>> print(type(message))
??
```

Variable Names

- Can be arbitrarily long.
- Can contain both letters and numbers,
- Cannot start with a number.
- (_) can appear in a name.
- Reserves 33 keywords

Variable Names Errors

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax

>>> more@ = 1000000
SyntaxError: invalid syntax

>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

Choosing mnemonic variable names

Variable names different but accomplishment same.

- ```
  a = 35.0
  b = 12.50
  c = a * b
  print(c)
  ```

- ```
  hours = 35.0
  rate = 12.50
  pay = hours * rate
  print(pay)
  ```

- ```
  x1q3z9ahd = 35.0
  x1q3z9afd = 12.50
  x1q3p9afd = x1q3z9ahd * x1q3z9afd
  print(x1q3p9afd)
  ```

Which one better?

Basic Data types

- **bool** : `True`, `False`.
- **int** Integer numbers.
- **long**: Switches internally from `int` to `long` when needed.
- **float** Double precision floating-point numbers
- **str** String (bytes).
- **list** Mutable list
- **tuple** Immutable list
- **set** Unordered collection
- **dict** Key/value mapping

Built-in object types

- Numbers : `3.1415, 1234, 999L, 3+4j`
- Strings : `'spam', ''guido's''`
- Lists : `[1, [2, 'three'], 4]`
- Dictionaries :`{'food':'spam', 'taste':'yum'}`
- Tuples : `(1,'spam', 4, 'U')`
- Sets: `{1,2,3,'foo','bar'}`

Numbers

- Integers : `1234, -24, 0`
- Unlimited precision integers : `999999999999L`
- Float : `3.1415, 2.7122`
- Oct and hex :`0177, 0x9ff`
- Complex : `3+4j, 3.0+4.0j, 3J`

Integer

```
int_1 = 1
int_2 = -2
int_3 = 100

print(int_1 + int_2 + int_3) # for integers, + is
    "plus"

# this is not what is looks like
print(1,000,000)

# eh, what is this then?
i = 1,000,000
type(i)

# Can I add these? Yes, but...
i = 1,000,000
j = 2,000,000
i + j

i / j
```

Floating Point Numbers

Decimal numbers, representations of fractions, "real-valued"

```
float_1 = 1.2
float_2 = -4.0
float_3 = 10.0

print(float_1 + float_2 + float_3)

males = 7
females = 10
```
```

```
fraction = males/(males+females)
print("Percentage men: {}:".format(fraction))
```

Booleans

'True' or 'False'.

```
bool_1 = True
bool_2 = False

print(bool_1)
print(bool_2)
```

String Quotes

- Single and double quotes interchangeable:

```
>>> "a string" == 'a string'
True
```

- One inside another ok:

```
>>> a = "Isn't it ok?"
>>> b = '"Yes", he said.'
```

- Just be consistent.

Multi Line

Multi-line strings with three quotes.

```
str_10 = """
(CNN)AirAsia Flight QZ8501 climbed rapidly before
    it crashed, a top Indonesian official said
    Tuesday, according to The Jakarta Post. Then
    the plane stalled, Transportation Minister
    Ignasius Jonan said at a parliamentary
    hearing, according to the AFP and Reuters
    news agencies. "The plane, during the last
    minutes, went up faster than normal speed ...
    after then, it stalled. That is according to
    the data from the radar," Jonan said,
    according to the news agencies.
"""
print(str_10)
```

String Indexing

Can be indexed (subscripted), start at 0.

```
word = 'Python'
word[0]  # character in position 0
word[1]  # character in position 5
word[-1]  # last character
word[-2]  # second-last character
word[-6] #?
```

No separate character type: a string of size one.

Slicing

```
word[0:2]  # from position 0 (included) to 2
    (excluded)
word[2:5]  # from position 2 (included) to 5
    (excluded)
word[2:]  # from position 2 (included) to the end
    (excluded '\0')
word[:3]  # from beginning to position 3 (excluded)
word[-3:] # last three characters.
word[-3:-1] # penultimate two characters
word[-1:-3] # Error. Wrong direction
Direction of substring is always from left to
    right.
```

String Exercise

- Assign the string 'Dealing with Data' to a Python variable.
- Show usages positive indexing/slicing
- Show usage of negative indexing/slicing

Start-End

Return `True` if `t` is the initial/final substring

```
s.startswith(t)

s.endswith(t)
```

String Exercise

- Write a Python program to get a new string from a given string where "Is" has been added to the front. If the given string already begins with "Is" then return the string unchanged.
- Write a Python program to get the n (non-negative integer) copies of the first 2 characters of a given string. Return the n copies of the whole string if the length is less than 2

Cases

```
s.capitalize() # First letter of sentence
uppercase

s.title() # First letter of all the words
uppercase

s.lower() # All letters lowercase

s.upper()  # All letters uppercase
```

Copy of s is made, modified and returned.

Split

Split `s` at every occurrence of `t` and return a list of parts.

```
s.split(t)
```

If `t` is omitted, split on whitespace.

Exercise

- Write a program that computes the net amount of a bank account based a transaction log from console input. The transaction log format is shown as following:

```
D 100
W 200
```

- D means deposit while W means withdrawal.
- Suppose the following input is supplied to the program:

```
D 300
D 300
W 200
D 100
```

- Then, the output should be: 500
- Hints:In case of input data being supplied to the question, it should be assumed to be a console input.

Solution

```
import sys
netAmount = 0
while True:
    s = input("Enter the transaction")
    if not s:
        break
    values = s.split(" ")
    operation = values[0]
    amount = int(values[1])
    if operation=="D":
        netAmount+=amount
    elif operation=="W":
        netAmount-=amount
    else:
        pass
print(netAmount)
```

Replace

Return a *copy* of `s` with all `old` replaced by `new`.

```
    s.replace(old, new)
```

Strip

Return a *copy* of with the leading (resp. trailing, resp. leading *and* trailing) whitespace removed.

```
      s.lstrip()

      s.rstrip()

      s.strip()
```

Finding text within string variables

```
word = "And on and on and on and on..."
ind = word.find("on")
print(ind)
print("The first time on is at position",
     word.find("on"))

first_appearance = word.find("on")
second_appearance =
     word.find("on",first_appearance+1)
print("The second time on is at ",
     second_appearance)
```

Finding text within string variables

Looking for "on" at the second half of "word"

```
midpoint = int(len(word)/2) # finds middle of
     string word
second_half_appearance = word.find("on",midpoint)
print("First time 'on' in the second half: ",
     second_half_appearance)
```

Finding text within string variables

```
word = "Python: And on and on and on and on..."
lookfor = "PYTHON"
count = word.count(lookfor)
print( "See '", lookfor  ,"' that many times: ",
     count)
```

Exercise: Convert the code above to make it case-insensitive.

Concatenating strings

Old way:

```
names = ['raymond', 'rachel', 'matthew', 'roger',
         'betty', 'melissa', 'judith', 'charlie']

s = names[0]
for name in names[1:]:
    s += ', ' + name
print(s)
```

Better:

```
print ', '.join(names)
```

(Ref: Transforming Code into Beautiful, Idiomatic Python - Raymond Hettinger)

Joining list of Strings

```


mystring1 = "practical data science"
mylist1 = mystring1.split(" ")
print(mystring1)
print(mylist1)

" ".join(mylist1) # Try different "<c>" here
```

String Comparisons

```
str_1 = "hello"
print("equality:")
print(str_1 == "hello")

print(str_1 == "Hello")
```

String Exercise

- Consider the string "billgates@microsoft.com".
- Write code that finds the username of the email address and the domain of the email address.
- Use multiple methods (eg. slicing with raw numbers, slicing with find, and the split command).

String Formatting

- To embed other information into strings.
- Sometimes with special formatting constraints.

```
print('Coordinates: {}, {}'.format('37.24N',
     '115.81W'))


print('Coordinates: {0}, {1}'.format('37.24N',
     '115.81W'))

lon = '37.24N'
lat = '115.81W'
print('Coordinates: {0}, {1}'.format(lon, lat)) #
     skipping 0,1 works?

print('Latitude: {0}, Longitude: {1} ==> [{0},
     {1}]'.format('37.24N', '115.81W'))

# Skipping 0,1 does not work as, then, it expects
     4 arguments.
```

String Formatting

Alternatively, instead of using the 1, 2, etc. format, specify names for the attributes:

```
print('Coordinates: {latitude}, {longitude}'
      .format(latitude='37.24N',
     longitude='115.81W'))
```

strings (recap)

- single quote: `s1 = 'egg'`
- double quotes: `s2 = ''spam's''`
- triple quotes: `block = '''...'''`
- concatenate: `s1 + s2`
- repeat: `s2 * 3`
- index,slice: `s2[i], s2[i:j]`
- length: `len(s2)`
- formatting: `''a {} parrot''.format('dead')`

Exercise

- Write a Python program to swap two string variables
- Write a Python program to check if a string is numeric.
- Write a Python program to check if lowercase letters exist in a string.
- Write a Python program to check if multiple variables have the same value

Exercises

Write a Python program to get a string made of the first 2 and the last 2 chars from a given a string. If the string length is less than 2, return instead of the empty string.

Sample String : 'w3resource'

Expected Result : 'w3ce'

Sample String : 'w3'

Expected Result : 'w3w3'

Sample String : ' w'

Expected Result : Empty String

Exercises

Write a Python program to get a string from a given string where all occurrences of its first char have been changed to '$', except the first char itself.

Sample String : 'restart'

Expected Result : 'resta$t'

Exercises

Write a Python program to get a single string from two given strings, separated by a space and swap the first two characters of each string.

Sample String : 'abc', 'xyz'

Expected Result : 'xyc abz'

Exercises

Write a Python program to add 'ing' at the end of a given string (length should be at least 3). If the given string already ends with 'ing' then add 'ly' instead. If the string length of the given string is less than 3, leave it unchanged.

Sample String : 'abc'

Expected Result : 'abcing'

Sample String : 'string'

Expected Result : 'stringly'

Exercises

Write a Python program to find the first appearance of the substring 'not' and 'poor' from a given string, if 'bad' follows the 'poor', replace the whole 'not'...'poor' substring with 'good'. Return the resulting string.

Sample String : 'The lyrics is not that poor!'

Expected Result : 'The lyrics is good!'

Operators

Operators

- The usual unary and binary arithmetic operators: $+, -, *, /, **, <<, >>$, etc.
- Logical operators: `and`, `or`, `not`.
- Numerical and string comparison: `<`, `>`, `<=`, `==`, `!=`, ...
- Order of operations: PEMDAS: Parentheses, Exponentiation, Multiplication, Division, Addition and Subtraction.
- Operators with the same precedence are evaluated from left to right.

First exercise

Calculate:

How much is 2^{38} ?

Operators

Some operators are defined for non-numeric types:

```
>>> "U" + 'ZH'
'UZH'
```

Some support operands of mixed type:

```
>>> "a" * 2
'aa'
>>> 2 * "a"
'aa'
```

Some do not:

```
>>> "aaa" / 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /:
    'str' and 'int'
```

Operators

There are two kinds of division operators:

- "true division" performed by "/"
- "floor division" performed by "//"

```
  >>> 10 / 3
3.3333333333333335
>>> 10.0 / 3.0
3.3333333333333335
>>> 10.5 / 3.5
3.0
>>>

>>> 9 // 3
3
>>> 10 // 3
3
>>> 11 // 3
3
>>> 12 // 3
4
>>> 10.0 // 3
3.0
```

Operators

The "%" operator computes the remainder of integer division.

```
>>> remainder = 7 % 3
>>> print(remainder)
1
```

Operators

The + operator works with strings, but it is not addition in the mathematical sense. Instead it performs concatenation.

```
>>> first = 10
>>> second = 15
>>> print(first+second)
25
>>> first = '100'
>>> second = '150'
```

```
>>> print(first + second)
100150
```

Operators

Use assignment '=' statement:

```
>>> a = 1
>>> print(a)
1
```

Shortcut notations:

`a += b` short for `a = a + b`,

`a -= b` short for `a = a - b`,

`a *= b` short for `a = a * b`,

The `is` operator

Allows to test whether two names refer to the same object:

```
>>> a = 1
>>> b = 1
>>> a is b
True
```

Exercise, I

- Assume that you go to a restaurant, and you order Rs.50 worth of food.
- Then you need to add the Sales Tax (8.875%) and add a tip (say, 20%).
- Write down the calculation that will print the total cost of the food.

Exercise, II

- You have a stock that closed at Rs.550 on Monday, and then closed at Rs.560 on Tuesday.
- Calculate its daily return: the daily return is defined as the difference in the closing prices, divided by the closing price the day before.

Exercise, III

- Write a Python program to solve $(x + y) * (x + y)$
- Write a Python program to compute the future value of a specified principal amount, rate of interest, and a number of years. Test Data : amt = 10000, int = 3.5, years = 7. Expected Output : 12722.79
- Write a Python program to compute the distance between the points (x1, y1) and (x2, y2).

Module 3: Constructs

Conditionals

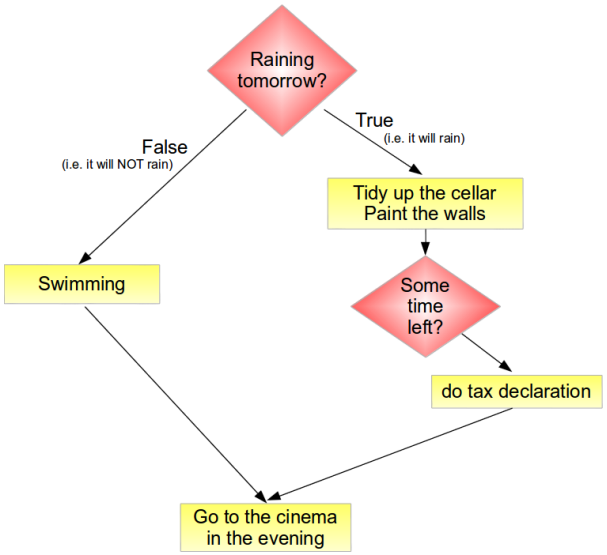
Conditionals

What can we deduce from the following text:

```
If it rains tomorrow, I will tidy up the cellar.
    After this I will paint the walls. If there
    is some time left, I will do my tax
    declaration. Otherwise, I will go swimming.
    In the evening, I will go to the cinema with
    my wife!
```

Conditionals

```
If it rains tomorrow, I will do the following:
    - tidy up the cellar
    - paint the walls
    - If there is some time left, I will
          - do my tax declaration
Otherwise, I will do the following:
    - go swimming
go to the cinema with my wife in the evening
```



Conditionals

C:

```
if (raining_tomorrow) {
    tidy_up_the_cellar();
    paint_the_walls();
    if (time_left)
        do_taxes();
} else
    enjoy_swimming();
    go_cinema();
```

Python:

```
if raining_tomorrow:
    tidy_up_the_cellar()
    paint_the_walls()
    if time_left:
        do_taxes()
else:
    enjoy_swimming()
go_cinema()
```

Conditionals

Conditional execution uses the `if` statement:

```
if expr:
  # indented block
elif  other-expr:
  # indented block
else:
  # executed if none of the above matched
```

The `elif` can be repeated, with different conditions, or left out entirely. Also the `else` clause is optional.
Where's the 'end if'? There's no 'end if': indentation delimits blocks!

Boolean expressions

A boolean expression is an expression that is either true or false.

```
>>> 5 == 5
True
>>> 5 == 6
False
```

True and False are special values that belong to the class bool; they are not strings:

```
>>> type(True)
<class 'bool'>
>>> type(False)
```

```
<class 'bool'>
```

Comparison operators

== operator is one of the comparison operators; others are:

```
x != y # x is not equal to y
x > y # x is greater than y
x < y # x is less than y
x >= y # x is greater than or equal to y
x <= y # x is less than or equal to y
x is y # x is the same as y
x is not y # x is not the same as y
```

Logical operators

- There are three logical operators: and, or, and not.
- `x > 0 and x < 10` is true only if x is greater than 0 and less than 10.
- `n%2 == 0 or n%3 == 0` is true if either of the conditions is true, that is, if the number is divisible by 2 or 3.

Quiz: Find Largest

Python program to find the largest number among the three input numbers

```
num1 = 10
num2 = 14
num3 = 12

# uncomment following lines to take three numbers
    from user
#num1 = float(input("Enter first number: "))
#num2 = float(input("Enter second number: "))
#num3 = float(input("Enter third number: "))
```

Solution: Find Largest

```
if (num1 >= num2) and (num1 >= num3):
   largest = num1
elif (num2 >= num1) and (num2 >= num3):
   largest = num2
else:
   largest = num3

print("The largest number
    between",num1,",",num2,"and",num3,"is",largest)
```

Exercises

- Write a Python program to get the difference between a given number and 17, if the number is greater than 17 return double the absolute difference.
- Write a Python program to calculate the sum of three given numbers, if the values are equal then return thrice of their sum
- Write a Python program to sum of two given integers. However, if the sum is between 15 to 20 it will return 20.

switch/case

- There is no such statement in Python.
- Conditional execution uses the `if` statement

Advance:
It can be implemented efficiently with a dictionary of functions:

```
result = {
  'a': lambda x: x * 5,
  'b': lambda x: x + 7,
  'c': lambda x: x - 2
}
result['b'](10)
```

Exercises

Rewrite your pay computation to give the employee 1.5 times the hourly rate for hours worked above 40 hours.

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error message. If the score is between 0.0 and 1.0, print a grade using the following table:

```
Score Grade
>= 0.9 A
>= 0.8 B
>= 0.7 C
>= 0.6 D
< 0.6 F
```

Loops

Looping

- Computers are often used to automate repetitive tasks.
- Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

```
n = 5
while n > 0:
  print(n)
  n = n - 1
print('Blastoff!')
```

`for`-loops

With the `for` statement, you can loop over the items of a sequence:

```
for i in range(0, 4):
  # loop block
  print (i*i)
```

To break out of a `for` loop, use the `break` statement.
To jump to the next iteration of a `for` loop, use the `continue` statement.

`for`

The `for` statement can be used to loop over elements in *any sequence*.

```
>>> for val in [1,2,3]:
...   print(val)
1
2
3
```

Looping over range of numbers

Two ways:

```
for i in [0, 1, 2, 3, 4, 5]:
    print i**2

for i in range(6):
    print i**2
```

Better:

```
for i in xrange(6):
    print i**2
```

xrange creates an iterator over the range producing the values one at a time. This approach is much more memory efficient than range. xrange was renamed to range in python 3

(Ref: Transforming Code into Beautiful, Idiomatic Python - Raymond Hettinger)

Looping over a collection

"C" way:

```
colors = ['red', 'green', 'blue', 'yellow']

for i in range(len(colors)):
    print(colors[i])
```

Better:

```
for color in colors:
    print(color)
```

(Ref: Transforming Code into Beautiful, Idiomatic Python - Raymond Hettinger)

for

The `for` statement can be used to loop over elements in *any sequence*.

```
>>> for val in {'UZH'}:
...   print(val)
'U'
'Z'
'H'
```

Looping backwards

"C" way:

```
colors = ['red', 'green', 'blue', 'yellow']

for i in range(len(colors)-1, -1, -1):
    print(colors[i])
```

Better:

```
for color in reversed(colors):
    print(color)
```

(Ref: Transforming Code into Beautiful, Idiomatic Python - Raymond Hettinger)

Sorted

If you want to loop over a *sorted* sequence you can use the function `sorted()` :

```
>>> for val in sorted([1,3,4,2]):
...  print(val)
1
2
3
4
```

and to loop over a sequence in *inverted* order you can use the `reversed()` function:

```
>>> for val in reversed([1,3,4,2]):
...     print(val)
2
4
3
1
```

Looping over a collection and indices

"C" way:

```
colors = ['red', 'green', 'blue', 'yellow']

for i in range(len(colors)):
    print(i, '--->', colors[i])
```

Better:

```
for i, color in enumerate(colors):
    print(i, '--->', color)
```

It's fast and beautiful and saves you from tracking the individual indices and incrementing them.

Whenever you find yourself manipulating indices [in a collection], you're probably doing it wrong

(Ref: Transforming Code into Beautiful, Idiomatic Python - Raymond Hettinger)

Looping over two collections

"C" way:

```
names = ['raymond', 'rachel', 'matthew']
colors = ['red', 'green', 'blue', 'yellow']

n = min(len(names), len(colors))
for i in range(n):
    print(names[i], '--->', colors[i])

for name, color in zip(names, colors):
    print(name, '--->', color)
```

Better:

```
for name, color in zip(names, colors):
    print(name, '--->', color)
```

zip creates a new list in memory and takes more memory. izip is more efficient than zip. Note: in python 3 izip was renamed to zip and promoted to a builtin replacing the old zip.

(Ref: Transforming Code into Beautiful, Idiomatic Python - Raymond Hettinger)

Looping

Conditional looping uses the `while` statement:

```
while expr:
  # indented block
 else:
   # executed at natural end of the loop
```

- To break loop, use the `break` statement.
- Use `continue` anywhere inside to jump back to the `while`.
- If a loop is exited via a `break` statement, the `else` is not executed.
- `else` is optional.

Looping Exercises

- Write a Python program to count the number 4 in a given list.
- Write a Python program to create a histogram from a given list of integers.

Looping

To find the largest value in a list or sequence, we construct the following loop:

```
largest = None
print('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
  if largest is None or itervar > largest :
    largest = itervar
  print('Loop:', itervar, largest)
print('Largest:', largest)
```

Finding smallest, the code is very similar with one small change:

```
smallest = None
print('Before:', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
  if smallest is None or itervar < smallest:
    smallest = itervar
  print('Loop:', itervar, smallest)
print('Smallest:', smallest)
```

Maximum and minimum loops

Automatic Stop:

```
n = 10
while n:
  print(n)
  n = n - 1
print('Done!')
```

Forced stop:

```
while True:
  line = input('> ')
  if line == 'done':
    break
  print(line)
print('Done!')
```

Guess the number

Output looks like this:

```
Hello! What is your name?
Albert
Well, Albert, I am thinking of a number between 1
    and 20.
Take a guess.
10
Your guess is too high.
Take a guess.
2
Your guess is too low.
Take a guess.
4
Good job, Albert! You guessed my number in 3
    guesses!
```

Write the program to do this. Use 'random' module to get the secret number.

Guess the number

```
import random
guessesTaken = 0
number = random.randint(1, 20)
print('Think a number between 1 and 20.')
while guessesTaken < 6:
    print('Take a guess.')
    guess = input()
    guess = int(guess)
    guessesTaken = guessesTaken + 1
    if guess < number:
        print('Your guess is too low.')
    if guess > number:
        print('Your guess is too high.')
    if guess == number:
        break
if guess == number:
    guessesTaken = str(guessesTaken)
    print('You guessed in ' + guessesTaken + '
    guesses!')
if guess != number:
    number = str(number)
    print('Nope. The number I was thinking of was
    ' + number)
```

Quiz: Matrix Multiplication

Given X and Y write code to do matrix multiplication

```
X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]

Y = [[5,8,1,2],
    [6,7,3,0],
    [4,5,9,1]]

result = [[0,0,0,0],
          [0,0,0,0],
          [0,0,0,0]]
```

Solution : Matrix Multiplication

```
for i in range(len(X)):
    for j in range(len(Y[0])):
        for k in range(len(Y)):
            result[i][j] += X[i][k] * Y[k][j]

for r in result:
    print(r)
```

Module 4: Collections

Sequences

Lists

Built-in Sequences

- **list** *mutable*, possibly heterogeneous
- **tuple** *immutable*, possibly heterogeneous
- **str** *immutable*, only holds characters

Sequences from Other Packages

- NumPy: **array** *mutable*, homogeneous (like C/Fortran arrays)
- Collections: OrderedDict

Lists

- Ordered collections of arbitrary objects
- Accessed by offset
- Variable length, heterogeneous, arbitrarily nest-able
- Mutable sequence

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
['spam', 2.0, 5, [10, 20]]
```

List mutation

Replace items by assigning them a new value:

```
>>> L = ['U', 'Z', 'H']
>>> L[2] = 'G'
>>> print(L)
['U', 'Z', 'G']
```

Try negative indices.

List mutation

Replace an entire slice:

```
>>> L[1:3] = ['a', 'b']
>>> print(L)
['U', 'a', 'b']
```

The range on the left is replaced by range on the right. If there is mismatch in the lengths? Try ....

Try negative ranges as well. If the range evaluates to null (L[-1:-2]) then right hand array is inserted at the -2 position?

List mutation

New slice does not need to have the same length:

```
>>> L[2:] = range(4)
>>> print(L)
['U', 'a', 0, 1, 2, 3]
```

From 2 onwards there are only 1 (last) element, in its place put all of the right hand side, ie, array from 0 to 4.

List mutation

Add an element

```
>>> L.append(4)
>>> L
['a', 0, 1, 2, 3, 4]
```

Return type of append() is None. It mutates the list itself. It does NOT return the modified list (like Strings).
So don't do:

```
L = L.append(4)
```

List mutation

Remove individual items from a list either by specifying the item:

```
>>> L.remove('U')
>>> L
['a', 0, 1, 2, 3, 4]
```

List mutation

or the position:

```
>>> L.pop(1)
0
>>> L
['a', 1, 2, 3, 4]
```

List mutation

Note: `remove()` method only removes *the first occurrence*:

```
>>> L = ['a', 'b', 'a']
>>> L.remove('a')
>>> L
['b', 'a']
```

List mutation

"del" operator can be used two ways:
Remoes definition of 2nd position, thus the list shrinks.

```
>>> L = ['a', 'b', 'a']
>>> del L[1]
>>> L
['a', 'a']
```

Remove the definition of whole list:

```
>>> del L
>>> L
"Not defined error"
```

List Sorting

Dont use old L.sort() method but newer sorted method.

Here sorted returns the copy of the sorted list keeping original intact.

```
>>> L = [5,3,4,7]
>>> M = sorted(L)
>>> L
[5,3,4,7]
>>> M
[3,4,5,7]
```

List Sorting

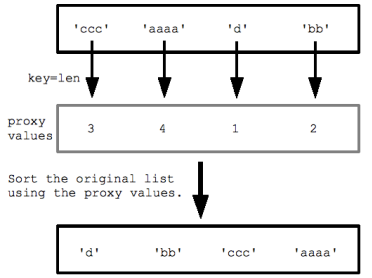
Applies to strings also. The sorted() function can be customized through optional arguments. The sorted() optional argument reverse=True, e.g. sorted(list, reverse=True), makes it sort backwards.

```
strs = ['aa', 'BB', 'zz', 'CC']
  print sorted(strs)  ## ['BB', 'CC', 'aa', 'zz']
     (case sensitive)
  print sorted(strs, reverse=True)   ## ['zz',
     'aa', 'CC', 'BB']
```

List Custom Sorting With key=

- For more complex custom sorting, sorted() takes an optional "key=" specifying a "key" function that transforms each element before comparison. The key function takes in 1 value and returns 1 value, and the returned "proxy" value is used for the comparisons within the sort.
- For example with a list of strings, specifying key=len (the built in len() function) sorts the strings by length, from shortest to longest. The sort calls len() for each string to get the list of proxy length values, and then sorts with those proxy values.

```
  strs = ['ccc', 'aaaa', 'd', 'bb']
  print sorted(strs, key=len)  ## ['d', 'bb',
     'ccc', 'aaaa']
```



(Ref: Google Python class)

Lists operators

You can concatenate two lists using the + operator:

```
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]
```

Lists operators

You can mutate a list in place with the += operator:

```
>>> L = [1, 2]
>>> L += [3, 4]
>>> print(L)
[1, 2, 3, 4]
```

Lists operators

The * operator also works on lists:

```
>>> L = [1, 2]
>>> print(L*3)
[1, 2, 1, 2, 1, 2 ]
```

Traversing a list

Most common, for loop:

```
for cheese in cheeses:
    print(cheese)

for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2

for x in empty:
  print('This never happens.')
```

Exercise

Take two lists, say for example these two:

a = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89] b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]

and write a program that returns a list that contains only the elements that are common between the lists (without duplicates). Make sure your program works on two lists of different sizes.

Lists and functions

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25
```

Average

```
total = 0
count = 0

while (True):
  inp = input('Enter a number: ')
  if inp == 'done': break
  value = float(inp)
  total = total + value
  count = count + 1

average = total / count
print('Average:', average)
```

The 'in' operator

To test for presence of an item in a collection:

- **x in S**: Evaluates to `True` if `x` is equal to a *value* contained in the `S` sequence (list, tuple, set).
- **x in T**: Evaluates to `True` if `x` is a substring of string `T`.

The `in` operator and the `if` conditional

Testing for the existence of an element in a container is a very common pattern: `in` operator:

```
>>> L = [1, 2, 3, 4]
>>> if 1 in L:
...   print("Found!")
...
Found!
```

The `in` operator and the `if` conditional

Equivalent to the following, more verbose, less *pythonic*:

```
 >>> L = [1, 2, 3, 4]
 >>> for item in L:
 ...     if item == 1:
 ...         print("Found!")
 ...         break
```

Zip and Argument Unpacking

- zip transforms multiple lists into a single list of tuples

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

- If the lists are different lengths, zip stops as soon as the first list ends.
- You can also unzip

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
```

- The asterisk performs argument unpacking. Whats the "type" of "letters" and "numbers"?

Lists and strings

- String: sequence of characters
- List: sequence of values
- A list of characters is not the same as a string.
- To convert from a string to a list of characters:

```
>>> s = 'spam'
>>> t = list(s) #The list function breaks a string
    into individual letters.
>>> print(t)
['s', 'p', 'a', 'm']
```

Lists and strings

'Split' to break a string into list of words:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
>>> print(t[2])
```

Lists Exercise

- Download www.py4e.com/code3/romeo.txt
- Open the file and read it line by line.
- Split each line into a list of words using the split function.
- For each word, check to see if the word is already in a list.
- If the word is not in the list, add it to the list.
- Sort and print the resulting words in alphabetical order.

Lists Exercise

```
Enter file: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair',
    'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick',
    'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

Lists operations (recap)

- empty list `L = []`
- four items `L2 = [0, 1, 2, 3]`
- nested `L3 = ['abc', ['def', 'ghi']]`
- index `L2[i]`, `L3[i][j]`
- slice, length `L2[i:j]`, `len(L2)`

- concatenate, repeat `L1 + L2, L2 * 3`
- iteration, membership `for x in L2, 3 in L2`
- methods `L2.append(4), L2.sort(), L2.index(1), L2.reverse()`
- shrinking `del L2[k], L2[i:j] = []`
- assignment `L2[i] = 1, L2[i:j] = [4,5,6]`
- create list `range(4) # useful to loop`

Sequences: List Comprehensions

List Comprehensions

- Compact syntax for *filtering* elements of a list and/or *applying* a function to them.
- To build a list.

Say, you wish to build a list of squares fro 0 to 5.

```
data = []
for num in range(6):
    data.append(num*num)
```

Can be written using *list comprehension*:

```
data = [num*num for num in range(6) ]
```

List Comprehensions

Say, you wish to build a list of squares fro 0 to 5, only for even numbers.

```
data = []
for num in range(6):
    if num%2==0:
        data.append(num*num)
```

Can be written using *list comprehension*:

```
data = [num*num for num in range(6) if num%2==0 ]
```

Example: List Comprehensions

Download https://raw.github.com/gc3-uzh-ch/python-course/master/values.dat.

Build its list.

```
data = []
for num in open('values.dat').readlines():
    data.append(int(num))
```

Can be written using *list comprehension*:

```
data = [ int(line) for line in
     open('values.dat').readlines() ]
```

List comprehensions

The general syntax of a list comprehension is:

```
    [expr for var in iterable if condition]
```

where:

- **expr** is any Python expression;
- **iterable** is a (generalized) sequence;
- **condition** is a boolean expression, depending on ***var***;
- **var** is a variable that will be bound in turn to each item in ***iterable*** which satisfies ***condition***.

*Create a new list, and for each **var** in the sequence **iterable**, if **condition** is true then add **expr** to the list.*

List comprehensions

Need not use all the values

```
zeros = [0 for _ in even_numbers] # just to have
     same size
```

Mutliple for loops:

```
pairs = [(x,y)
    for x in range(10)
      for y in range(10)] # 100 pairs of [(0,0),
     (0,1)...]
```

Exercise

- Write a program which will find all such numbers which are divisible by 7 but are not a multiple of 5, between 2000 and 3200 (both included).
- The numbers obtained should be printed in a comma-separated sequence on a single line.
- Use "join" to make the single string. May need to cast integers by str() to be join-able.

Exercise

- Write a program that calculates and prints the value according to the given formula:
- Q = Square root of [(2 * C * D)/H]
- The fixed values of C and H: C is 50. H is 30.
- D is the variable whose values should be input to your program in a comma-separated sequence.
- Example: Let us assume the following comma separated input sequence is given to the program: 100,150,180
- The output of the program should be: 18,22,24
- Hints: If the output received is in decimal form, it should be rounded off to its nearest value (for example, if the output received is 26.0, it should be printed as 26)

Solution

```
import math
c=50
h=30
value = []
items=[x for x in input("Enter : ").split(',')]
for d in items:

    value.append(str(int(round(math.sqrt(2*c*float(d)/h)))

print(','.join(value))
```

Exercise

- Write a program that accepts a comma separated sequence of words as input and prints the words in a comma-separated sequence after sorting them alphabetically.
- Suppose the following input is supplied to the program:without,hello,bag,world
- Then, the output should be: bag,hello,without,world

Solution

```
items=[x for x in input("Enter : ").split(',')]
items.sort()
print(','.join(items))
```

Exercise

- Write a program that accepts a sequence of whitespace separated words as input and prints the words after removing all duplicate words and sorting them alphanumerically.
- Suppose the following input is supplied to the program: hello world and practice makes perfect and hello world again
- Then, the output should be: again and hello makes perfect practice world
- We use set container to remove duplicated data automatically and then use sorted() to sort the data.

Solution

```
s = input("Enter : ")
words = [word for word in s.split(" ")]
print(" ".join(sorted(list(set(words)))))
```

Exercise

- Write a program which takes 2 digits, X,Y as input and generates a 2-dimensional array. The element value in the i-th row and j-th column of the array should be i*j.
- Note:$i = 0, 1 \ldots, X - 1; j = 0, 1, \ldots Y - 1$.
- Example: Suppose the following inputs are given to the program: 3,5
- Then, the output of the program should be: [[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8]]
- Hints: Note: In case of input data being supplied to the question, it should be assumed to be a console input in a comma-separated form.

Solution

```
input_str = input("Enter : ")
dimensions=[int(x) for x in input_str.split(',')]
rowNum=dimensions[0]
colNum=dimensions[1]
multilist = [[0 for col in range(colNum)] for row
    in range(rowNum)]

for row in range(rowNum):
    for col in range(colNum):
        multilist[row][col]= row*col

print(multilist)
```

Sequences: Tuples

Tuples

- They are like lists but immutable. Why Lists and Tuples?
- When you want to make sure the content won't change.

```
>>> T = (1, 2, 3)
>>> T[0]
1
>>> T[0:1]
(1,)
```

Tuples

But they are *immutable*

```
>>> T[0] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
    assignment
```

Multiple assignment

You can assigning multiple variables at the same time

```
>>> a, b, c = (1, 2, 3)
>>> print(a)
1
>>> print(b)
2
```

Multiple assignment

It works with any sequence:

```
>>> a, b, c = 'UZH'
>>> print(a)
U
```

Can you think of a way to swap the values of two variables using this?

```
>>> a, b = b, a
```

Multiple assignment

In `for`:

```
>>> L = [(1,'a'), (2,'b'), (3, 'c')]
>>> for x, y in L:
...     print ("first is " + str(x)
...             + ' and second is ' + y)
```

Useful with functions that return a tuple.

"returning a comma separated elements creates a tuple. Multiple values can only be returned inside containers. It looks a bit peculiar, but it's actually the comma that forms a tuple, not the parentheses"

(Ref: How does Python return multiple values from a function? - Stack Overflow)

Clarify multiple return values with named tuples

Is this good or bad? You don't know because it's not clear.

```
# Old testmod return value
doctest.testmod()
# (0, 4)
```

Better:

```
# New testmod return value, a namedTuple
doctest.testmod()
# TestResults(failed=0, attempted=4)
```

A namedTuple is a subclass of tuple so they still work like a regular tuple, but are more friendly. To make a namedTuple:

```
TestResults = namedTuple('TestResults', ['failed',
    'attempted'])
```

(Ref: Transforming Code into Beautiful, Idiomatic Python - Raymond Hettinger)

Unpacking sequences

```
p = 'Yogesh', 'Kulkarni', 0x30,
    'python@example.com'

# A common approach / habit from other languages
fname = p[0]
lname = p[1]
age = p[2]
email = p[3]
```

Better:

```
fname, lname, age, email = p
```

This approach uses tuple unpacking and is faster and more readable.

(Ref: Transforming Code into Beautiful, Idiomatic Python - Raymond Hettinger)

Exercise

- Write a program which accepts a sequence of comma-separated numbers from console and generate a list and a tuple which contains every number.
- Suppose the following input is supplied to the program: 34,67,55,33,12,98
- Then, the output should be: ['34', '67', '55', '33', '12', '98'] ('34', '67', '55', '33', '12', '98')

Solution

```
values=input("Enter : ")
l=values.split(",")
t=tuple(l)
print(l)
print(t)
```

Containers

Sets

Sets

- Collection with all elements unique.
- Unordered.
- Variable length, heterogeneous, arbitrarily nest-able (*)
- Ideal for membership queries

```
some_set = {1, 2, 3, 4, 4, 4, 4}
another_set = {4, 5, 6}
print(some_set)
```

(* non mutable tuples can be added but not mutable Lists or Sets)

Empty Set

- Creating an empty set
- Do *not* use the `empty set = {}`
- Use `empty_set = set()`

Sets: Hash tables

- The Python dict is essentially a hash table.
- Essentially the keys are transformed into table positions by a hashing function
- Insertion in set is done through set.add() function, where an appropriate record value is created to store in the hash table.
- Union:- Two sets can be merged using union() function or — operator. Both Hash Table values are accessed and traversed with merge operation perform on them to combine the elements, at the same time duplicates are removed.
- And so on . . .

(Ref: Internal working of Set in Python - Geeks for Geeks)

Unique Membership

```
my_list = [1, 2, 3, 0, 5, 10, 11, 1, 5]
my_set = set(my_list)
print(my_set)
print(len(my_set))
print(len(my_list))

>>> S = set()
>>> S.add(1)
>>> S.add('two')
>>> S.add(1)
>>> S
set([1, 'two'])
```

Checking for membership

```
my_set = {1, 2, 3, 4}
val = 1
print("The value", val ,"appears in the variable
    my_set:", val in my_set)

val = 0
print("The value", val ,"appears in the variable
    my_set:", val in my_set)
```

Immutable Sets

Sets are implemented in a way, which doesn't allow mutable objects. The following example demonstrates that we cannot include for example lists as elements:

```
>>> cities = set(["Frankfurt", "Basel","Freiburg"])
>>> cities.add("Strasbourg")
>>> cities
{'Freiburg', 'Basel', 'Frankfurt', 'Strasbourg'}
>>>
```

Frozensets are like sets except that they cannot be changed, i.e. they are immutable:

```
>>> cities = frozenset(["Frankfurt",
    "Basel","Freiburg"])
>>> cities.add("Strasbourg")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no
    attribute 'add'
>>>
```

Frozensets

Though sets can't contain mutable objects, sets are mutable:

```
>>> cities = set((("Python","Perl"), ("Paris",
    "Berlin", "London")))
>>> cities = set((["Python","Perl"], ["Paris",
    "Berlin", "London"]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```

Sets Operators

```
    set_a.add(x) # add an element to a set

    set_a.remove(x) # remove an element from a set
```

```
set_a - set_b # elements in a but not in b.
 Equivalent to set_a.difference(set_b)
```

What are the operators for Union, Intersection? Intuitively?

Sets Operators

```
set_a | set_b # elements in a or b. Equivalent
 to set_a.union(set_b)

set_a & set_b # elements in both a and b.
 Equivalent to set_a.intersection(set_b)

set_a ^ set_b # elements in a or b but not
 both. Equivalent to
 set_a.symmetric_difference(set_b)

set_a <= set_b # tests whether every element
 in set_a is in set_b. Equivalent to
 set_a.issubset(set_b)
```

Sets Exercise

Try the above yourself using the my_set and another_set variables from above, and compute the difference, union, intersection, and symmetric difference, between the two sets.

```
set_A = {1, 2, 3, 4, 5}
set_B = {4, 5, 6, 7}
print("Set A", set_A)
print("Set B", set_B)
print("Difference A-B", ... )
print("Union", ...)
print("Intersection", ...)
print("Symmetric Difference", ...)
```

Dictionaries

Dictionaries

- Dictionaries, sometimes called dicts, maps, or, rarely, hashes are data structures containing key-value pairs.
- Dictionaries have a set of unique keys and are used to retrieve the value information associated with these keys.
- Lookup into a dictionary is very efficient.
- Accessed by key, not offset
- Un-ordered collections of arbitrary objects
- Variable length, heterogeneous, arbitrarily nest-able

Dictionaries

- Dictionaries are specified by curly braces, , containing zero or more comma separated key-value pairs, where the keys and values are separated by a colon, :
- Like a list, values for a particular key are retrieved by passing the query key into square brackets.

```
>>> D = { }
>>> D['a'] = 1
>>> D[2] = 'b'
>>> D
{'a': 1, 2: 'b'}
```

Dictionaries can be created and initialized using the following syntax:

```
>>> D = { 'a':1, 2:'b' }
>>> D['a']
1
```

Exercise

- Write a Python program to count the number of characters (character frequency) in a string. Sample String : google.com' Expected Result : 'o': 3, 'g': 2, '.': 1, 'e': 1, 'l': 1, 'm': 1, 'c': 1

Dictionary

The `for` statement can be used to loop over keys of a dictionary:

```
>>> D = { 'a':1, 'b':2 }
>>> for val in D.keys():
...   print(val)
'a'
'b'
```

Loop over dictionary *keys.*

The `.keys()` part can be omitted, as it's the default!

If you want to loop over dictionary *values*, you have to explicitly request it.

```
>>> D = { 'a':1, 'b':2 }
>>> for val in D.values():
...   print(val)
1
2
```

Loop over dictionary *values*

The `.values()` cannot be omitted!

Dictionaries Exercise

- Find the common keys in 'a_dict' and 'b_dict'
- Find the common values in 'a_dict' and 'b_dict'

```
a_dict = {"a":"e", "b":5, "c":3, "d": 4}
b_dict = {"c":5, "d":6}

# your code here

print("Common keys", ...)
print("Common values", ...)
```

Exercise

- With a given integral number n, write a program to generate a dictionary that contains (i, i*i) such that is an integral number between 1 and n (both included). and then the program should print the dictionary.
- Suppose the following input is supplied to the program:8
- Then, the output should be:1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64

Solution

```
n=int(input("Enter : "))
d=dict()
for i in range(1,n+1):
    d[i]=i*i

print(d)
```

Exercises

- Write a function `invert(D)` that takes a dictionary `D` and returns a dictionary `Dinv` with keys and values swapped. (We assume that `D` is 1-1.)
- Example correct output:

```
>>> D = { 'CH':'Switzerland', 'I':'Italy',
     'F':'France' }
>>> Dinv = # Your code here
>>> print(Dinv)
{ 'Switzerland':'CH', 'France':'F',
     'Italy':'I' }
```

Dictionaries and files

Counting words in a given text

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Dictionaries can be created and initialized using the following syntax:

```
fhand = open(fname)

counts = dict()
for line in fhand:
  words = line.split()
  for word in words:
    if word not in counts:
      counts[word] = 1
    else:
        counts[word] += 1
print(counts)
```

Default Dict

- Is like a regular dictionary, except that when you try to look up a key it doesn't contain, it first adds a value for it using a zero-argument function you provided when you created it.

```
word_counts = defaultdict(int) # assigns 0
     for the newly added key
for word in document:
  word_counts[word] += 1
```

- Can also be useful with list or dict or even your own functions

```
ddlist = defaultdict(list) # assigns empty
     list for the newly added key
ddlist[2].append(1)
```

- When 2 was not present, added it with value as empty list, to which 1 was added

Without Default Dict

Without:

```
names = ['raymond', 'rachel', 'matthew', 'roger',
         'betty', 'melissa', 'judith', 'charlie']
```

```
# In this example, we're grouping by name length
d = {}
for name in names:
    key = len(name)
    if key not in d:
        d[key] = []
    d[key].append(name)

# {5: ['roger', 'betty'], 6: ['rachel', 'judith'],
     7: ['raymond', 'matthew', 'melissa',
     'charlie']}

d = {}
for name in names:
    key = len(name)
    d.setdefault(key, []).append(name)
```

With Default Dict

Better:

```
names = ['raymond', 'rachel', 'matthew', 'roger',
         'betty', 'melissa', 'judith', 'charlie']

d = defaultdict(list)
for name in names:
    key = len(name)
    d[key].append(name)
```

Counter

- A Counter turns a sequence of values into a defaultdict(int)-like object mapping keys to counts.

```
from collections import Counter
c = Counter([0, 1, 2, 0]) # c is (basically)
     { 0 :  2, 1 : 1, 2 : 1}
```

- This gives us a very simple way to solve our word_counts problem

```
word_counts = Counter(document)
```

The 'in' operator

- `x in D`; `x in D.keys()`: Evaluates to `True` if `x` is equal to a *key* in the `D` dictionary.
- `x in D.values()`: Evaluates to `True` if `x` is equal to a *value* in the `D` dictionary.

Combining (Nesting) Data Structures

- There are many opportunities to combine data types.
- Lists can be populated by arbitrary data structures.
- Similarly, you can use any type as the value in a dictionary.

```
print("lists of lists")
lol = [[1, 2, 3], [4, 5, 6, 7]]
lol_2 = [[4, 5, 6], [7, 8, 9]]
print("lists of lists of lists")
lolol = [lol, lol_2]
print("Lolol:", lolol)
print("data structures as values in a dictionary")
dlol = {"lol":lol, "lol_2":lol_2}
print(dlol)
print("retrieving data from this dictionary")
print(dlol["lol"])
print(dlol["lol"][0])
print(dlol["lol"][0][0])
```

Exercise

You are given the following data structure.

```
data = {
    "Panos": {
        "Job":"Professor",
        "YOB": "1976",
        "Children": ["Gregory", "Anna"]
        },
    "Joe": {
        "Job":"Data Scientist",
        "YOB": "1981"
        }
    }
```

Exercise

You need to write code that

- Prints the job of Joe
- Prints the year of birth of Panos; prints the age of Panos
- Prints the children of Panos
- Prints the second child of Panos
- Prints the number of people_entries_ in the data. (Notice that it is much harder to find all the people in the data, eg the children)
- Checks if Maria is in the data
- Checks if Panos has children. Would you code work when the list of children is empty?
- Checks if Joe has children. How can you handle the lack of the corresponding key?

Exercise

- You need to write code that categorizes each mail message by which day of the week the mail came. Sample line: `From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008`
- Sample answer: `{'Fri': 20, 'Thu': 6, 'Sat': 1}`

Dictionaries operations (recap)

- empty `d1 = {}`
- two-item `d2 = {'spam': 2, 'eggs': 3}`
- nesting `d3 = {'food': {'ham': 1, 'egg': 2}}`
- indexing `d2['eggs'], d3['food']['ham']`
- methods `d2.has key('eggs'), d2.keys(), d2.values()`
- length `len(d1)`
- add/change `d2[key] = new`
- deleting `del d2[key]`

Module 5: Procedures

Functions

Function calls

- A function is a named sequence of statements that performs a computation.
- When you define a function, you specify the name and the sequence of statements.
- Later, you can "call" the function by name.

```
>>> type(32)
<class 'int'>
```

- The name of the function is type.
- The expression in parentheses is called the argument of the function.
- The argument is a value or variable that we are passing into the function as input to the function.
- The result, for the type function, is the type of the argument.

Built-in functions

Python provides a set of functions already built-in. You are already very familiar with one of them:

```
print(''Oh yeah, I am a function!\n'')

name = input(''I get inputs from you! What is your
    name?'')
print(''Hello'', name)

nums = [3, 41, 12, 9, 74, 15]
print(''Length:'', len(nums))
print(''Max:'', max(nums))
print(''Min:'', min(nums))
print(''Sum:'', sum(nums))

list(range(-10,10,2))

a = [0.819, 0.277, 0.817, 0.575, 0.168, 0.973,
    0.987, 0.883, 0.293, 0.933]
# Keep only numbers above 0.5 and round them to 2
    decimals
b = [round(num,2) for num in a if num>0.5]
print(sorted(b))
```

Functions from Libraries

We can also add more functions by import-ing libraries, like the math library.

```
# Let's have some fun
import math
for i in range(32):
    # math.fabs returns the absolute value
    # math.cos returns the cosine of the value
    x = int(math.fabs((i*math.cos(i/4)))+1)
    print(x*'#')

import random
for i in range(10):
    x = random.random()
    print(round(x,3))
```

User Defined Functions

A function in Python is defined by a def statement. The general syntax looks like this:

```
def function-name(Parameter list):
    statements, i.e. the function body
```

- The parameter list consists of none or more parameters.
- Parameters are called arguments, if the function is called.
- The function body consists of indented statements.
- The function body gets executed every time the function is called.

User Defined Functions

Functions assign a name to a block of code the way variables assign names to bits of data.

```
def print_hi():
    print(''hi!'')

for i in range(10):
    print_hi()

def hi_you(name):
    print(''HI {n}!''.format(n=name.upper()))

hi_you(''David'')
```

The 'return' statement

- Function bodies can contain one or more return statement.
- They can be situated anywhere in the function body.
- A return statement ends the execution of the function call and "returns" the result, i.e. the value of the expression following the return keyword, to the caller.
- If the return statement is without an expression, the special value None is returned.
- If there is no return statement in the function code, the function ends, when the control flow reaches the end of the function body and the value "None" will be returned.

The 'return' statement

Example of computing a math function:

```
def square(num):
    squared = num*num
    return squared

x = square(123232)
print(x)

for i in range(15):
    print(''The square of {a} is
     {aa}''.format(a=i, aa=square(i)))
```

Returning Multiple Values

- A function can return exactly one value, or we should better say one object.
- An object can be a numerical value, like an integer or a float.
- But it can also be e.g. a list or a dictionary.
- So, if we have to return for example 3 integer values, we can return a list or a tuple with these three integer values.
- This means that we can indirectly return multiple values.
- If only "," are seen, thats actually a tuple.

Exercises

- Write a function 'in_range' that checks if a number 'n' is within a given range '(a,b)' and returns True or False.
- The function takes n, a, and b as parameters.

Exercise

- Insert at least two distinct functions.
- Factorial
- Fibonacci

Solution

Factorial

```
def fact(x):
    if x == 0:
        return 1
    return x * fact(x - 1)

x=int(input())
print(fact(x))
```

Solutions

Fibonacci

```
    def fibonacci(n):
        if n == 0:
            return 1
        elif n == 1:
            return 1
        else:
            return fibonacci(n-1)+ fibonacci(n-2)
```

Exercise

Please write a program to shuffle and print the list [3,6,7,8].
Hints: Use random.shuffle() function to shuffle a list.

Solution

```
from random import shuffle
li = [3,6,7,8]
shuffle(li)
print(li)
```

Exercise

- Please write a binary search function which searches an item in a sorted list.
- The function should return the index of element to be searched in the list.
- Hints: Use if/elif to deal with conditions.

Solution

```
import math
def bin_search(li, element):
    bottom = 0
    top = len(li)-1
    index = -1
    while top>=bottom and index==-1:
        mid = int(math.floor((top+bottom)/2.0))
        if li[mid]==element:
            index = mid
        elif li[mid]>element:
            top = mid-1
        else:
            bottom = mid+1

    return index

li=[2,5,7,9,11,17,222]
print(bin_search(li,11))
print(bin_search(li,12))
```

Example function: Cleaning up a string

```
def clean(phone):
    result = ''''
    digits =
     {''0'',''1'',''2'',''3'',''4'',''5'',''6'',''7'',''8'',''9''}
    for c in phone:
        if c in digits:
            result = result + c
    return result
p = ''(800) 555-1214 Panos Phone number''
print(clean(p))
```

Output??

Exercise

- Write a program that computes the value of a+aa+aaa+aaaa with a given digit as the value of a.
- Suppose the following input is supplied to the program: 9
- Then, the output should be: 11106
- Hints: int cast on tuple of numbers, yields the number

Solution

```
a = input()
n1 = int("{}".format(a))
n2 = int("{}{}".format(a,a) )
n3 = int("{}{}{}".format(a,a,a) )
n4 = int("{}{}{}{}".format(a,a,a,a) )
print(n1+n2+n3+n4)
```

Variable number of arguments

Some functions can take a variable number of arguments:

- **sum**($[x_0, \ldots, x_n]$) Return $x_0 + \cdots + x_n$.
- **max**($x_0, \ldots, x_n$) Return the maximum of the set $\{x_0, \ldots, x_n\}$
- **min**($x_0, \ldots, x_n$) Return the minimum of the set $\{x_0, \ldots, x_n\}$

Examples:

```
>>> sum([1,2,3])
6
>>> min(1,2,3)
1
>>> max(1,2)
2
```

The most important function of all

help(fn) Display help on the function named `fn` What happens if you type these at the prompt?

- `help(abs)`
- `help(max)`

Without any argument, `help()`: starts an interactive help prompt.

```
 >>> help()
```

Default values

Function arguments can have default values.

```
>>> def hello(name='world'):
...   print (''Hello, '' + name)
...
>>> hello()
'Hello, world'
```

Named arguments

Python allows calling a function with named arguments:

```
hello(name=''Alice'')
```

When passing arguments by name, they can be passed in any order:

```
>>> from fractions import Fraction
>>> Fraction(numerator=1, denominator=2)
Fraction(1, 2)
>>> Fraction(denominator=2, numerator=1)
Fraction(1, 2)
```

Variable Scope Rules

Whatever happens in a function stays in a function

- Variables set inside of functions are scoped to those functions: changes, including any new variables created, are only accessible inside in the function.
- If "outside" variables are modified inside a function's context, the contents of that variable are first copied.
- Similarly, changes or modifications to a function's arguments aren't reflected once the scope is returned; The variable will continue to point to the original thing.
- However, it is possible to modify the thing that is passed, assuming that it is mutable.

Variable Scope

Whatever happens in a function stays in a function

```
def times_two(inp):
    inp = 2*inp
    return inp

variable_four = 4
print(times_two(variable_four))
print(variable_four)
```

Variable Scope

```
def f():
    print(s)
s = ''I love Paris in the summer!''
f()
```

- The variable s is defined as the string "I love Paris in the summer!", before calling the function f().
- The body of f() consists solely of the "print(s)" statement.
- As there is no local variable s, i.e. no assignment to s, the value from the global variable s will be used.
- So the output will be the string "I love Paris in the summer!".

Variable Scope

- What will happen, if we change the value of s inside of the function f()?
- Will it affect the global variable as well?

```
def f():
    s = ''I love London!''
    print(s)

s = ''I love Paris!''
f()
print(s)

I love London!
I love Paris!
```

Variable Scope

```
name = ''panos''
def do_something():
    print(''We are now in the function!'')
    name = ''not panos''
    print(name)
    print(''something! ... and we are out'')

print(''We start here!'')
print(''The name is'', name)
print(''Let's call the function...'')
do_something()
print(''Done with the function...'')
print(name)
```

Variable Scope

- What if we combine the first example with the second one, i.e. we first access s with a print() function, hoping to get the global value, and then assigning a new value to it?
- Assigning a value to it, means - as we have previously stated - creating a local variable s.
- So, we would have s both as a global and a local variable in the same scope, i.e. the body of the function.
- Python fortunately doesn't allow this ambiguity. So, it will throw an error

```
>>> def f():
...   print(s)
...   s = ''I love London!''
...   print(s)
...
>>> s = ''I love Paris!''
>>> f()
Traceback (most recent call last):
  File ''<stdin>'', line 1, in <module>
  File ''<stdin>'', line 2, in f
UnboundLocalError: local variable 's' referenced
    before assignment
>>>
```

Variable Scope

- A variable can't be both local and global inside of a function.
- So Python decides that we want a local variable due to the assignment to s inside of f(), so the first print statement before the definition of s throws the error message above.
- Any variable which is changed or created inside of a function is local, if it hasn't been declared as a global variable.
- To tell Python, that we want to use the global variable, we have to explicitly state this by using the keyword "global"

Variable Scope

```
def f():
    global s
    print(s)
    s = ''Only in spring, but London is great as
     well!''
    print(s)

s = ''I am looking for a course in Paris!''
```

```
f()
print(s)

I am looking for a course in Paris!
Only in spring, but London is great as well!
Only in spring, but London is great as well!
```

Variable Scope

Local variables of functions can't be accessed from outside, when the function call has finished:

```
def f():
    s = ''I am globally not known''
    print(s)

f()
print(s)


I am globally not known
Traceback (most recent call last):
  File ''ex.py'', line 6, in <module>
    print(s)
NameError: name 's' is not defined
```

Variable Scope

The following example shows a wild combination of local and global variables and function parameters:

```
def foo(x, y):
    global a
    a = 42
    x,y = y,x
    b = 33
    b = 17
    c = 100
    print(a,b,x,y)

a, b, x, y = 1, 15, 3,4
foo(17, 4)
print(a, b, x, y)

42 17 4 17
42 15 3 4
```

Variable Scope

```
# composite data structures (lists, sets,
    dictionaries) _can_ be modified
def add_sum(parameter_list):
    s = sum(parameter_list)
    parameter_list.append(s)
    return s

a_list = [1,2,3]
total = add_sum(a_list)
print(total)
print(a_list)

# try again!
tot = add_sum(a_list)
print(tot)
print(a_list)
```

Variable Scope

```
def append_to_sequence (myseq):
    myseq += (9,9,9)
    return myseq

tuple1 = (1,2,3) # tuples are immutable
list1 = [1,2,3] # lists are mutable

tuple2 = append_to_sequence(tuple1)
list2 = append_to_sequence(list1)

print 'tuple1 = ', tuple1 # outputs (1, 2, 3)
print 'tuple2 = ', tuple2 # outputs (1, 2, 3, 9,
    9, 9)
print 'list1 = ', list1 # outputs [1, 2, 3, 9, 9,
    9]
print 'list2 = ', list2 # outputs [1, 2, 3, 9, 9,
    9]
```

Variable Scope

- myseq is a local variable of the append_to_sequence function, but when this function gets called, myseq will nevertheless point to the same object as the variable that we pass in (t or l in our example).
- If that object is immutable (like a tuple), there is no problem. The += operator will cause the creation of a new tuple, and myseq will be set to point to it.
- However, if we pass in a reference to a mutable object, that object will be manipulated in place (so myseq and l, in our case, end up pointing to the same list object).

Parameters and Arguments

- Very often the terms parameter and argument are used synonymously, but there is a clear difference.
- Parameters are inside functions or procedures, while arguments are used in procedure calls, i.e. the values passed to the function at run-time.

"Call by value" and "Call by reference/-name"

- The evaluation strategy for arguments, i.e. how the arguments from a function call are passed to the parameters of the function, differs from programming language to programming language.
- The most common evaluation strategies are "call by value" and "call by reference"

"Call by Value"

- Sometimes also called pass-by-value.
- Used in C and C++
- The argument expression is evaluated, and the result of this evaluation is bound to the corresponding variable in the function.
- So, if the expression is a variable, its value will be assigned (copied) to the corresponding parameter.
- This ensures that the variable in the caller's scope will be unchanged when the function returns.

"Call by Reference"

- Sometimes also called pass-by-reference.
- Used in C and C++ (& in signature)
- A function gets an implicit reference to the argument, rather than a copy of its value.
- As a consequence, the function can modify the argument, i.e. the value of the variable in the caller's scope can be changed.

Python: "Call by ??"

- "Call-by-Object", sometimes also called "Call by Object Reference" or "Call by Sharing".
- If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like call-by-value. ["Why Integers are Immutable?" (next slide)]
- The object reference is passed to the function parameters.
- They can't be changed within the function, because they can't be changed at all, i.e. they are immutable.

"Why Integers are Immutable?'

- Making integers mutable would be very counter-intuitive to the way we are used to working with them.

```
a = 1       # assign 1 to a
b = a+2     # assign 3 to b, leave a at 1
```

- After these assignments are executed we expect a to have the value 1 and b to have the value 3.
- The addition operation is creating a new integer value from the integer stored in a and an instance of the integer 2.
- If the addition operation just took the integer at a and just mutated it then both a and b would have the value 3.
- So we expect arithmetic operations to create new values for their results - not to mutate their input parameters.

(Ref: https://stackoverflow.com/questions/37535694/why-are-integers-immutable-in-python)

"Why Integers are Immutable?'

- However, there are cases where mutating a data structure is more convenient and more efficient.
- Let's suppose for the moment that list.append(x) did not modify list but returned a new copy of list with x appended. Then a function like this:

```
def foo():
  nums = []
  for x in range(0,10):
    nums.append(x)
  return nums
```

- Would just return the empty list. (Remember - here nums.append(x) doesn't alter nums - it returns a new list with x appended. But this new list isn't saved anywhere.)

"Why Integers are Immutable?'

- We would have to write the foo routine like this:

```
def foo():
  nums = []
  for x in range(0,10):
    nums = nums.append(x)
  return nums
```

- If we forget the return then?? A consequence to making lists mutable is that after these statements:

```
a = [1,2,3]
b = a
a.append(4)
```

- The list b has changed to [1,2,3,4].

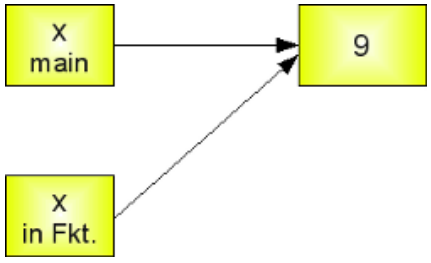
Python: "Call by ??"

- It's different, if we pass mutable arguments. They are also passed by object reference, but they can be changed in place in the function.
- If we pass a list to a function, we have to consider two cases:
  - Elements of a list can be changed in place, i.e. the list will be changed even in the caller's scope.
  - If a new list is assigned to the name, the old list will not be affected, i.e. the list in the caller's scope will remain untouched.

Python: "Call by ??"

- First, let's have a look at the integer variables.
- The parameter inside of the function remains a reference to the arguments variable, as long as it is not changed.
- Say, in the main scope, x has the identity 41902552.
- In the first print statement of the ref_demo() function, the x from the main scope is used, because we can see that we get the same identity.

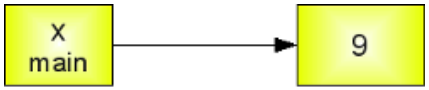
```
x = 9
def ref_demo(x):
    print("x=",x," id=",id(x))
    x=42
    print("x=",x," id=",id(x))
```



Python: "Call by ??"

- As soon as a new value will be assigned to it, Python creates a separate local variable.
- The caller's variable will not be changed this way:
- After we have assigned the value 42 to x, x gets a new identity 41903752, i.e. a separate memory location from the global x.
- So, when we are back in the main scope x has still the original value 9.

```
x = 9
def ref_demo(x):
    print("x=",x," id=",id(x))
    x=42
    print("x=",x," id=",id(x))
```



Python: "Call by ??"

- This means that Python initially behaves like call-by-reference, but as soon as we are changing the value of such a variable, i.e. as soon as we assign a new object to it, Python "switches" to call-by-value.
- This means that a local variable x will be created and the value of the global variable x will be copied into it.

```
>>> x = 9
>>> id(x)
9251936
>>> ref_demo(x)
x= 9  id= 9251936
x= 42  id= 9252992
>>> id(x)
9251936
>>>
```

(Ref: https://www.python-course.eu/python3_passing_arguments.ph

Recap (Functions)

- Indentation is used to delimit blocks of code!
- Variables are just names, a reference to real object.
- `def function(arg1, arg2, ...):` to define functions
- `import filename` to use code from other files.
- To get information on something: `help(something)`

Exceptions

Exceptions

- "Exceptions" are error conditions.
- Code that intercepts some error conditions and reacts.
- Inherits from the built-in Exception class.

```
class NewKindOfError(Exception):
   pass
```

- Exceptions are handled by class name, so they usually do not need any new methods (although you are free to define some if needed).

What does an exception look like?

```
>>> stream.write('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: File not open for writing
```

This is the exception *message*: it is supposed to be read by the (human) user.

Syntax

```
try:
  # code that might raise an exception
except SomeException:
  # handle some exception
except AnotherException as ex:
  # the actual Exception instance
  # is available as variable 'ex'
finally:
  # performed on exit in any case
```

The optional finally clause is executed on exit from the try or except block in *any* case.

Raising exceptions

- Use the raise statement with an `Exception` instance:

```
if an_error_occurred:
  raise AnError("Spider sense is tingling.")
```

- The exception class name specifies what kind of exception to raise.
- You must use an *existing* exception class; we shall learn how to define new exceptions in the object-orientation lectures.
- The exception message is an arbitrary string. It is meant for humans to read, so try to describe the error condition clearly and concisely.

Raising exceptions

Within an except clause, you can use raise with no arguments to re-raise the current exception:

```
try:
  something()
except ItDidntWork:
  do_cleanup()
  # re-raise exception to caller
  raise
```

raise statement

Throw or raise an exception.

- Forms:
  - `raise instance`
  - `raise MyExceptionClass(value)` – preferred.
  - `raise MyExceptionClass, value`
- The `raise` statement takes:
  - An (instance of) a built-in exception class.
  - An instance of class `Exception` or
  - An instance of a built-in subclass of class `Exception` or
  - An instance of a user-defined subclass of class `Exception` or
  - One of the above classes and (optionally) a value (for example, a string or a tuple).

Exercise

Write a function to compute 5/0 and use try/except to catch the exceptions.

Solution

```
def throws():
    return 5/0

try:
    throws()
except ZeroDivisionError:
    print "division by zero!"
except Exception as err:
    print 'Caught an exception'
finally:
    print 'In finally block for cleanup'
```

Exercise

Define a custom exception class which takes a string message as attribute.
Hints:
To define a custom exception, we need to define a class inherited from Exception.

Solution

```
class MyError(Exception):
    """My own exception class

    Attributes:
        msg  -- explanation of the error
    """

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

error = MyError("something wrong")
```

Module 6: Object Oriented Programming

Object-Oriented Programming: Concepts

Managing Larger Programs

Procedural programming is:

- Sequential code
- Conditional code (if statements)
- Repetitive code (loops)
- Store and reuse (functions)

In a very large codebase, object oriented programming is a way to arrange your code so that you can zoom into 500 lines of the code, and understand it while ignoring the other 999,500 lines of code for the moment.

Starting with Programs

- One way to think about object oriented programming is that we are separating our program into multiple "zones".
- Each "zone" contains some code and data (like a program) and has well defined interactions with the outside world and the other zones within the program.

Sample program

```
import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup
import ssl

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url,
    context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
  print(tag.get('href', None))
```

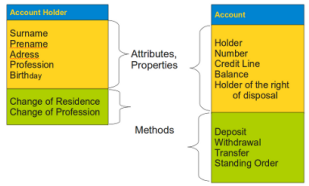
Subdividing a Problem

- One of the advantages of the object oriented approach is that it can hide complexity.
- For example, while we need to know how to use the urllib and BeautifulSoup code, we do not need to know how those libraries work internally.
- It allows us to focus on the part of the problem we need to solve and ignore the other parts of the program.

What's an *object*?

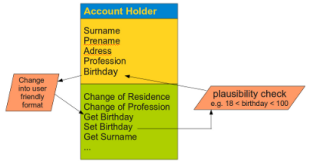
A Python object is a bundle of variables and functions.

- What variable names and functions comprise an object is defined by the object's *class*.
- From one class specification, many objects can be *instantiated*. Different instances can assign different values to the object variables.
- Variables and functions in an instance are collectively called *instance attributes*; functions are also termed *instance methods*.



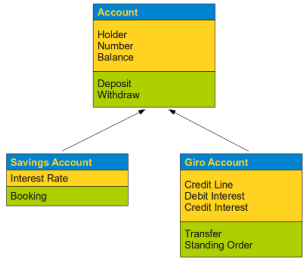
Encapsulation of Data

- Encapsulation is the mechanism for restricting the access to some of an object's components, this means that the internal representation of an object can't be seen from outside of the objects definition.
- Access to this data is typically only achieved through methods



Inheritance

- A class can inherit attributes and behaviour (methods) from other classes, called super-classes.
- Inheritance: to create new classes by using existing classes.
- New ones can both be created by extending and by restricting the existing classes.



Using Objects

Python provides us with many built-in objects.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))
```

The first line is constructing an object of type list, the second and third lines are calling the `append()` method, the fourth line is calling the `sort()` method, and the fifth line is retrieving the item at position 0. The sixth line is calling the `__getitem__()` method in the stuff list with a parameter of zero.

Example

Import the `date` class from the standard library module `datetime`

```
>>> from datetime import date
>>> dt1 = date(2012, 9, 28)
>>> dt2 = date(2012, 10, 1)
```

To instantiate an object, call the class name like a function.

```
>>> from datetime import date
>>> dt1 = date(2012, 9, 28)
>>> dt2 = date(2012, 10, 1)
```

Example

```
>>> dir(dt1)
['__add__', '__class__', 'ctime', 'day',
'fromordinal', 'fromtimestamp', 'isocalendar',
'isoformat', 'isoweekday', 'max', 'min', 'month',
'replace', 'resolution', 'strftime', 'timetuple',
'today', 'toordinal', 'weekday', 'year']
```

The `dir` function can list all objects attributes. Note there is no distinction between instance variables and methods! Access to object attributes is done by suffixing the instance name with the attribute name, separated by a dot ".".

```
>>> dt1.day
28
>>> dt1.month
9
>>> dt1.year
2012
```

Starting with OOP

```
class PartyAnimal:
  x = 0
  def party(self) :
    self.x = self.x + 1
    print("So far",self.x)

an = PartyAnimal()
an.party()
an.party()
an.party()
PartyAnimal.party(an)
```

When the party method is called, the first parameter (self) points to the particular instance of the PartyAnimal object that party is called from within. Within the party method, we see the line: `self.x = self.x + 1`

Starting with OOP

Following line is another way to call the party method within the an object: `PartyAnimal.party(an)` When the program executes, it produces the following output:

```
So far 1
So far 2
So far 3
So far 4
```

The `self` argument

Every method of a Python object always has `self` as first argument.

However, you do not specify it when calling a method: it's automatically inserted by Python:

```
>>> class ShowSelf(object):
...   def show(self):
...     print(self)
...
>>> x = ShowSelf() # construct instance
>>> x.show() # 'self' automatically inserted!
<__main__.ShowSelf object at 0x299e150>
```

The `self` name is a reference to the object instance itself. You *need to* use `self` when accessing methods or attributes of this instance.

Member functions

There are different types of methods/functions.

```
class MyClass:
    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

Instead of using a plain class MyClass: declaration you may choose to declare a new-style class inheriting from object with the class MyClass(object):

Member functions: Instance Methods

```
class MyClass:
    def method(self):
        return 'instance method called', self
```

Through the self parameter, instance methods can freely access attributes and other methods on the same object.

Member functions: Class Methods

This method is with a @classmethod decorator to flag it as a class method.

```
class MyClass:
    @classmethod
    def classmethod(cls):
        return 'class method called', cls
```

- Instead of accepting a self parameter, class methods take a cls parameter that points to the class-and not the object instance-when the method is called.
- Because the class method only has access to this cls argument, it can't modify object instance state.
- That would require access to self. However, class methods can still modify class state that applies across all instances of the class.

Member functions: Static Methods

Marked with a @staticmethod decorator to flag it as a static method.

```
class MyClass:
    @staticmethod
    def staticmethod():
        return 'static method called'
```

- This type of method takes neither a self nor a cls parameter (but of course it's free to accept an arbitrary number of other parameters).
- Therefore a static method can neither modify object state nor class state. Static methods are restricted in what data they can access - and they're primarily a way to namespace your methods.

Member functions: Instance Methods

```
>>> obj = MyClass()
>>> obj.method()
('instance method called', <MyClass instance at
    0x101a2f4c8>)
```

This confirmed that method (the instance method) has access to the object instance (printed as "MyClass instance") via the self argument. When the method is called, Python replaces the self argument with the instance object, obj.

Member functions: Instance Methods

We could ignore the syntactic sugar of the dot-call syntax (obj.method()) and pass the instance object manually to get the same result:

```
>>> MyClass.method(obj)
('instance method called', <MyClass instance at
    0x101a2f4c8>)
```

Member functions: Class Methods

```
>>> obj.classmethod()
('class method called', <class MyClass at
    0x101a2f4c8>)
```

- Calling classmethod() showed us it doesn't have access to the "MyClass instance" object, but only to the "class MyClass" object, representing the class itself (everything in Python is an object, even classes themselves).
- Please note that naming these parameters self and cls is just a convention. You could just as easily name them the_object and the_class and get the same result.

Member functions: Static Methods

```
>>> obj.staticmethod()
'static method called'
```

- Aren't we surprised when they learn that it's possible to call a static method on an object instance.
- Behind the scenes Python simply enforces the access restrictions by not passing in the self or the cls argument when a static method gets called using the dot syntax.

- This confirms that static methods can neither access the object instance state nor the class state.
- They work like regular functions but belong to the class's (and every instance's) namespace.

Member functions: Cases

When we attempt to call these methods on the class itself - without creating an object instance beforehand:

```
>>> MyClass.classmethod()
('class method called', <class MyClass at
    0x101a2f4c8>)

>>> MyClass.staticmethod()
'static method called'

>>> MyClass.method()
TypeError: unbound method method() must
    be called with MyClass instance as first
    argument (got nothing instead)
```

- We didn't create an object instance and tried calling an instance function directly on the class blueprint itself.
- This means there is no way for Python to populate the self argument and therefore the call fails.

Member functions: Why Static methods?

```
import math

class Pizza:
    def __init__(self, radius, ingredients):
        self.radius = radius
        self.ingredients = ingredients

    def __repr__(self):
        return (f'Pizza({self.radius!r}, '
                f'{self.ingredients!r})')

    def area(self):
        return self.circle_area(self.radius)

    @staticmethod
    def circle_area(r):
        return r ** 2 * math.pi
```

Instead of calculating the area directly within area(), using the well-known circle area formula, I factored that out to a separate circle_area() static method.

Member functions: Why Static methods?

- As don't take a cls or self argument. That's a big limitation- but it's also a great signal to show that a particular method is independent from everything else around it.
- Now, why is that useful?
- Flagging a method as a static method is not just a hint that a method won't modify class or instance state - this restriction is also enforced by the Python runtime.
- Techniques like that allow you to communicate clearly the intention.

Member functions: Properties

getters and setters:

```
class TestProperty(object):
    def __init__(self, description):
        self._description = description
    @property
    def description(self):
        print 'getting description'
        return self._description
    @description.setter
    def description(self, description):
        print 'setting description'
        self._description = description
```

We mark the instance variable as private by prefixing it with an underscore.
The name of the instance variable and the name of the property must be different. If they are not, we get recursion and an error.

Member function: __init__

- "__init__" is a reseved method in python classes.
- Python doesn't have explicit constructors like C++ or Java, but the __init__() method in Python is something similar, though it is strictly speaking not a constructor.
- It behaves in many ways like a constructor, e.g. it is the first code which is executed, when a new instance of a class is created.

```
class Car(object):
  def __init__(self, model, color, company,
     speed_limit):
    self.color = color
    self.company = company
    self.speed_limit = speed_limit
    self.model = model
```

Member function: Destructor

- There is no "real" destructor, but something similiar, i.e. the method __del__.
- It is called when the instance is about to be destroyed.
- If a base class has a __del__() method, the derived class's __del__() method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.

```
class Greeting:
    def __init__(self, name):
        self.name = name
    def __del__(self):
        print "Destructor started"
    def SayHello(self):
        print "Hello", self.name
```

Member function: Example Construction Destruction

```
>>> from hello_class import Greeting
>>> x1 = Greeting("Guido")
>>> x2 = x1
>>> del x1
>>> del x2
Destructor started
```

- If we use this class, we can see, the "del" doesn't directly call the __del__() method.
- It's apparent that the destructor is not called, when we delete x1.
- The reason is that del decrements the reference count for the object of x1 by one.
- Only if the reference count reaches zero, the destructor is called

No access control

There are no "public", "private" and 'protected". qualifiers for object attributes.

***Any* code can create/read/overwrite/delete *any* attribute on *any* object.**

There are *conventions*, though:

- "protected" attributes: _name_
- "private" attributes: __name__

(But again, note that this is not *enforced* by the system in any way.)

No overloading

- Python does not allow overloading of functions.
- Any function.
- Hence, no overloading of constructors.
- So: **a class has one and only one constructor.**

Constructor chaining

When a class is instantiated, Python only calls the first constructor it can find in the class inheritance call-chain.
If you need to call a parent constructor, you need to do it *explicitly*:

```
class Application(Task):
  def __init__(self, ...):
    # do Application-specific stuff here
    Task.__init__(self, ...)
    # some more Application-specific stuff
```

Calling a superclass constructor is optional, and it can happen anywhere in the init method body.

Classes as Types

All variables have a type. And we can use the built-in dir function to examine the capabilities of a variable. We can use type and dir with the classes that we create.

```
class PartyAnimal:
  x = 0
  def party(self) :
    self.x = self.x + 1
    print("So far",self.x)

an = PartyAnimal()
print ("Type", type(an))
print ("Dir ", dir(an))
print ("Type", type(an.x))
print ("Type", type(an.party))
```

Object Life-cycle

As our objects become more complex, we need to take some action within the object to set things up as the object is being constructed and possibly clean things up as the object is being discarded.

```
class PartyAnimal:
  x = 0
  def __init__(self):
    print('I am constructed')

  def party(self) :
    self.x = self.x + 1
    print('So far',self.x)

  def __del__(self):
    print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)
```

Many Instances

When we are making multiple objects from our class, we might want to set up different initial values for each of the objects.

```
class PartyAnimal:
  x = 0
  name = ''
  def __init__(self, nam):
    self.name = nam
    print(self.name,'constructed')

  def party(self) :
    self.x = self.x + 1
    print(self.name,'party count',self.x)

s = PartyAnimal('Sally')
s.party()
j = PartyAnimal('Jim')
j.party()
s.party()
```

Inheritance

Ability to create a new class by extending an existing class. When extending a class, we call the original class the 'parent class' (object) and the new class as the 'child class' (PartyAnimal).

```
class PartyAnimal(object):
  x = 0
  name = ''
  def __init__(self, nam):
    self.name = nam
    print(self.name,'constructed')

  def party(self) :
    self.x = self.x + 1
    print(self.name,'party count',self.x)
```

Child Class

We can 'import' the PartyAnimal class in a new file and extend it as follows:

```
from party import PartyAnimal

class CricketFan(PartyAnimal):
  points = 0

  def six(self):
    self.points = self.points + 6
    self.party()
    print(self.name,"points",self.points)

s = PartyAnimal("Sally")
s.party()
j = CricketFan("Jim")
j.party()
j.six()
print(dir(j))
```

Inheritance: Try

```
class Vehicle:
    def __init__(self, name, color):
        self.__name = name      # __name is
     private to Vehicle class
        self.__color = color
    def getColor(self):         # getColor()
     function is accessible to class Car
        return self.__color
    def setColor(self, color):  # setColor is
     accessible outside the class
        self.__color = color
    def getName(self):          # getName() is
     accessible outside the class
        return self.__name
```

Inheritance: Try

```
class Car(Vehicle):
    def __init__(self, name, color, model):
        # call parent constructor to set name and
     color
        super().__init__(name, color)
        self.__model = model
    def getDescription(self):
        return self.getName() + self.__model + "
     in " + self.getColor() + " color"

# in method getDescrition we are able to call
     getName(), getColor() because they are
```

```
# accessible to child class through inheritance
c = Car("Ford Mustang", "red", "GT350")
print(c.getDescription())
print(c.getName()) # car has no method getName()
     but it is accessible through class Vehicle
```

Multiple Inheritance

A class can inherit from more than one class.

```
class MySuperClass1():

    def method_super1(self):
        print("method_super1 method called")

class MySuperClass2():

    def method_super2(self):
        print("method_super2 method called")

class ChildClass(MySuperClass1, MySuperClass2):

    def child_method(self):
        print("child method")

c = ChildClass()
c.method_super1()
c.method_super2()
```

Overriding methods

To override a method in the base class, sub class needs to define a method of same signature. (i.e same method name and same number of parameters as method in base class).

```
class A():
    def __init__(self):
        self.__x = 1
    def m1(self):
        print("m1 from A")

class B(A):
    def __init__(self):
        self.__y = 1
    def m1(self):
        print("m1 from B")

c = B()
c.m1() # m1 from B
```

Try commenting m1() method in B class and now m1() method from Base class i.e class A will run.

isinstance() function

Used to determine whether the object is an instance of the class or not.

```
>>> isinstance(1, int)
True

>>> isinstance(1.2, int)
False

>>> isinstance([1,2,3,4], list)
True
```

Inspection

- Learn what the type of an object is – Example: `type(obj)`
- Learn what attributes an object has and what it's capabilities are – Example: `dir(obj)`
- Get help on a class or an object – Example: `help(obj)`
- In Ipython: `In [49]: a.upper?`

Exercise

- Define a class which has at least two methods:
- getString: to get a string from console input
- printString: to print the string in upper case.
- Also please include simple test function to test the class methods.
- Hints: Use __init__ method to construct some parameters

Solution

```
class InputOutString(object):
    def __init__(self):
        self.s = ""

    def getString(self):
        self.s = input()

    def printString(self):
        print self.s.upper()

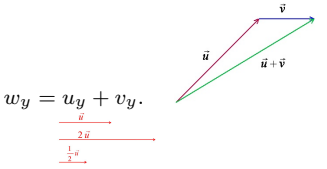
strObj = InputOutString()
strObj.getString()
strObj.printString()
```

Object Oriented Programming: Examples

Recall: *What is a 2D vector?*

- A *2D* vector is an element of the vector space $\mathbb{R}^2$.
- Every *2D* vector $\mathbf{u}$ is completely described by a pair of real coordinates $\langle u_x, u_y \rangle$.
- Note: It is not a point in space. It gives Direction, like a movement recipe.
- When added to a point, results into a transformed point.

Two operations are defined on vectors:
vector addition: if $\mathbf{w} = \mathbf{u} + \mathbf{v}$, then $w_x = u_x + v_x$ and $w_y = u_y + v_y$.



scalar multiplication: if $\mathbf{v} = \alpha \cdot \mathbf{u}$ with $\alpha \in \mathbb{R}$, then $v_x = \alpha \cdot u_x$ and $v_y = \alpha \cdot u_y$.

A *2D vector* in Python

```
class Vector(object):

  def __init__(self, x, y):
    self.x = x
    self.y = y

  def add(self, other):
    return Vector(self.x+other.x,
                  self.y+other.y)

  def mul(self, scalar):
    return Vector(scalar*self.x, scalar*self.y)

  def show(self):
    return (''<{},{}>''.format(self.x, self.y))
```

DocString

Classes can have docstrings. The content of a class docstring will be shown as help text for that class.

```
class Vector(object):
  """A 2D Vector."""
  def __init__(self, x, y):
    self.x = x
```

```
    self.y = y
  def add(self, other):
    return Vector(self.x+other.x,
                  self.y+other.y)
  def mul(self, scalar):
    return Vector(scalar*self.x, scalar*self.y)
  def show(self):
    return (''<{},{}>''.format(self.x, self.y))
```

Member functions

```
class Vector(object):
  """A 2D Vector."""
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def add(self, other):
    return Vector(self.x+other.x,
                  self.y+other.y)
  def mul(self, scalar):
    return Vector(scalar*self.x, scalar*self.y)
  def show(self):
    return (''<{},{}>''.format(self.x, self.y))
```

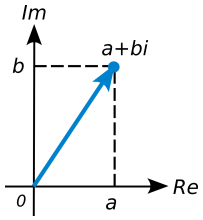
The **def** keyword introduces a method definition.
Every method *must* have at least one argument, named **self**.

Exercises

- Add a new method `norm` the `Vector` class: if `v` is an instance of class `Vector`, then calling `v.norm()` returns the norm $\sqrt{v_x^2 + v_y^2}$ of the associated vector.
- Add a new method `unit` to the `Vector` class: if `v` is an instance of class `Vector`, then calling `v.unit()` returns the vector `u` having the same direction as `v` but norm 1.

Recall: *What is a complex number?*

A complex number $\mathbf{z}$ has the form $\mathbf{z} = z_1 + z_2 i$, where z_1 and z_2 are real numbers; z_1 is called the real part and z_2 the imaginary part.



The set of complex numbers is a field with the two operations:

- *addition:* if $\mathbf{z} = \mathbf{x} + \mathbf{y}$, where $\mathbf{x} = x_1 + x_2 i$ and $\mathbf{y} = y_1 + y_2 i$ then: $z_1 = x_1 + y_1$, and $z_2 = x_2 + y_2$.
- *multiplication:* if $\mathbf{z} = \mathbf{x} \cdot \mathbf{y}$, then: $z_1 = x_1 \cdot y_1 - x_2 \cdot y_2$, and $z_2 = x_2 \cdot y_1 + x_1 \cdot y_2$.

Naive complex numbers in Python

Python object that implements a complex number.

```
class ComplexNum(object):
  "A complex number z = z1 + z2*i."
  def __init__(self, z1, z2):
    self.re = z1
    self.im = z2
  def __add__(self, other):
    return ComplexNum(
      self.re+other.re, self.im+other.im)
  def __mul__(self, other):
    return ComplexNum(
      self.re*other.re - self.im*other.im,
      self.re*other.im + self.im*other.re)
  def __str__(self):
    return ("%g + %gi" % (self.re, self.im))
  def __eq__(self, other):
    return (self.re == other.re) \
           and (self.im == other.im)
```

What does `ComplexNum` do?

We can create complex numbers by initializing them with the real and imaginary part:

```
>>> u = ComplexNum(1,0)
>>> i = ComplexNum(0,1)
>>> z = ComplexNum(1,1)
```

The str method provides the familiar representation:

```
>>> print(u)
1 + 0i
>>> print(i)
0 + 1i
>>> print(z)
1 + 1i
```

The add method makes the "+" operator work:

```
>>> w = u + i
>>> print(w)
1 + 1i
```

The mul method makes the "*" operator work:

```
>>> m = i*i
>>> print(m)
-1 + 0i
```

Exercises

- Add a to_power method to class `ComplexNum`, that takes an integer number as argument and returns the complex number raised to that power.
- Example:

```
>>> z = Complex(1,1)
>>> w = z.to_power(3)
>>> print(w)
-2 + i2
```

- Verify that z.to_power(1), z.to_power(2) and z.to_power(3) yield the same result as z, z*z and z*z*z.

Exercises

Modify the mul method of class `ComplexNum`:

- If second argument `other` is an object of class `int` (integer) or `float` (floating-point number) then return the result of scalar multiplication by that number.
- If second argument `other` is an object of class `ComplexNum`, then return the result of complex multiplication by that number.

`Vector` vs `ComplexNum`

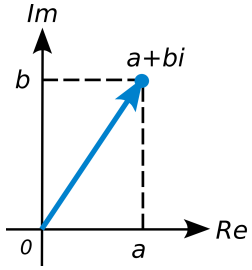
There's a lot of similarities in the two classes!
How can we share code between the two?

```
class Vector(object):
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def __eq__(self, other):
    return (self.x == other.x) \
           and (self.y == other.y)
  def __add__(self, other):
    return Vector(
      self.x+other.x, self.y+other.y)
  def __mul__(self, scalar):
    return Vector(
      scalar*self.x,
      scalar*self.y)
  def __str__(self):
    return ("<%g,%g>" % (self.x, self.y))
```

```
class ComplexNum(object):
  def __init__(self, z1, z2):
     self.re = z1
     self.im = z2
  def __eq__(self, other):
      return (self.re == other.re) \
             and (self.im == other.im)
  def __add__(self, other):
      return ComplexNum(
      self.re+other.re, self.im+other.im)
  def __mul__(self, other):
    return ComplexNum(
      self.re*other.re - self.im*other.im,
      self.re*other.im + self.im*other.re)
  def __str__(self):
     return ("%g + %gi" % (self.re, self.im))
```

Vectors and *Complex Numbers*

Complex Numbers are in $1-1$ correspondence with points in the real plane $\mathbb{R}^2$: the set $\mathbb{C}$ of Complex Numbers is the set of real *2D* vectors.



- So we can define the class of Complex Numbers as the class of Vectors augmented with a multiplication operation.
- So: a Complex Number is a Vector plus some behavior.

Inheritance, I

```
class Vector(object):

  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __eq__(self, other):
    return (self.x == other.x) \
           and (self.y == other.y)

  def __add__(self, other):
    return self.__class__(
      self.x+other.x, self.y+other.y)

  def __mul__(self, scalar):
    return Vector(
      scalar*self.x,
      scalar*self.y)

  def __str__(self):
    return ("<%g,%g>" % (self.x, self.y))
```

```
class ComplexNum(Vector):

  def __mul__(self, other):
    return ComplexNum(
      self.x*other.x - self.y*other.y,
      self.x*other.y + self.y*other.x)

  def __str__(self):
    return ("%g + %gi" % (self.x, self.y))
```

- `class ComplexNum` is a *child/descendant/subclass* of class `Vector`.
- `class Vector` is a *parent/ancestor/superclass* of class `ComplexNum`.

Inheritance, II

```
class Vector(object):

  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __eq__}(self, other):
    return (self.x == other.x) \
           and (self.y == other.y)

  def __add__(self, other):
    return self.__class__(
      self.x+other.x, self.y+other.y)

  def __mul__(self, scalar):
    return Vector(
      scalar*self.x,
      scalar*self.y)

  def __str__(self):
    return ("<%g,%g>" % (self.x, self.y))
```

```
class ComplexNum(Vector):

  def __mul__(self, other):
    return ComplexNum(
      self.x*other.x - self.y*other.y,
      self.x*other.y + self.y*other.x)

  def __str__(self):
    return ("%g + %gi" % (self.x, self.y))
```

All methods defined in class `Vector` are automatically defined *(inherited)* in class `ComplexNum`.

Inheritance, III

```
class Vector(object):

  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __eq__(self, other):
    return (self.x == other.x) \
           and (self.y == other.y)

  def __add__(self, other):
    return self.__class__(
      self.x+other.x, self.y+other.y)

  def __mul__(self, other):
    return Vector(
      scalar*self.x,
      scalar*self.y)

  def __str__}(self):
    return ("<%g,%g>" % (self.x, self.y))
```

```
class ComplexNum(Vector):

  def __mul__(self, other):
    return ComplexNum(
      self.x*other.x - self.y*other.y,
      self.x*other.y + self.y*other.x)

  def __str__(self):
    return ("%g + %gi" % (self.x, self.y))
```

Methods mul and str are defined in both classes: instances of a class use the definition from that class.
We say that `ComplexNum` *overrides* those methods from `Vector`.

Method chaining

Actually, init is not special in this regard.
When any instance method is called, Python only calls the first constructor it can find in the class inheritance call-chain.

You can always call a superclass method by prefixing it with the superclass name and explicitly writing "`self`" as the first argument:

```
class ComplexNum(Vector):
  # ...
  def print_as_vector(self):
    return Vector.__str__(self)
```

Inheritance, IV

Take a look at this Python interaction:

```
>>> z = ComplexNum(1,0)
>>> print(z)
1 + i0
>>> w = ComplexNum(0,1)
>>> print(w)
0 + i1
>>> u = z + w
>>> print(u)
```

This is no `ComplexNum`! What's happening here?

Inheritance, V

The answer is in the code:

```
class Vector(object):
  # ...
  def __add__(self, other):
    return Vector(self.x+other.x, self.y+other.y)
```

The add method returns a new Vector instance, even when called from a ComplexNum !

Inheritance, VI

Correct code:

```
class Vector(object):
  # ...
  def __add__(self, other):
    return self.__class__(self.x+other.x,
     self.y+other.y)
```

Use the class of the actual instance that's passed, instead of hard-coding a class name.

Polymorphism, I

The multiplication operator "*" on instances of the Vector class works as *scalar multiplication*:

```
>>> v = Vector(1,0)
>>> print (v * 3)
<3,0>
```

The same operator "*" on instances of the ComplexNum class works as *complex multiplication*:

```
>>> z = ComplexNum(1,1)
>>> w = ComplexNum(2,0)
>>> print (z * w)
2 + i2
```

The ability to implement different behavior for the same method/operator in different classes is called *polymorphism*.

Polymorphism, II

You may observe that an integer is (in particular) a complex number, still we cannot multiply a `ComplexNum` instance by an integer number:

```
>>> print (z * 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_and_complexnum.py", line 20, in
    __mul__
    self.x*other.x - self.y*other.y,
AttributeError: 'int' object has no attribute 'x'
```

Polymorphism, III

```
>>> print (z * 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_and_complexnum.py", line 20, in
    __mul__
    self.x*other.x - self.y*other.y,
AttributeError: 'int' object has no attribute 'x'
```

Our code for mul implicitly assumes that argument other has attributes `x` and `y`, and integers do not!

Recall: Object Oriented Programming

- A Python object is a bundle of variables and functions.
- Defined by the object's *class*.
- From class, many objects can be *instantiated*.
- Different instances can assign different values to the object variables.

Module 7: IO

Input/Output

Asking the user for input

- Python provides a built-in function called input that gets input from the keyboard:
- When the user presses Return or Enter, the program resumes and input returns what the user typed as a string.

```
>>> input = input()
Some silly stuff
>>> print(input)
Some silly stuff
```

- Before getting input from the user, it is a good idea to print a prompt telling the user what to input:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an
     unladen swallow?
What do you mean, an African or a European
     swallow?
>>> int(speed)
ValueError: invalid literal for int() with
     base 10:
```

Prompt

- Combine prompt and input

```
>>> prompt = 'What...is the airspeed
     velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an
     unladen swallow?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

- But if the user types something other than a string of digits, you get an error:

```
>>> name = input('What is your name?\n')
What is your name?
Chuck
>>> print(name)
Chuck
```

File IO

- Can read data from a file, or write into a file
- The open function takes two arguments, the name of the file, and the mode. The modes are:

```
    'r': open a file for reading
    'w': open a file for writing. Caution:
     this will overwrite any previously
     existing file
    'a': append. Write to the end of a file.
```

- The function returns a file object that performs the various tasks you'll be performing: a_file = open(filename, mode).

```
    file.read(): read the entire contents of
     a file into a string
    file.write(some_string): writes to the
     file, note this doesn't automatically
     include any new lines.
    file.flush(): write out any buffered
     writes
    file.close(): close the open file.
```

Writing a file to disk

```
# Create the file temp.txt, and get it ready for
    writing
f = open("temp.txt", "w")
f.write("This is my first file! The end!\n")
f.write("Oh wait, I wanted to say something else.")
f.close()

# Create a file numbers.txt and write the numbers
    from 0 to 24 there
f = open("numbers.txt", "w")
for num in range(25):
    f.write(str(num)+'\n')
f.close()
```

Reading a file from disk

```
# We now open the file for reading
f2 = open("temp.txt", "r")
# And we read the full content of the file in
    memory, as a big string
f2_content = f2.read()
f2.close()

# Read the file in the cell above, the content is
    in f2_content

# Split the content of the file using the newline
    character \n
lines = f2_content.split("\n")

# Iterate through the line variable (it is a list
    of strings)
# and then print the length of each line
for line in lines:
    print("Length of line `{0}` is
    {1}".format(line,len(line)))
```

Reading a file from disk

```
# We now open the file for reading
f2 = open("numbers.txt", "r")
# And we read the full content of the file in
    memory, as a big string
f2_content = f2.read()
f2.close()

lines = f2_content.split("\n")
print(lines)

numbers = [int(line) for line in lines if
    len(line)>0]
print(numbers)
```

Using try, except, and open

```
fname = input('Enter the file name: ')
try:
  fhand = open(fname)
except:
  print('File cannot be opened:', fname)
  exit()
count = 0
for line in fhand:
  if line.startswith('Subject:'):
    count = count + 1
print('There were', count, 'subject lines in',
    fname)
```

Searching through a file

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
  line = line.rstrip()
  if line.startswith('From:'):
    print(line)
```

Filesystem operations, I

`os` module functions:

```
os.getcwd(),   os.chdir(path)
```

Get current working directory. Change to `path`.

```
os.listdir(dir)
```

Return list of entries in directory `dir` (omitting '.' and '..')

```
os.mkdir(path)
```

Create a directory; fails if the directory already exists. Assumes that all parent directories exist already.

```
os.makedirs(path)
```

Create a directory; no-op if the directory already exists. Creates all the intermediate-level directories needed to contain the leaf.

Filesystem operations, II

These functions are available from the `os.path` module.

```
os.path.exists(path), os.path.isdir(path),
    os.path.isfile(path)
```

Return `True` if `path` exists / is a directory / is a regular file.

```
os.path.basename(path),
os.path.dirname(path)
```

Return the base name (the part after the last '/' character) or the directory name (the part before the last / character).

```
os.path.abspath(path)
```

Make `path` absolute (i.e., start with a /).

Exercises

- Write a program that reads the https://raw.github.com/gc3-uzh-ch/python-course/master/euro.csv file and populates a dictionary from it: currency names (first column) are the dictionary keys, conversion rates (second column) are the dictionary values.

Exercises

- Write a function that reads the n-th column of a CSV file and returns its contents. (Reuse the function that you wrote above.) Then reads the file data/baseball.csv and return the content of the 5th column (team).
- The command below will create a file called phonetest.txt. Write code that: Reads the file phonetest.txt; Write a function that takes as input a string, and removes any non-digit characters; Print out the "clean" string, without any non-digit characters;

Exercises

- Write a function `wordcount(filename)` that reads a text file and returns a dictionary, mapping words into occurrences (disregarding case) of that word in the text. Test it with the https://raw.github.com/gc3-uzh-ch/python-course/master/lorem_ipsum.txt file:

```
>>> wordcount('lorem_ipsum.txt')
{'and': 3, 'model': 1, 'more-or-less': 1,
 'letters': 1, ...
```

Files Operations (recap)

- input `input = open('data', 'r')`
- read all `S = input.read()`
- read N bytes `S = input.read(N)`
- read next `S = input.readline()`
- read in lists `L = input.readlines()`
- output `output = open('/tmp/spam', 'w')`
- write `output.write(S)`
- write strings `output.writelines(L)`
- close `output.close()`

Module 8: Libraries I

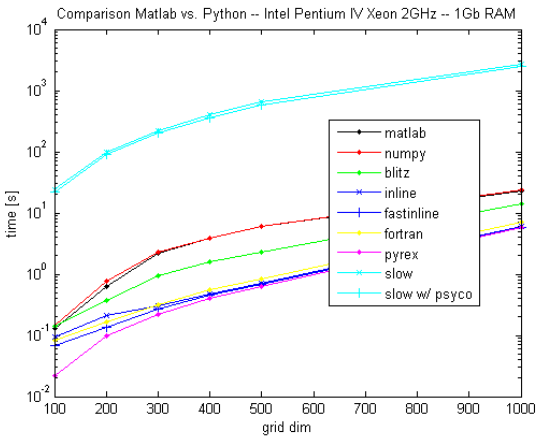
Packages: Scientific

NumPy: linear algebra package

- NumPy is a package for linear algebra and advanced mathematics in Python.
- It provides a *fast* implementation of multidimensional numerical arrays (C/FORTRAN like), vectors, matrices, tensors and operations on them.
- *Use it if:* you long for MATLAB core features.
- Reference: http://www.numpy.org/
- Examples: http://wiki.scipy.org/Numpy_Example_List

NumPy vs. Matlab vs. Bytecode

Comparison of $\nabla^2 u = 0$ solvers (500×500, 100 iterations):



NumPy Array Generation

Convert a list to a NumPy array. NumPy arrays have values and a data type (*dtype*).

```
>>> import numpy as np
>>> list1 = [1, 2, 3, 4, 5]
>>> x = np.array(list1)
>>> x
array([1, 2, 3, 4, 5])
>>> x.dtype
dtype('int64')
```

NumPy Array Generation

`arange()` is similar to `range()` for lists:

```
>>> np.arange(5)
array([0, 1, 2, 3, 4])
>>> np.arange(5.)
array([0., 1., 2., 3., 4.])
```

`zeros` for pre-allocation:

```
>>> np.zeros([2, 3])    # Input is a list
array([[ 0., 0., 0.],
       [ 0., 0., 0.]])
```

NumPy Grid Generation

Suppose you want to construct a numerical grid. e.g. $x = x_0 + j\Delta x$, $\Delta x = (x_1 - x_0)/N$ `arange()` uses Δx and excludes x_1:

```
>>> x = np.arange(3., 6., 0.5)
array([ 3. ,  3.5,  4. ,  4.5,  5. ,  5.5])
```

`linspace()` uses $N + 1$ and includes x_1:

```
>>> x = np.linspace(0., 1., 6)
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
```

NumPy Arithmetic

All NumPy operations are vectorized

```
>>> x = np.linspace(0., 4., 5)
>>> y = np.linspace(-2., -2., 5)
>>> 2*x
array([ 0.,  2.,  4.,  6.,  8.])
>>> x+y
array([-2.,  0.,  2.,  4.,  6.])

% Most mathematical functions are supported:

>>> np.exp(x)
>>> np.arctan(x)
>>> np.pi
```

Inefficient Arithmetic

Never do this:

```
>>> for i in range(10):
        z[i] = x[i] + y[i]

    % It will send 10 separate jobs to the C
     libraries.
    \\~\\
    % Always try to do vectorized calculations:
>>> z = x+y

  %     It only sends one job.
```

Arithmetic Quiz

Use `np.mean()` to estimate the mean value of $f(x)$ over $[-1, 1]$:

$$\frac{1}{2}\int_{-1}^{1} f(x)\,dx$$

- $f(x) = \sin x$
- $f(x) = 1/(1 + x^2)$
- $f(x) = x^2 \exp(-x^2)$

```
import numpy as np

N = 1001    # (N-1) intervals
x = np.linspace(-1., 1., N)

print(np.mean(np.sin(x)))
print(np.mean(1/(1 + x**2)))
print(np.mean(x**2 * np.exp(-x**2)))
```

Arithmetic Quiz

Note: the formulation stated earlier is an approximation to original approximation suggested in Trapezoidal rule

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \frac{\Delta x}{2}\sum_{k=1}^{N}\left(f(x_{k-1}) + f(x_k)\right)\\ &= \frac{\Delta x}{2}\left(f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{N-1}) + f(x_N\right)\\ &= \frac{\Delta x}{2}\left(f(x_0) + 2\sum_{k=1}^{N-1} f(x_k) + f(x_N)\right)\end{aligned}$$

(Ref: https://en.wikipedia.org/wiki/Trapezoidal_rule)

Multidimensional Arrays

```
%     NumPy supports multidimensional arrays:

>>> x = np.zeros((3,4))
>>> x
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> x[:,0]
array([ 0.,  0.,  0.])
>>> x[0] # or x[0,:]
array([ 0.,  0.,  0., 0.])

%    NumPy arrays support comma-separated
     dimension indexing
```

Reshaping

Two ways to reshape an array: `reshape()` outputs a new reshaped array:

```
>>> x = np.arange(12)
>>> x.reshape(3,4)
```

Broadcasting

```
%    Arithmetic usually requires arrays to be the
     same shape:

>>> x = np.arange(12).reshape(3,4)
>>> y = np.arange(12).reshape(4,3)
>>> x*y   # Does this work?

  %    But broadcasting will copy outer dimensions
     inward:

>>> x = np.ones(12).reshape(3,4)
>>> y = np.arange(4)
>>> x*y      # Outer dimension matches
%    As long as the last dimensions match, you can
     broadcast.
```

Extending Array Dimensions

What if you need to multiply along the first dimension?

```
>>> x = np.ones((3,4))
>>> y = np.arange(3)
>>> x * y                    # Won't work
>>> x * y[:, np.newaxis]     # Works!
```

`np.newaxis` extends any missing dimension! (Also see `np.tile` and `np.repeat`)

Combining Arrays

Combine two arrays along some dimension:

```
>>> x = np.arange(5)
>>> y = np.arange(5)
>>> np.hstack((x,y))    # Stack on axis=0
>>> np.vstack((x,y))    # Stack on axis=1

# Use \lstinline|np.concatenate| for higher
    dimensions:


>>> x = np.ones((4,3,2))
>>> y = np.ones((4,3,1))
>>> np.concatenate((x,y),axis=2)
```

NumPy Variables are References

NumPy arrays are mutable, so NumPy variables are *references*. Try these commands, then look at `x` and `y`:

```
>>> x = np.arange(5)
>>> y = x
>>> x[0] = 5

#     Changing x will change y
```

Copying NumPy Arrays

Deep Copy: Duplicate the array in memory

```
>>> x = np.arange(10)
>>> y = np.copy(x)
>>> x[0] = 5
>>> x
>>> y
```

Masking

Filtering with logical operators:

```
>>> x = np.random.rand(3,4)
>>> x > 0.5
>>> x[(x>0.5)]

#     This can be useful, but you lose the shape!
```

Masked Arrays

NumPy provides Masked Array support: `np.ma`:

```
>>> x = np.random.rand(3,4)
>>> x_m = np.ma.masked_array(x, x>0.5)
>>> print x_m

#     Support isn't universal, but it's not bad
```

SciPy: a toolbox for numerics

- SciPy is open-source software for mathematics, science, and engineering. *[...]* The SciPy library provides many user-friendly and efficient numerical routines such as routines for numerical integration and optimization.
- One of its main aim is to provide a reimplementation of the MATLAB toolboxes.
- *Use it if:* you long for MATLAB toolbox features.
- Tutorial: http://docs.scipy.org/doc/scipy/reference/tutorial/index.html
- Examples: http://nbviewer.ipython.org/github/jrjohansson/scientific-python-lectures/blob/master/Lecture-3-Scipy.ipynb

SciPy

SciPy provides lots of useful science tools:

- `scipy.interpolate` Grid interpolation tools
- `scipy.stats`, `scipy.random` Statistical analysis
- `scipy.signal` Filtering, Signal Processing

and many more

I/O with SciPy

NumPy has a unique binary data format:

```
>>> np.save('mydata.npy', data)
>>> data = np.load('mydata.npy')

# Several I/O routines are provided by SciPy
    (scientific python).
```

```
#   Matlab:
>>> import scipy.io as sio
>>> data = sio.loadmat('mydata.mat')
>>> sio.savemat('mydata.mat', {'var': data})
```

Plotting

matplotlib: publication quality plotting library

- matplotlib is a python 2D plotting library for quality figures
- matplotlib can be used in python scripts, the python and ipython shell (ala MATLAB or Mathematica), web application servers, and six graphical user interface toolkits
- Tutorial: http://www.loria.fr/~rougier/teaching/matplotlib/
- Examples: http://matplotlib.org/1.2.1/gallery.html

What is Matplotlib?

- Matplotlib is a rendering API for 2D (and limited 3D) plotting
- `matplotlib.pyplot` is a streamlined Matlab-like interface to Matplotlib
- `pylab` is a bundle of NumPy, SciPy and Matplotlib. One often sees it invoked like this:

```
from pylab import *
```

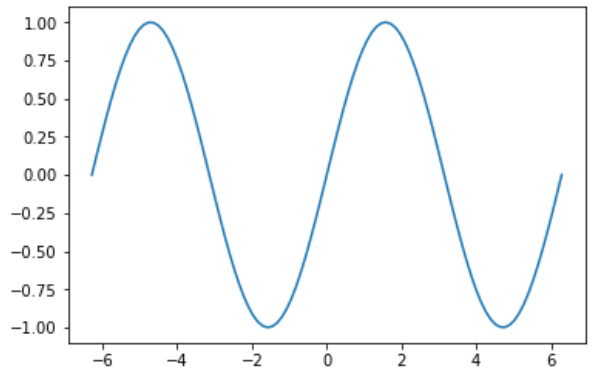
My First Plot

Plot $\sin x$ from -2π to 2π:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-2.*np.pi, 2.*np.pi, 100)
y = np.sin(x)

plt.plot(x, y)
plt.show()
```



Saving your Figure

Append this to the end of your script:

```
plt.savefig('myplot.pdf')
```

It usually figures out the file type from the extension. To remove space around the plot, use:

```
plt.savefig('myplot.pdf', bbox_inches='tight')
```

Multiple Plots

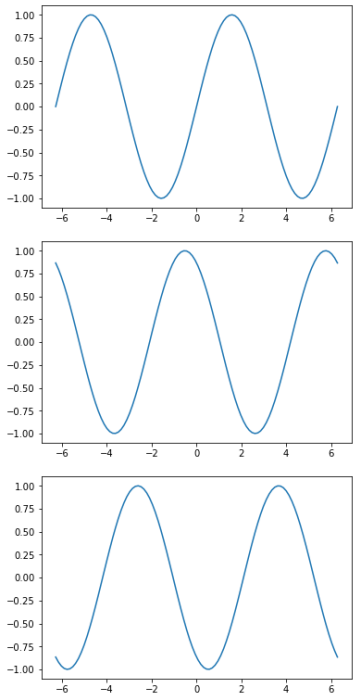
Plot three sine curves with different phase shifts: $y_n = \sin(x + \phi_n)$ Call `plot()` three times, then `show()` the results

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-2.*np.pi, 2.*np.pi, 100)
phase = np.arange(0., 2*np.pi, 2*np.pi/3.)

y_n = [np.sin(x + p) for p in phase]

for y in y_n:
    plt.plot(x, y)
plt.show()
```



Dots and Dashes

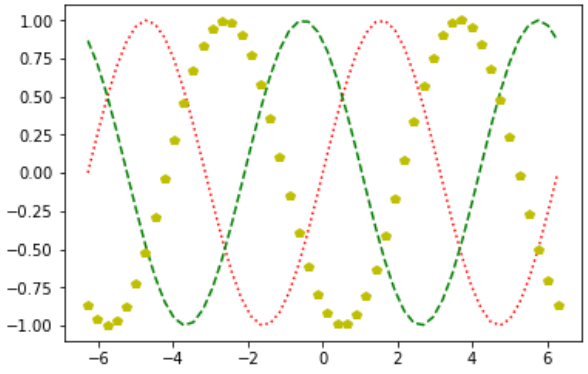
Stylise curves with dashes, shapes, and colours (like Matlab):

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-2.*np.pi, 2.*np.pi, 50)
phase = np.arange(0., 2*np.pi, 2*np.pi/3.)

y_p = [np.sin(x + p) for p in phase]
style = ['r:', 'g--', 'py']

pdata = zip(y_p, style)
for y,s in pdata:
    plt.plot(x, y, s)
plt.show()
```



Axis Labels

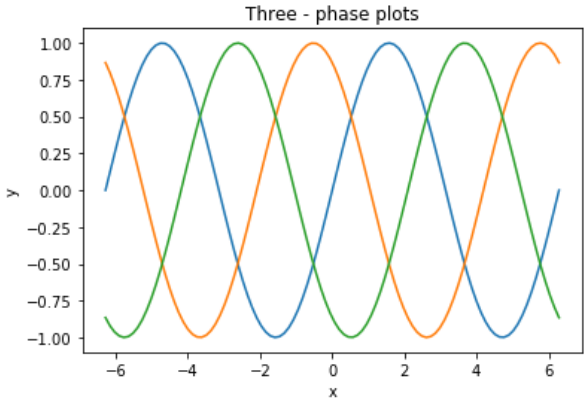
Labeling axes is similar to Matlab:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-2.*np.pi, 2.*np.pi, 100)
phase = np.arange(0., 2*np.pi, 2*np.pi/3.)
y_n = [np.sin(x + p) for p in phase]

for y in y_n:
    plt.plot(x, y)

plt.title('Three-phase plots')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Legends

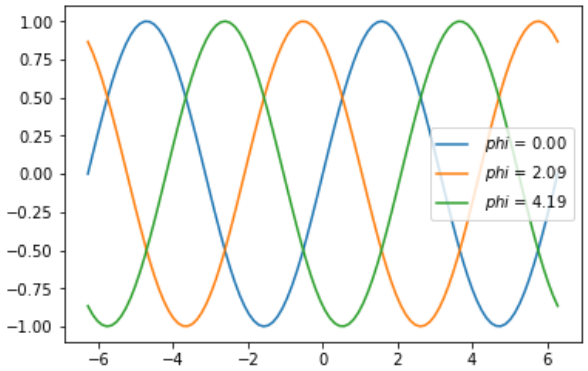
Include a legend in your plot

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-2.*np.pi, 2.*np.pi, 100)
phase = np.arange(0., 2*np.pi, 2*np.pi/3.)
y_n = [np.sin(x + p) for p in phase]

for y in y_n:
    plt.plot(x, y)

legend_text = ['$\phi$ = %.2f' % p
                                for p in phase]
plt.legend(legend_text)
plt.show()
```



Contour Plots

Plot contours on range $[-2, 2] \times [-2, 2]$ for the function:

$$z(x, y) = \left(x - \frac{1}{2}\right) e^{-\sqrt{(x^2+y^2)}}$$

```
import numpy as np
import matplotlib.pyplot as plt
x_ax = np.linspace(-2., 2., 100)
y_ax = np.linspace(-2., 2., 100)
x, y = np.meshgrid(x_ax, y_ax)
z = (x - 0.5)*np.exp(-np.sqrt(x**2 + y**2))
plt.contour(x, y, z, 25)
plt.show()
```

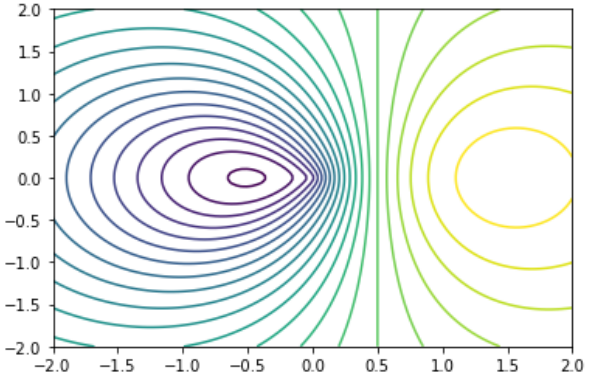
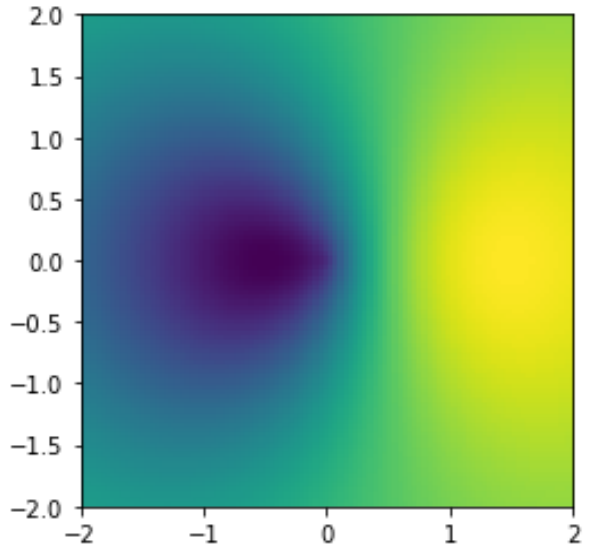


Image Plots

`imshow` plots pixel fields (`pcolor` is very slow)

```
import numpy as np
import matplotlib.pyplot as plt
x_ax = np.linspace(-2., 2., 100)
y_ax = np.linspace(-2., 2., 100)
x, y = np.meshgrid(x_ax, y_ax)
z = (x-0.5) * np.exp(-np.sqrt(x**2 + y**2))
z_ext = (x_ax[0], x_ax[-1], y_ax[0], y_ax[-1])
plt.imshow(z, origin='lower', extent=z_ext)
plt.show()
```



Object-Oriented Matplotlib

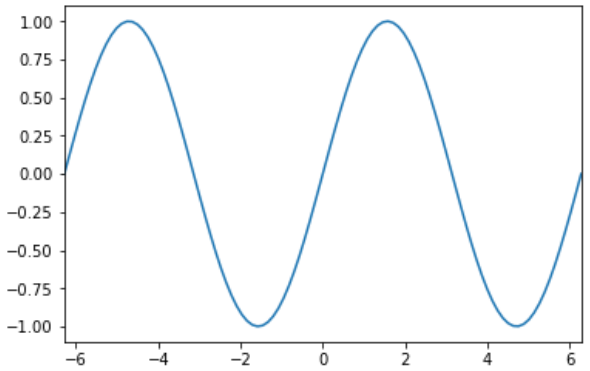
As your plots become more complex, you may need to start using object-oriented matplotlib

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-2.*np.pi, 2.*np.pi, 100)
y = np.sin(x)

fig = plt.figure()
ax = fig.add_subplot(1,1,1)
line = ax.plot(x, y)
ax.set_xlim([-2*np.pi,2*np.pi])

plt.show()
```



Subplots

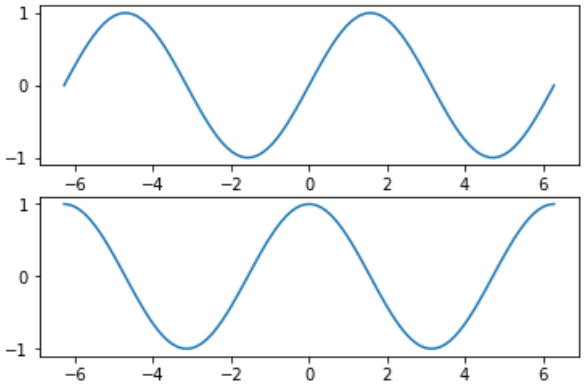
Put two plots on the same figure:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-2.*np.pi, 2.*np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

fig, (ax1, ax2) = plt.subplots(nrows=2,
                               ncols=1)
ax1.plot(x, y1)
ax2.plot(x, y2)

plt.show()
```



Basemap: Earth Grid Plotting

Basemap provides tools for plotting on several geographic grids.

Orthographic Grids

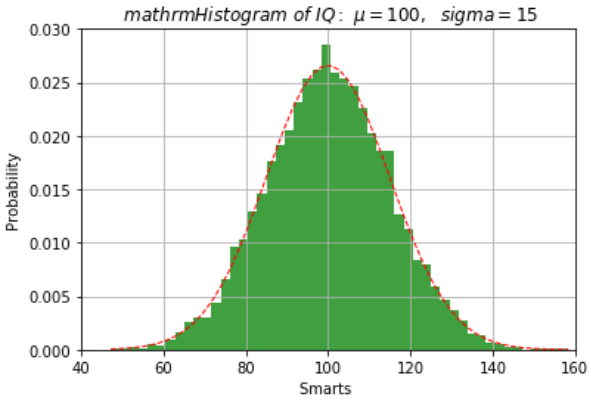
```
from mpl_toolkits.basemap import Basemap
import numpy as np
import matplotlib.pyplot as plt

m = Basemap(projection='ortho',lon_0=-105,
            lat_0=40,resolution='l')
m.drawcoastlines()
m.fillcontinents(color='coral',
                 lake_color='aqua')
m.drawparallels(np.arange(-90.,120.,30.))
m.drawmeridians(np.arange(0.,420.,60.))
m.drawmapboundary(fill_color='aqua')
plt.show()
```

Mercator Grids

```
from mpl_toolkits.basemap import Basemap
import numpy as np
import matplotlib.pyplot as plt
m = Basemap(projection='merc',llcrnrlat=-80,
            urcrnrlat=80, llcrnrlon=-180,
            urcrnrlon=180,lat_ts=20,
            resolution='c')
m.drawcoastlines()
m.fillcontinents(color='coral',
                 lake_color='aqua')
m.drawparallels(np.arange(-90.,91.,30.))
m.drawmeridians(np.arange(-180.,181.,60.))
m.drawmapboundary(fill_color='aqua')
```

```
plt.show()
```

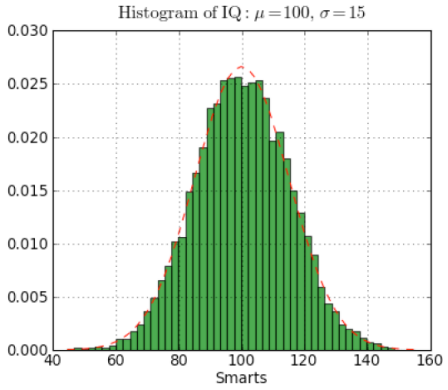


Vectors and *Complex Numbers*

```
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

mu, sigma = 100, 15
x = mu + sigma*np.random.randn(10000)
# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1,
     facecolor='green',
alpha=0.75)
# add a 'best fit' line
y = mlab.normpdf( bins, mu, sigma)
l = plt.plot(bins, y, 'r--', linewidth=1)
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title(r'$\mathrm{Histogram\ of\ IQ:}\
     \mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```

Vectors and *Complex Numbers*



Module 9: Libraries II

Data Mining

Scraping

- To analyze data, we typically need structure.For instance, same number of rows for each column.
- But found data often with human readable structure.
- Copy and paste, type, to a dataset.
- But error prone, and not scalable.
- *Idea:* Find the less accessible structure, automate based on it.

Web Scraping Workflow

Web scraping downloads and processes content from the Web

- Get the website - using HTTP library
- Parse the html document - using any parsing library
- Store the results - either a db, csv, text file, etc

Libraries:

- **Requests**: Downloads files and web pages from the Internet
- **Beautiful Soup**: Parses HTML, the format that web pages are written in.
- **Selenium**: To fill in forms and simulate mouse clicks in this browser.

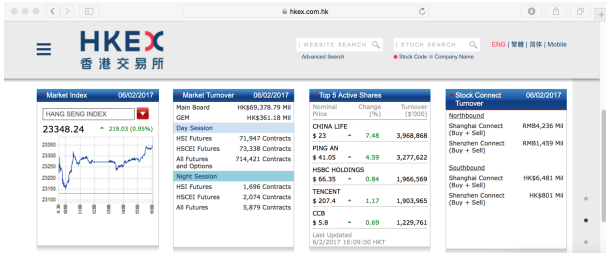
Web Scraping Issues

Web-scraping is difficult for some annoying (i.e. not particularly intellectually challenging) reasons:

- Web pages change frequently and will break your code.
- Web page source code is often not logical and consistent (major browsers are incredibly good at overlooking this, but python and your own code probably aren't).
- Content may be different for different User Agents (which client you're using).
- Content may be hidden behind logins, require cookies or other fancier browser options, etc.
- Websites may attempt to limit automated crawling of their pages (robots.txt) — your code may have to go out of its way to be nice, or risk getting banned.

Web Scraping: Get the website

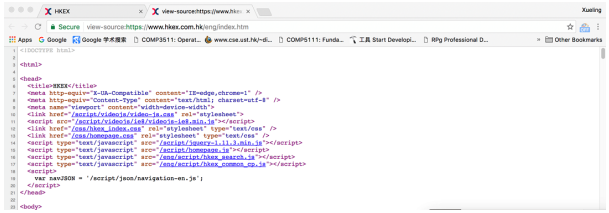
Suppose that we want to crawl this web page: https://www.hkex.com.hk/eng/index.htm



(Reference: https://studyslide.com/doc/288619/tutorial-2—hkust-cse)

Web Scraping: Get the website

In Chrome: Right-click, choose "View Page Source"



Web Scraping: Get the website

We need to collect useful information from source HTML code:

Requests

Web scraping downloads and processes content from the Web

- Do not need to worry about complicated issues such as network errors, connection problems, and data compression.
- `requests.get()` function takes a string of a URL to download.

```
import requests
res =
     requests.get('https://automatetheboringstuff.com/
```

- Response contains the response that the web server gave.

```
if res.status_code == requests.codes.ok:
  print(res.text)
```

Beautiful Soup

Web scraping downloads and processes content from the Web

- To parse a document, pass it into the BeautifulSoup constructor. You can pass in a string or an open filehandle:

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(open("index.html"))
soup = BeautifulSoup("<html>data</html>")
```

- Print the content in the soup object in better format `print(soup.prettify())`
- A Tag object corresponds to an XML or HTML tag in the original document: `<title>The Dormouse's story</title>`
- Use BeautifulSoup to get the tags in a convenient way `print(soup.title)` `print(soup.head)`

Beautiful Soup: Tag

Web scraping downloads and processes content from the Web

- Every tag has a name, accessible as .name

```
print(soup.head.name)
#head
```

- A tag may have any number of attributes. The tag <b class=''boldest''> has an attribute "class" whose value is "boldest". You can access a tag's attributes by treating the tag like a dictionary:

```
print(soup.p.attrs)
#{'class': ['title'], 'name': 'dromouse'}
print(soup.p['class'])
#['title']
print(soup.p.get('class'))
#['title']
print(type(soup.p))
#<class 'bs4.element.Tag'>
```

Beautiful Soup: Sample workflow

- To begin, we need to import Beautiful Soup and urllib, and grab source code:

```
import bs4 as bs
import urllib.request

source =
    urllib.request.urlopen('https://pythonprogramming.net/parsememcparseface/').read()
```

- Then, we create the "soup". This is a beautiful soup object:

```
soup = bs.BeautifulSoup(source,'lxml')
```

- Find more about whats in the soup object:

```
print(soup.title)
print(soup.title.name)
print(soup.title.string)

# beginning navigation:
print(soup.title.parent.name)

# getting specific values:
print(soup.p)
```

Beautiful Soup: Sample workflow

- Finding paragraph tags ¡p¿ is a fairly common task. What if we wanted to find them all?

```
print(soup.find_all('p'))
```

- We can also iterate through them:

```
for paragraph in soup.find_all('p'):
    print(paragraph.string)
    print(str(paragraph.text))
```

- The difference between string and text is that string produces a NavigableString object, and text is just typical unicode text.
- Notice that, if there are child tags in the paragraph item that we're attempting to use .string on, we will get None returned.

Beautiful Soup: Sample workflow

- Another common task is to grab links. For example:

```
for url in soup.find_all('a'):
    print(url.get('href'))
```

- In this case, if we just grabbed the .text from the tag, you'd get the anchor text, but we actually want the link itself.
- That's why we're using .get('href') to get the true URL.
- Finally, you may just want to grab text. You can use .get_text() on a Beautiful Soup object, including the full soup:

```
print(soup.get_text())
```

This concludes the introduction to Beautiful Soup.

Beautiful Soup: Sample workflow

- Another common task is to grab links. For example:

```
for url in soup.find_all('a'):
    print(url.get('href'))
```

- In this case, if we just grabbed the .text from the tag, you'd get the anchor text, but we actually want the link itself.
- That's why we're using .get('href') to get the true URL.
- Finally, you may just want to grab text. You can use .get_text() on a Beautiful Soup object, including the full soup:

```
print(soup.get_text())
```

This concludes the introduction to Beautiful Soup. Next is the Navigation.

Beautiful Soup: Sample workflow

- In this case, we're grabbing the first nav tags that we can find (the navigation bar):

```
nav = soup.nav
for url in nav.find_all('a'):
    print(url.get('href'))
```

- You could also go for soup.body to get the body section, then grab the .text from there:

```
body = soup.body
for paragraph in body.find_all('p'):
    print(paragraph.text)
```

- To avoid multiple tags with same names, use other things such as classes, to differentiate:

```
for div in soup.find_all('div',
     class_='body'):
    print(div.text)
```

Beautiful Soup: Sample workflow

- Parsing the table:

```
table = soup.find('table')
```

- Next, we can find the table rows within the table:

```
table_rows = table.find_all('tr')
```

- Then we can iterate through the rows, find the td tags, and then print out each of the table data tags:

```
for tr in table_rows:
    td = tr.find_all('td')
    row = [i.text for i in td]
    print(row)
```

Module 10: Closure

Testing

Testing is not a waste of time.

A *test* is a piece of code that checks if the code you wrote behave as expected.

Testing...

- ▷ makes sure your code works properly (under given conditions).
- ▷ ensures changes to the code do not break existing functionality.
- ▷ forces you to think about unusual conditions and corner cases.

Test Driven Development: writing tests *before* writing the code helps you to write *better code*.

Different kind of tests

There are two kind of tests:

- ▷ unit tests: check that a method or class, in isolation, actually performs the tasks that it is supposed to do
- ▷ functional tests: check that the global behavior of an application is the expected one

It's better to have both :-) but we are only going to cover only "unit tests" here.

Doctest (1/7)

- ▷ The `doctest` module searches for text that looks like interactive Python sessions in *docstrings*, and then executes them to verify that they work exactly as shown.
- ▷ Combines *documentation* and *tests*. Example:

```
def square(self, n):
  """
  This function compute the square of a
   number.

    >>> square(2)
    4
    >>> square(-2)
    4
  """
  return n*n
```

Doctest (2/7)

```
def square(x):
    """
    This function
     compute
    the square of a
     number.

      >>> square(2)
      4
    """
```

Line *not* starting with `>>>` or *not* following a line starting with `>>>` are documentation lines, and are thus ignored for testing purposes.

Doctest (2/7)

```
def square(x):
    """
    This function
     compute
    the square of a
     number.

      >>> square(2)
      4
    """
```

Line *not* starting with `>>>` or *not* following a line starting with `>>>` are documentation lines, and are thus ignored for testing purposes.

Doctest (3/7)

```
def square(x):
    """
    This function
     compute
    the square of a
     number.

      >>> square(2)
      4
    """
```

This line starts with `>>>`, so it's Python code: it will be executed inside a Python shell.

Doctest (4/7)

```
def square(x):
    """
    This function
     compute
    the square of a
     number.

      >>> square(2)
      4
    """
```

This line follows a line starting with `>>>` *and* is indented in the same way: it's the output of the previous Python statement.

Doctest (5/7)

To execute all the doctests of module `foo.py`, run:

```
$ python -m doctest foo.py
```

Doctest (6/7)

By default, running doctests only shows failed tests:

```
**********************************
File "vector6.py", line 32, in vector6.Vector
Failed example:
    print v * 2
Expected:
    <2,1>
Got:
    <2,4>
**********************************
1 items had failures:
   2 of  10 in vector6.Vector
***Test Failed*** 2 failures.
```

Otherwise: no output means that all the tests passed.

Doctest (7/7)

`-v` for more verbose output:

```
Trying:
    print v * 2
Expecting:
    <2,4>
ok
5 items had no tests:
    vector6
    vector6.Vector.__add__
    vector6.Vector.__eq__
    vector6.Vector.__init__
    vector6.Vector.__str__
1 items passed all tests:
   2 tests in vector6.Vector.__mul__
```

```
**********************************************
1 items had failures:
   2 of  10 in vector6.Vector
12 tests in 7 items.
10 passed and 2 failed.
***Test Failed*** 2 failures.
```

Exercises

The file vector6.py has some doctest, but they are wrong. Run the tests, find the errors and fix them!

The unittest module (1/4)

Allows you to create tests in a more structured way using the *Template method pattern*.

```
import unittest

class MyTest(unittest.TestCase):
  def test_square1(self):
    assert square(1) == 1

  def test_square2(self):
    self.assertEqual(square(2), 4)
    self.assertEqual(square(-2), 4)
```

Any method whose name starts with `test_` will be run. A test is successful iff it does *not* raise an exception!
Specialized "asserts" are defined in the `TestCase` class: they provide better logging and reporting of failures.

The unittest module (2/4)

The `unittest.TestCase` class defines some useful methods:

assertEqual(x,y)	check that `x == y`
assertTrue(x)	check that `x is True`
assertGreater(x, y)	check that `x > y`
assertIsInstance(obj, cls)	check that `obj` is an instance of `cls`
...	a lot more, cf. `help(unittest)`

Each one of these method is able to print detailed informations on why the test failed, this is why you don't just use `assertTrue()` for all the tests...

The unittest module (3/4)

Supports *fixtures*, code to run before and/or after each test to prepare and cleanup the testing environment.

```
import unittest

class MyTest(unittest.TestCase):
  def setUp(self):
    """This code is run *before* each test
     method."""

  def tearDown(self):
    """This code is run *after* each test
     method."""
```

The unittest module (4/4)

Run unit tests with:

```
$ python -m unittest ex10b
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

Exercises

Write a set of unit tests for the following function:

```
def is_prime(number):
  """Return True if 'number' is prime."""
  for element in range(number):
    if number % element == 0:
      return False
  return True
```

Pay special attention to the corner cases!

Assignments

Find similar company names

Quite often, we have strings that refer to the same entity but have different string representations (e.g., McDonald's vs McDonalds vs McDonald). Need to find such similar entries in our data.

- Step 1: Read the data into a list of strings. The list of unique restaurant names from the NYC Restaurant Inspection dataset (data/restaurant-names.txt) and a list of bank names (data/bank-names.txt)
- Step 2: How to compare two strings and find their similarity.
- Step 3: Write a function that takes as input a company name, and returns back a list of matching company names, together with their similarity.
- Step 4: In the final step, we will just perform the similarity computation across all companies in the dataset.

STEP 1

Read the list of names from a file and create a list of names

```
# STEP 1: Read the list of names
path = 'data/'
filename = path + 'restaurant-names.txt'
```

STEP 2

Computing the similarity between two strings. There are many ways that we can calculate the similarity between two strings.

- We will use the q-gram similarity metric.
- What is a q-gram? Simply a sequence of q-consecutive characters in the string.
- For example, the name "Panos" has the following 2-grams: "Pa", "an", "no", "os".
- Strings that share a large number of q-grams are often similar. Compute the Jaccard coefficient between these sets.

```
# STEP 2: Now we implement our similarity function

# This returns a list of q-grams for a name
# getQgrams("Panos", 1) should return ["P", "a",
    "n", "s"]
# getQgrams("Panos", 2) should return ["Pa", "an"]
def getQgrams(name, q):
    pass
```

STEP 3

```
# STEP 3: We now create a function that accepts a
    company name
# and a list of companies, and computes their
    similarity
# We have a threshold parameter (by default set to
    be 0.7)
# that restricts the results to only string pairs
    with similarity
# above the threshold

# This function takes as input two names, computes
    their qgrams, and then computes
# the Jaccard coefficient (=intersection/union) of
    the two sets of qgrams
def computeSimilarity(name1, name2, q):
    # your code here
    pass
```

```
print computeSimilarity("Peter", "Pete", 2)
```

STEP 4

Perform the similarity computation across all companies in the dataset.

```
# STEP 4: We are almost done. We now just go
    through all the companies in the list
# and we call the companySimilarity function that
    computes the similar company names
# for all the companies in the list. We store the
    results in a dictionary, with the
# key being the company name, and the value being
    a "list of tuples" with the
# similar company names and the corresponding
    similarity value.
```

What's next?

Recap

- A scripting language, Python, is suitable:
  - for embedding
  - for writing small unstructured scripts,
  - for "quick and dirty" programs.
- Not JUST ANY a scripting language:
  - Python scales.
  - Python encourages us to write code that is clear and well-structured.

Recap

- Provides an interactive command line and interpreter shell.
- Dynamic:
  - Types are bound to values, not to variables.
  - Values are inspect-able.
  - You can list the methods supported by any given object.
- Strongly typed at run-time, not compile-time.
- Objects (values) have a type, but variables do not (that's why you can assign to any value type).
- Reasonably high level – High level built-in data types;
- High level control structures (for walking lists and iterators, for example).

Recap

- Object-oriented – Almost everything is an object.
- Readability, locate-ability, modifiability.
- Indented block structure – "Python is pseudo-code that runs."
- Embedding and extending Python:
  - Embed the Python interpreter in C/C++ applications (FreeCAD)
  - Modules and objects implemented in C/C++ (SWIG)
  - Cython enables us to generate C code from Python
- Automatic garbage collection

The Zen of Python (by Tim Peters)

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Practicality beats purity.
- Errors should never pass silently, unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one– and preferably only one –obvious way to do it, although that way may not be obvious at first unless you're Dutch.
- Now is better than never, although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea – let's do more of those!

(Presented at a talk on the History of Python by Guido van Rossum in 2005 at EuroPython in Gothenburg, Sweden)

Bye

Life is Short (You Need Python) - Bruce Eckel (Thinking in C++)

Extra

Kid's Play

Finding Phone Number

- Now a days, many email/chat programs find and highlight phone numbers in a message, so that you can call the number directly by clicking it.
- Imagine you need to write that program: Finding phone numbers in a message.
- How would you go about it?
- Psuedo code?

Finding Phone Number

- Split the message with ' ' (space) to get words
- for each word, check if it is a phone number of format, xxx-xxx-xxxx (US style, for now)

```
def isPhoneNumber(text):
  if len(text) != 12:
            return False
  for i in range(0, 3):
    if not text[i].isdecimal():
      return False
  if text[3] != '-':
    return False
  for i in range(4, 7):
    if not text[i].isdecimal():
              return False
  if text[7] != '-':
    return False
  for i in range(8, 12):
    if not text[i].isdecimal():
      return False
  return True
```

Regular Expressions

- Regular expressions (regex), are descriptions for a pattern.
- `\d` stands for any digit from 0 to 9.
- `\d\d\d-\d\d\d-\d\d\d\d` represents the phone number.
- Same, but slightly shorter regex is `\d{3}-\d{3}-\d{4}`

Prints "Phone number found: 415-555-4242" Becomes as easy as Kid's play?

```
import re

message = 'Call me at 415-555-1011 tomorrow.'
phoneNumRegex =
     re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
mo = phoneNumRegex.search(message)
print('Phone number found: ' + mo.group())
```

Google Search

My Steps (most likely your's too):

- Type search term in the box of *google.com*.
- Middle-click top few search result links to open in new tabs.
- Doing this one by one is tedious.
- Any way to automate this?

Using Python

- Gets search keywords from the command line arguments.
- Retrieves the search results page.
- Opens a browser tab for each result.

Opens result links in new browser tabs. Becomes as easy as Kid's play?

```
import requests, sys, webbrowser, bs4

res = requests.get('http://google.com/search?q=' +
     \
           ' '.join(sys.argv[1:]))
soup = bs4.BeautifulSoup(res.text)
linkElems = soup.select('.r a')
numOpen = min(5, len(linkElems))
for i in range(numOpen):
    webbrowser.open('http://google.com' +\
         linkElems[i].get('href'))
```

Fetching Weather info

- Reads the requested locations from the command line.
- Downloads JSON weather data from OpenWeatherMap.org.
- Prints the weather for today and the next two days.

Becomes as easy as Kid's play?

```
import json, requests, sys

location = ' '.join(sys.argv[1:])
url
     ='http://api.openweathermap.org/data/2.5/forecast/daily?q={}&cnt=3'.format(location)
response = requests.get(url)

weatherData = json.loads(response.text)
w = weatherData['list']

print(w[0]['weather'][0]['main'], '-',
     w[0]['weather'][0]['description'])
print(w[1]['weather'][0]['main'], '-',
     w[1]['weather'][0]['description'])
print(w[2]['weather'][0]['main'], '-',
     w[2]['weather'][0]['description'])
```

Automating Boring Stuff

(Ref: https://automatetheboringstuff.com/)

Backing up music files

- Say, you have hundreds of mp3s scattered all over the folders on your hard disk.
- Wish to get them in a folder and zip them up for backup

Is there any automatic program you have used to do so?

Copying files

- *shutil* (or shell utilities) module has functions to let you copy, move, rename, and delete files in your Python programs
- use *shutil.make_archive*. It supports both zip and tar formats.

All these lines can be typed on command line Python Shell as well.
Only some bits are shown here:

```
import glob, shutil
src_dir = '/home/my/Downloads/'
dest_dir = '/home/my/Music/Songs/'

for f in glob.iglob(src_dir +'**/*.cmp3',
     recursive=True):
    shutil.copy(f, dest_dir)

shutil.make_archive('mysongs.zip', 'zip', dest_dir)
```

Engineering

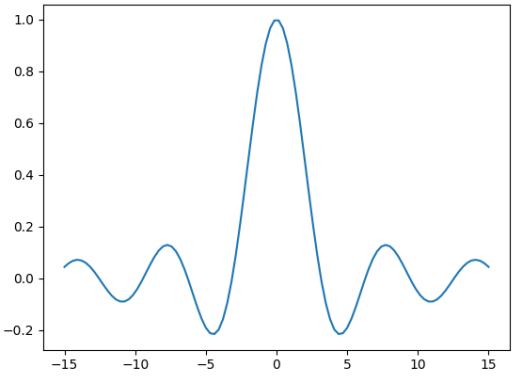
Engineering

Wish to check how this eqn looks like: $sin(x)/x$

```
import numpy
import matplotlib.pyplot as plt

x = numpy.linspace(-15,15,100) # 100 linearly
     spaced numbers
y = numpy.sin(x)/x # computing the values of
     sin(x)/x

plt.plot(x,y) # sin(x)/x
plt.show() # show the plot
```



Python for AI

What's going on in AI?

- Companies/platforms
- Achievements,
- and products to use.

What Languages Are Used for Building AI?

- LISP (used to be?) one of the most popular languages for creating AI. Dynamic typing and lists.
- C/C++ is used for building performance AI, but not so easy to program.
- Java is not as fast as C but its portability and built-in types make Java a choice of many developers.
- And finally, there is Python.

Why Python scores?

- Simple, clean syntax
- Many powerful libraries
- Many domains
- Language interoperability
- Free!!

Libraries

Related to AI-ML-DL

- Numpy, Scipy for mathematics
- Pandas from data processing
- Matplotlib for visualizations
- Scikit Learn for Machine Learning
- and many more

Machine Learning Topics with Python

- Regression
- Classification
- Clustering
- Dimension Reduction
- and many more

Deep Learning with Python

- Neural Networks
- Convolution Neural Network
- Recurrent Neural Network
- Libraries: Tensorflow, Keras, Pytorch, etc

What Next?

Resources:

- Google Developers Python Course (highly recommended for visual learners)
- An Introduction to Python for Scientific Computing (from UCSB Engineering), by M. Scott Shell (a great scientific Python intro 60 pages)
- Datacamp online course
- Learn X in Y Minutes (X = Python)
- Book "Python The Hard Way" by Zed A. Shaw (a 30 minute crash course)

If you think, you need a human to teach, then join the workshop !!!

References

References

Recommended Learning Resources

- Data Camp's Intro to Python for data science
- Al Sweigart published Automate the Boring stuff. Available through multiple channels: e-book, printed book, and Udemy course.
- edX offers Python for data science and Introduction to Computer Science and Programming Using Python, taught by Eric Grimson.
- Coursera's course on Introduction to interactive programming in Python
- Kaggle offers an online Python course based on Jake Vanderplas' Whirlwind Tour of Python
- Zed A. Shaw's book Learn Python the hard way
- Brian Godsey's book Think like a data scientist
- Learn X in Y minutes has a useful set of examples in Python3

(Ref: "Python resources" - Brandon Rohrer)

References

Many publicly available resources have been refereed for making this presentation. Some of the notable ones are:

- Python language courses - University of Zurich
- An introduction to Python programming with NumPy, SciPy and Matplotlib/Pylab - Antoine Lefebvre
- Automate the Boring Stuff with Python - Al Sweigart
- Python Evangelism 101 - Peter Wang
- A Python Book: Beginning Python, Advanced Python, and Python Exercises - Dave Kuhlman
- Python for Everybody - Charles R. Severance
- Python for Science - Ward Marshall ,

Suggested Learning Resources:

- Book: van Rossum, G. (2011). The Python Tutorial
- Book: Learning Python by Mark Lutz & David Ascher. O'Reilly and Associates
- Python Cookbook
- 100+ Python challenging programming exercises