Computational Environments and Toolchains

Topic 01 — Intro to Python

Lecture 01 — A Quick Python Tour

Kieran Murphy and David Power

Department of Science,
Waterford IT.
(kmurphy@wit.ie,dpower@wit.ie)

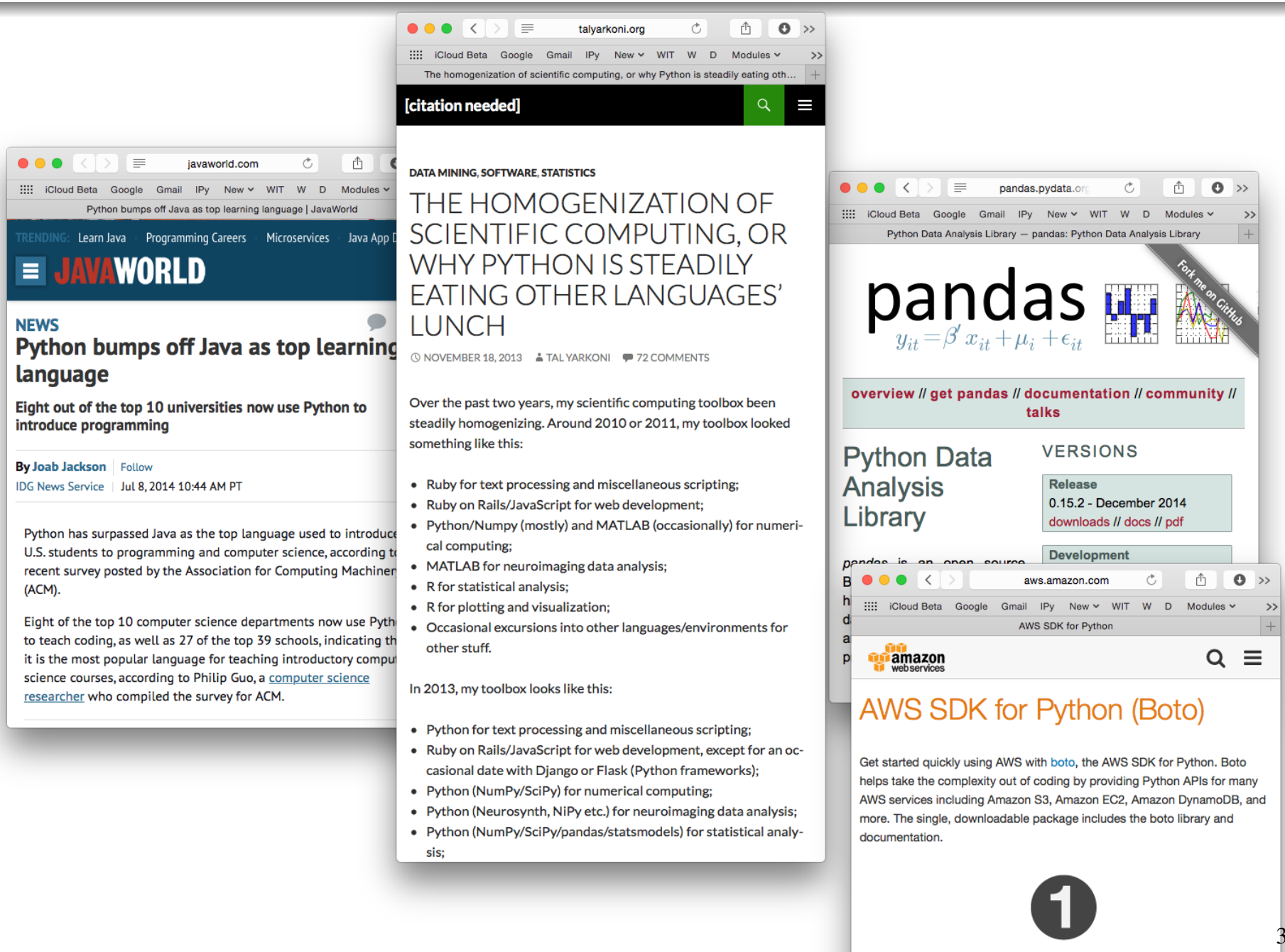
Autumn Semester, 2021

Outline

- A brief history of Python
- Some Python workflows
- Python language and idioms

Outline

Why Python?


javaworld.com
Python bumps off Java as top learning language | JavaWorld
JAVAWORLD
NEWS
Python bumps off Java as top learning language
Eight out of the top 10 universities now use Python to introduce programming
By Joab Jackson
IDG News Service
Jul 8, 2014 10:44 AM PT
talyarkoni.org
The homogenization of scientific computing, or why Python is steadily eating oth...
[citation needed]
DATA MINING, SOFTWARE, STATISTICS
THE HOMOGENIZATION OF SCIENTIFIC COMPUTING, OR WHY PYTHON IS STEADILY EATING OTHER LANGUAGES' LUNCH
NOVEMBER 18, 2013
TAL YARKONI
72 COMMENTS
Over the past two years, my scientific computing toolbox been steadily homogenizing. Around 2010 or 2011, my toolbox looked something like this:
Ruby for text processing and miscellaneous scripting;
Ruby on Rails/JavaScript for web development;
Python/Numpy (mostly) and MATLAB (occasionally) for numerical computing;
MATLAB for neuroimaging data analysis;
R for statistical analysis;
R for plotting and visualization;
Occasional excursions into other languages/environments for other stuff.
In 2013, my toolbox looks like this:
Python for text processing and miscellaneous scripting;
Ruby on Rails/JavaScript for web development, except for an occasional date with Django or Flask (Python frameworks);
Python (NumPy/SciPy) for numerical computing;
Python (Neurosynth, NiPy etc.) for neuroimaging data analysis;
Python (NumPy/SciPy/pandas/statsmodels) for statistical analysis;
pandas.pydata.org
Python Data Analysis Library — pandas: Python Data Analysis Library
pandas
Fork me on GitHub
overview // get pandas // documentation // community // talks
Python Data Analysis Library
VERSIONS
Release
0.15.2 - December 2014
downloads // docs // pdf
Development
aws.amazon.com
AWS SDK for Python
amazon web services
AWS SDK for Python (Boto)
Get started quickly using AWS with boto, the AWS SDK for Python. Boto helps take the complexity out of coding by providing Python APIs for many AWS services including Amazon S3, Amazon EC2, Amazon DynamoDB, and more. The single, downloadable package includes the boto library and documentation.

Python in 30 Seconds

Highlights

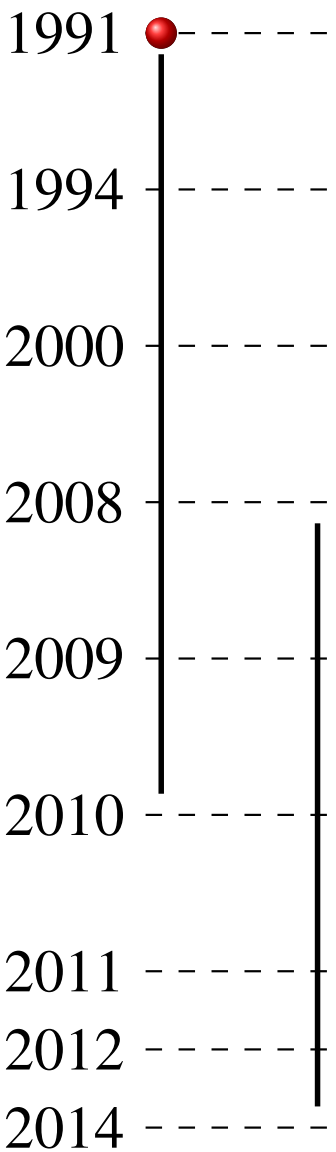
- Modern, interpreted, object-oriented, multi-paradigm (object-oriented, procedural, functional) programming language.
- Portable (Unix/Linux, Mac OS X, Windows, iOS (sort of))
- Open source, IP rights held by the Python Software Foundation.

Language

- A dynamically typed and strongly typed language.
- Indentation is meaningful — no braces to delimitate control blocks.
- Powerful subscripting (slicing)
- Everything is an object.
- Object attributes can be added/removed at run time.
- Exceptions as in Java.
- Iterators (like Java) and generators.
- Optional and named parameters.
- Goal:

"There should be one—and preferably only one—obvious way to do it."

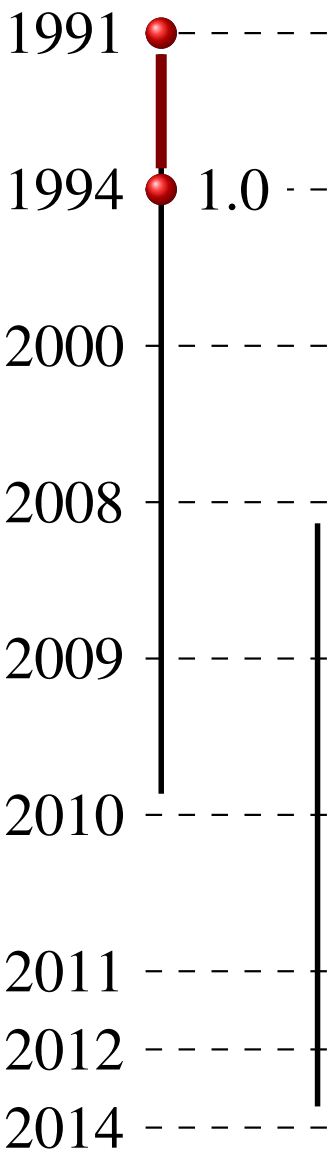
History of Python



- Developed by Guido van Rossum.

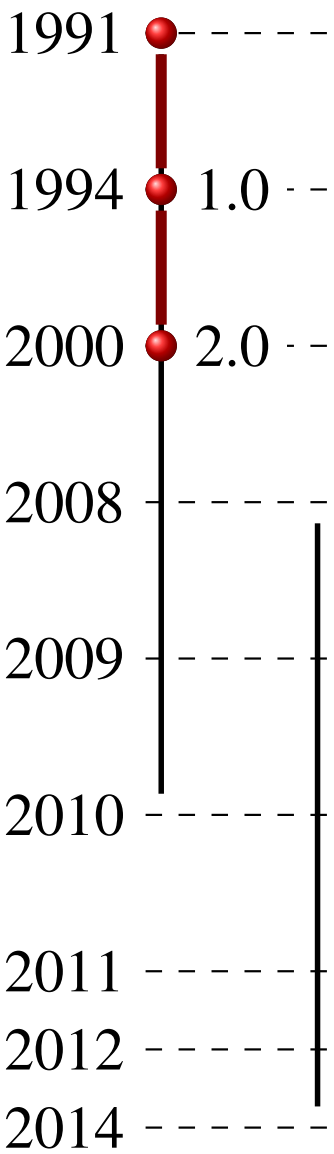
- Free (as in speech) from the beginning, currently released under a GPL compatible license.
- Guido is now the Python “Benevolent dictator for life”.

History of Python



- Developed by Guido van Rossum.
- Introduced functional programming tools: `lambda`, `map`, `filter`, and `reduce`.

History of Python



- Developed by Guido van Rossum.
- Introduced functional programming tools: **`lambda`**, **`map`**, **`filter`**, and **`reduce`**.
- Introduced list comprehensions — compact syntax for building lists based on existing lists.

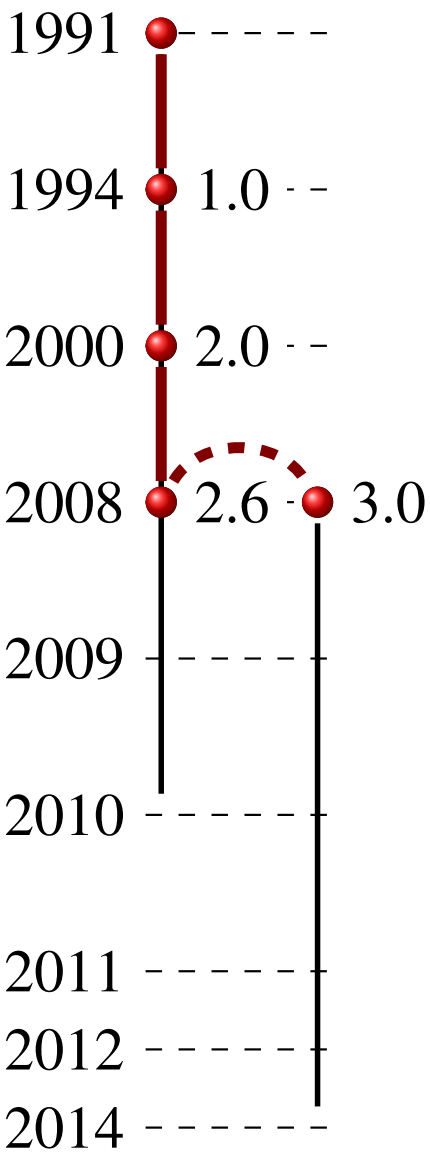
List comprehensions are modelled on the set-builder notation in mathematics, for example consider the set of all positive, even integers less than 100. Using the set builder notation used in mathematics we would write

$$\{x | x \in \mathbb{N}, x < 100, x \text{ is even}\}$$

In python, using list comprehensions, we write

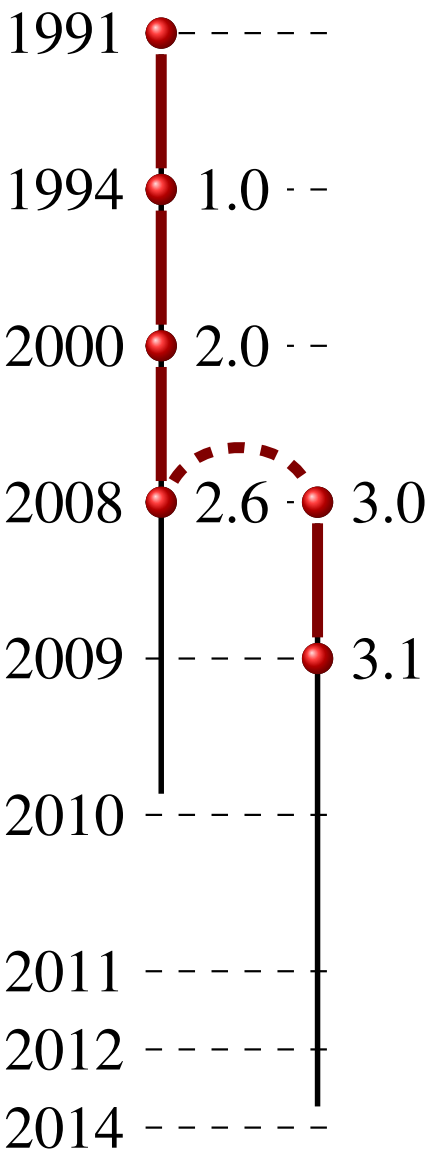
```
[x for x in range(1,100) if x%2==0]
```

History of Python



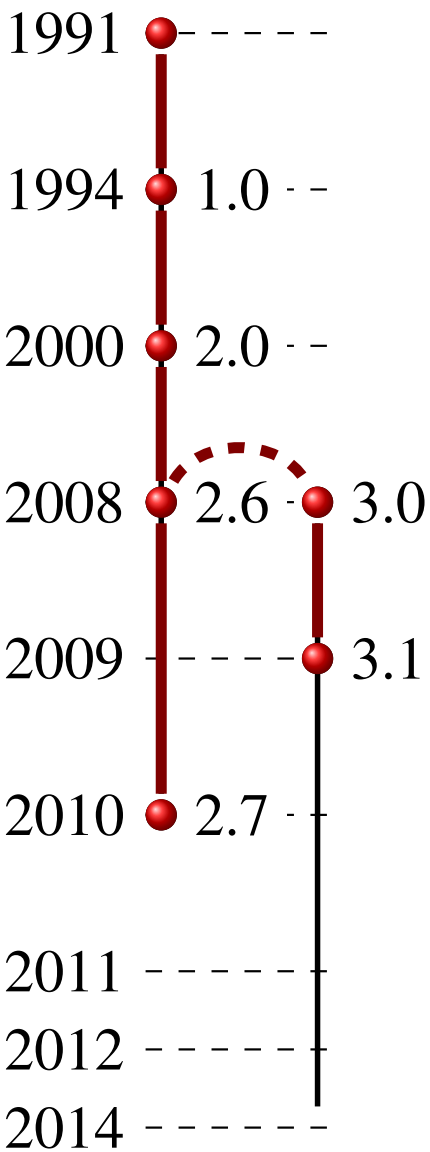
- Developed by Guido van Rossum.
- Introduced functional programming tools: `lambda`, `map`, `filter`, and `reduce`.
- Introduced list comprehensions — compact syntax for building lists based on existing lists.
- Python 3.0 was intended to rectify fundamental design flaws in the language which resulted in breaking full backwards compatibility.
  - Still multi-paradigm but *"There should be one—and preferably only one—obvious way to do it."*
  - Branch 2.x updated with some of the 3.x features.
  - Transition from 2.x to 3.0 is difficult (not automatic).
  - Coders wanting to maintain compatibility with both branches should develop in 2.x and use script 2to3.py to aid conversion.

History of Python



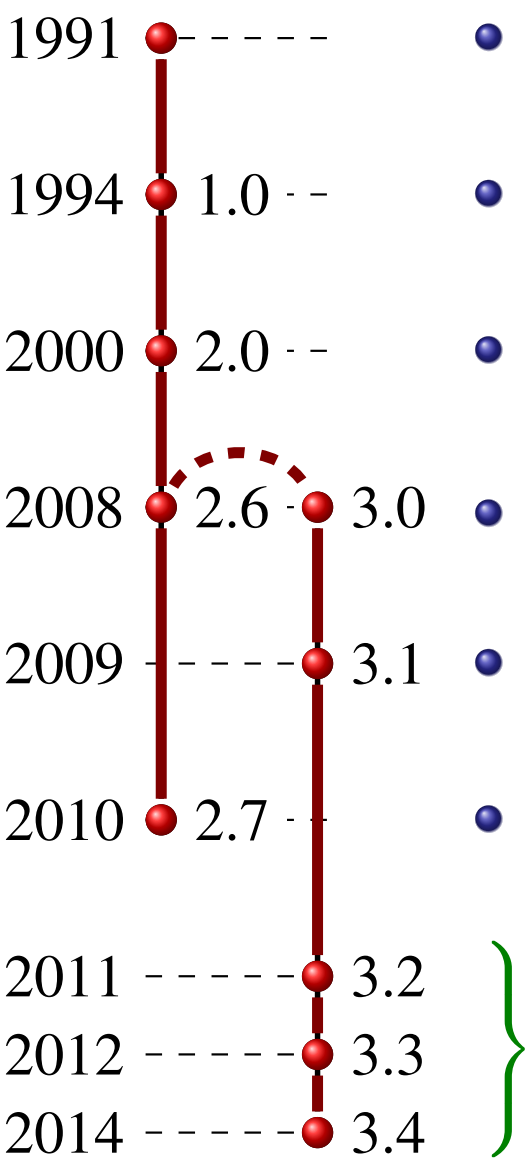
- Developed by Guido van Rossum.
- Introduced functional programming tools: `lambda`, `map`, `filter`, and `reduce`.
- Introduced list comprehensions — compact syntax for building lists based on existing lists.
- Two separate incompatible branches.
- Added new collections: OrderedDict, Counter, ...
- Dictionary and set comprehensions.

History of Python



- Developed by Guido van Rossum.
- Introduced functional programming tools: `lambda`, `map`, `filter`, and `reduce`.
- Introduced list comprehensions — compact syntax for building lists based on existing lists.
- Two separate incompatible branches.
- Added new collections (OrderedDict and Counter) and dictionary and set comprehensions.
- Update of 2.x branch with some of the 3.x features, in particular, collections: OrderedDict, Counter
  - Last of the 2.x releases.
  - Will be supported until 2020.

History of Python



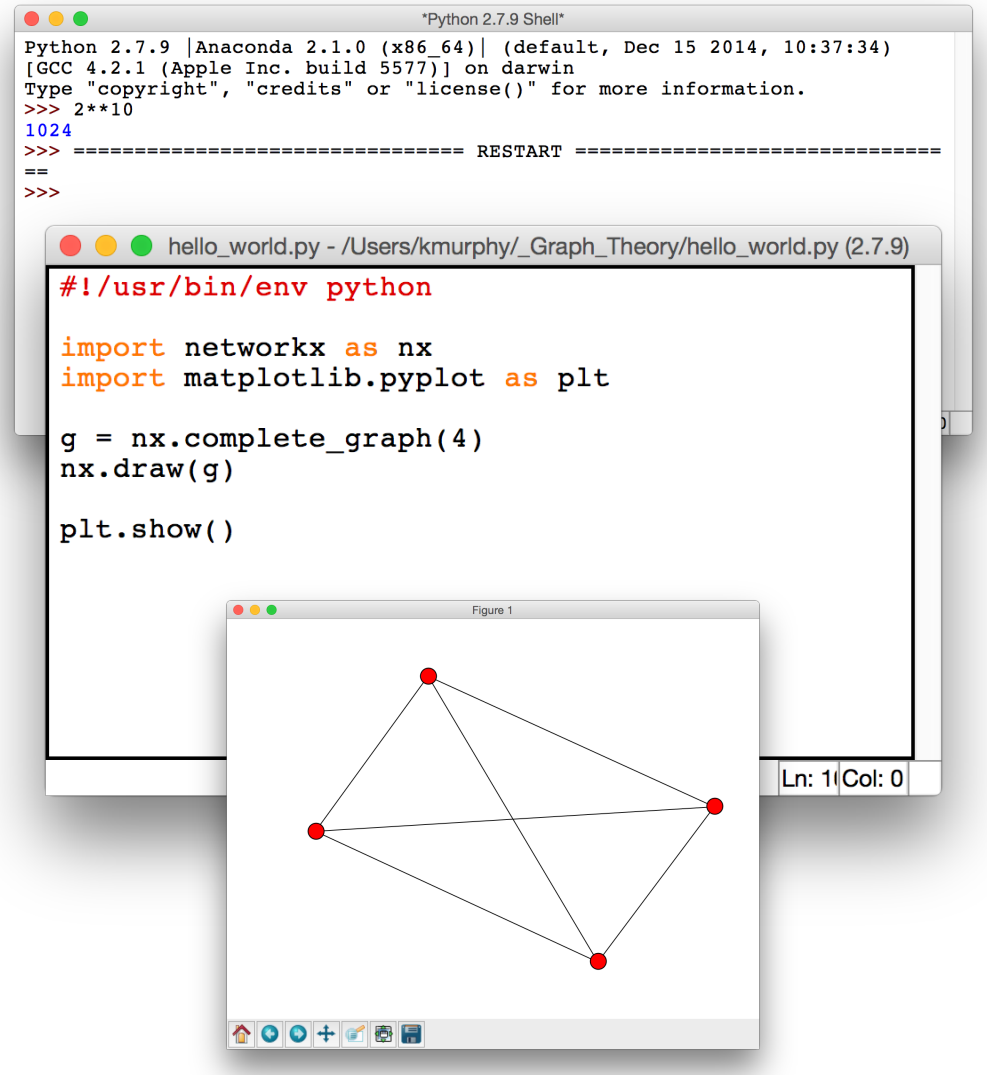
- Developed by Guido van Rossum.
- Introduced functional programming tools: **`lambda`**, **`map`**, **`filter`**, and **`reduce`**.
- Introduced list comprehensions — compact syntax for building lists based on existing lists.
- Two separate incompatible branches.
- Added new collections (OrderedDict and Counter) and dictionary and set comprehensions.
- Last of the 2.x releases, some features from 3.1.

Improved libraries, performance enhancements . . .

"Python 2.x is legacy, Python 3.x is the present and future of the language"

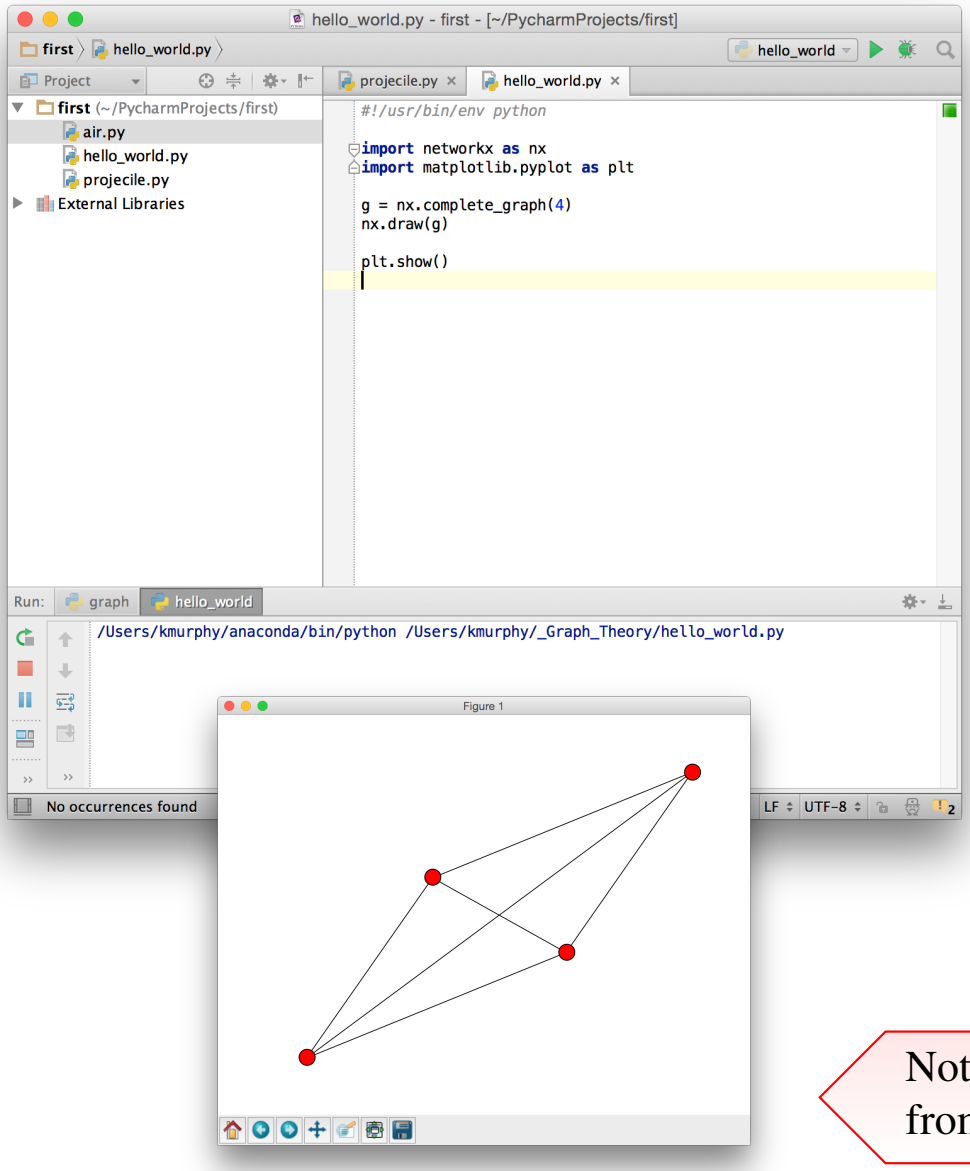
Outline

IDLE



- Default IDE with python.
- Consists of an interactive window running a python shell and separate window for each file.
- Limited code-assist features (call tip, code completion, but no line numbers).
- The IDLE extensions for Python (idlex) adds numerous extensions to IDLE from line numbers, running code selections, to a IPython shell.

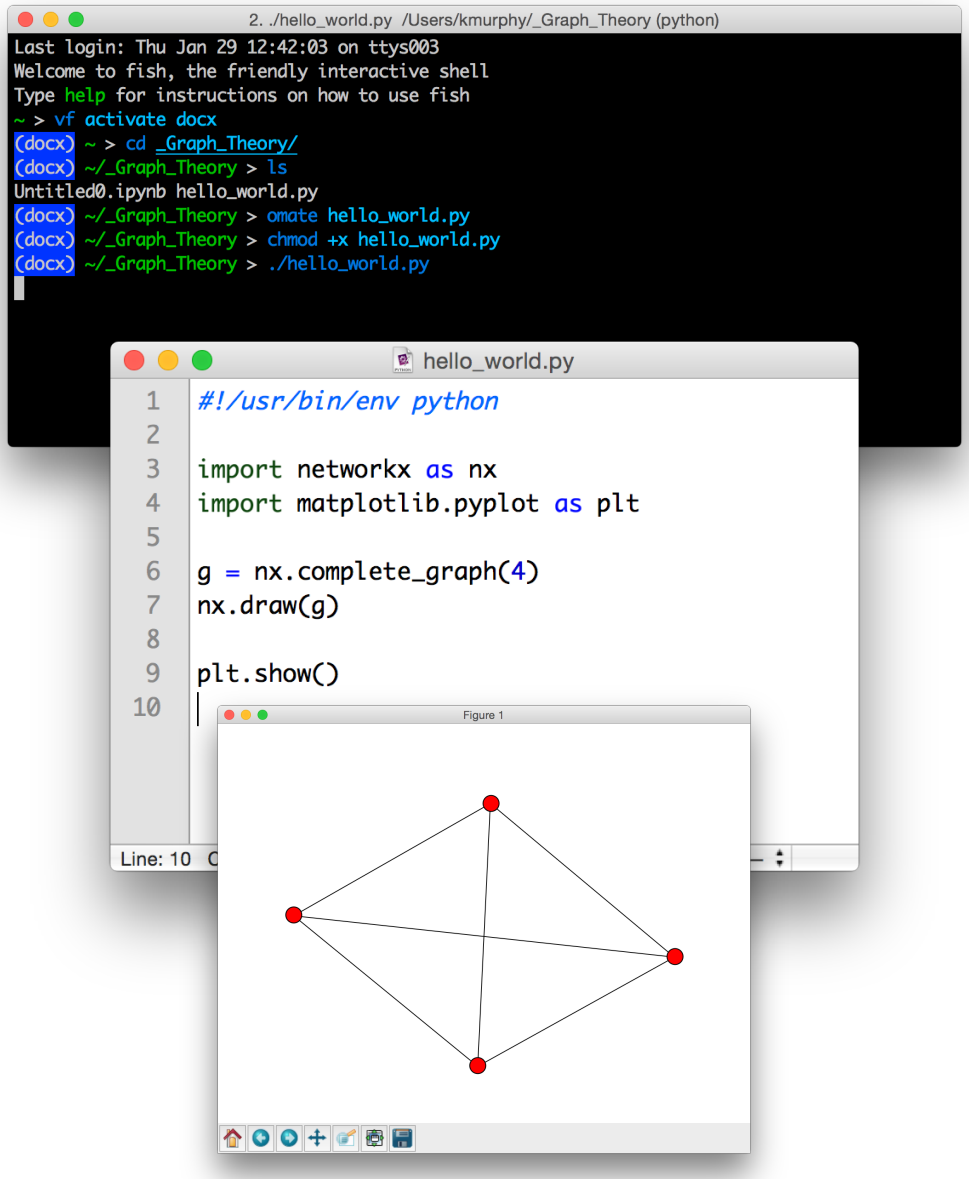
PyCharm



- The eclipse of the python world ...
- Three versions:
  - Community (used here),
  - Educational (includes an interactive python tutorial)
  - Professional (I will email licence for use due this module on request
- Has an Interactive Python (Python) window but it is limited (no code competition, tool tips, etc).

Note that the graph layout is different from previous slide!

Terminal + Editor



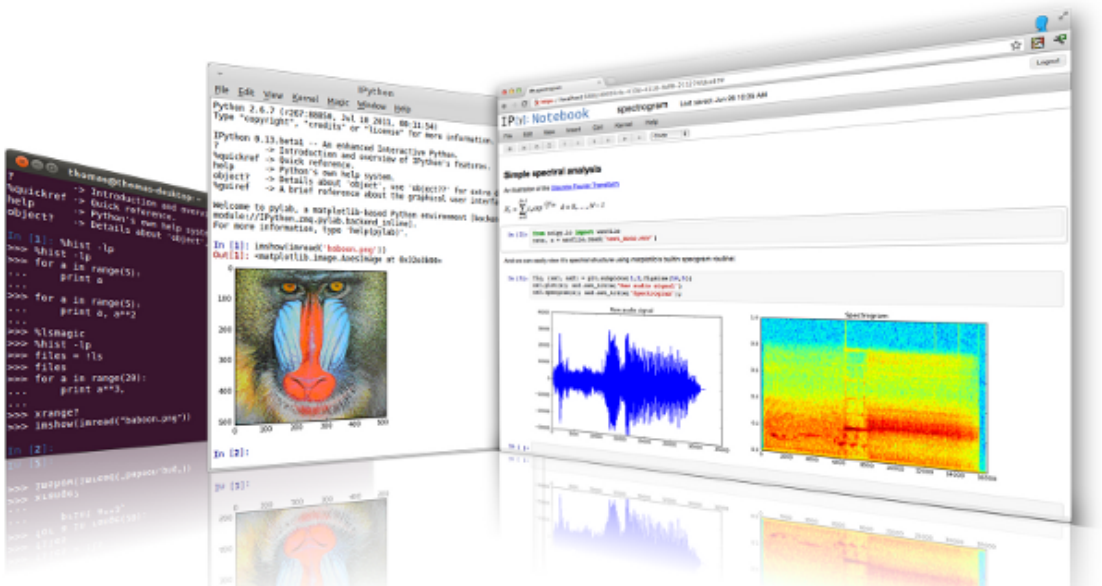
- Near ideal, low-level quick and simple development environment for Python.
- Code development tools based on editor (TextMate).
- Make script executable and run directly with

  ./ script_name . py

  or via

  python script_name . py
- Ideal for automation but slow starting python interpreter on every execution $\Longrightarrow$ interactive python.

IPython



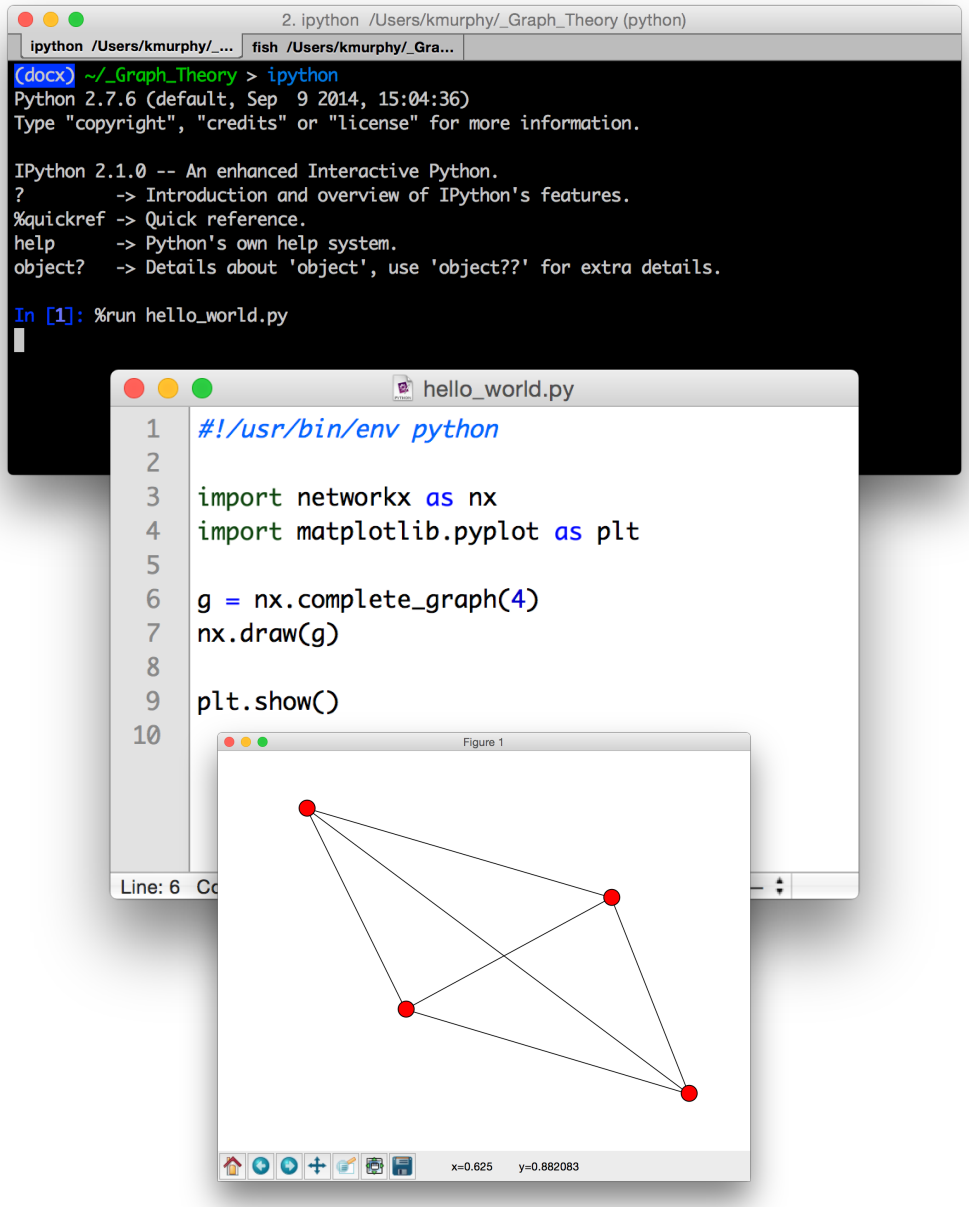
- Formatted interactive documents using markdown for non-code content, MathJax for equations and javascript for animation.
- Interactive experimentation and development of open source academic papers.

An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures.

—D. Donoho, Stanford

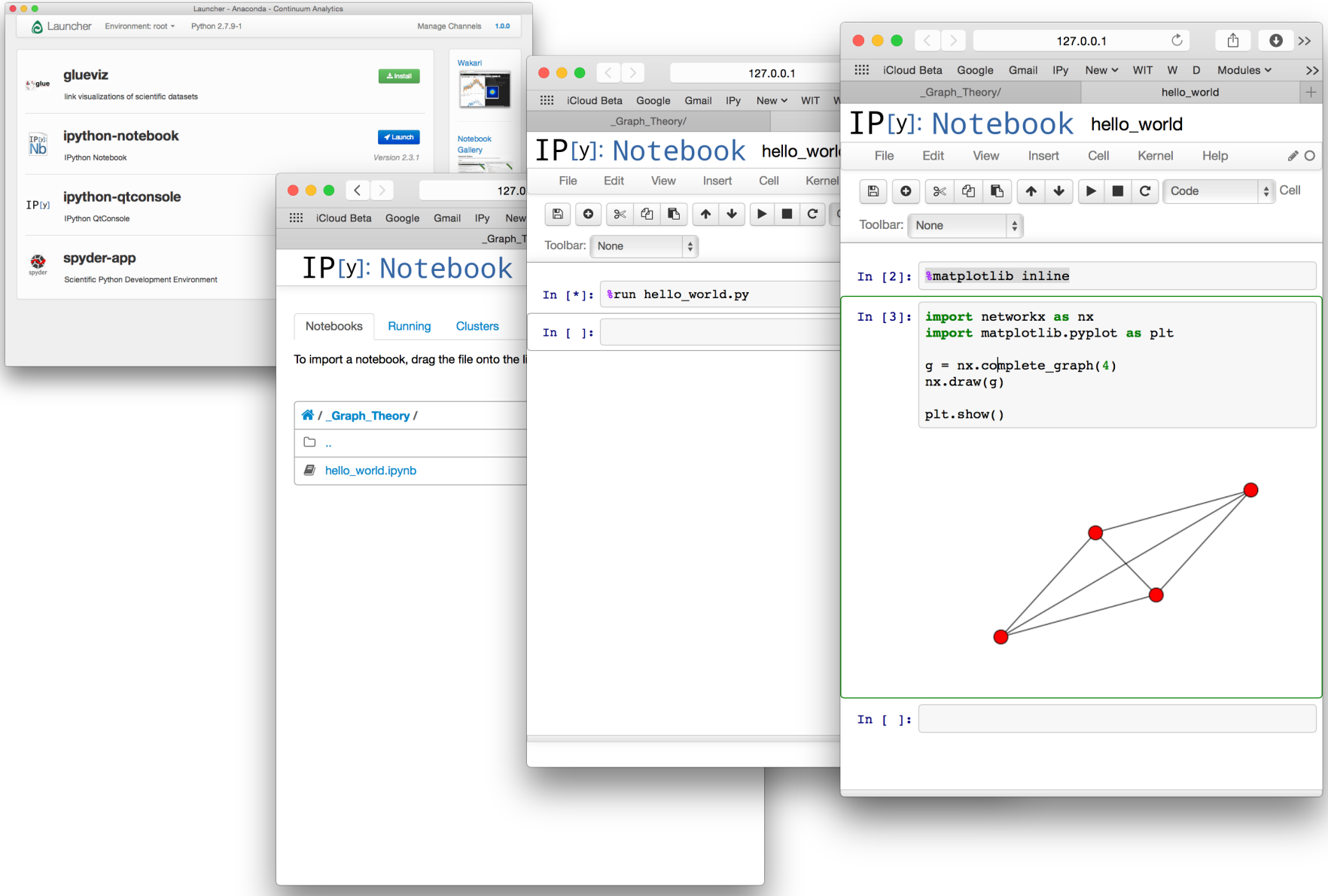
- Supports multiple interactive shells (terminal and Qt-based).
- **A browser-based notebook with support for code, text, mathematical expressions, inline plots and other rich media.**
- Easy to use, high performance tools for parallel computing.
- Growing set of magic commands (%run, %timeit, %timeit, etc) to simplify interactive experimentation and development.

IPython in a Terminal Window



- Ideal, low-level quick and simple development environment for Python.
- All advantages of standard terminal python development but now with interactive prompt to check code and faster startup on execution using IPython magic command %run as in %run script_name.py

IPython Notebook



Outline

General Comments

Whitespace

- Whitespace is meaningful in Python: especially indentation and placement of newlines.
  - Use a newline to end a line of code.
  - Use backslash, or better round brackets, when must go to next line prematurely.
- No braces { and } to mark blocks of code in Python . . . Use consistent indentation instead.
  - The first line with less indentation is outside of the block.
  - The first line with more indentation starts a nested block
- Usually a colon appears at the start of a new block.

Comments

- End of line comments character #
- Can include a "documentation string" as the first line of any new function or class that you define.

Data Types and Variables

Java: Statically typed

- Variables are declared to refer to objects of a given type.
- Methods use type signatures to enforce contracts

Python: Strongly but dynamically typed

- Variables come into existence when first assigned to.
- A variable can refer to an object of any type.
- All types are (almost) treated the same way.
- Strongly typed $\Longrightarrow$ objects have (strong) types, names don't. So need explicit conversion between data types.

Basic Data Types

int

- Whole numbers (negative, positive or zero)
- "No upper limit".
- Operators:

  +, −, ∗, / (division), ∗∗ (exponentiation).
  % (modulus), // (integer division).

float

- Unlike integers, there are limits on scale and precision with floats; float values are approximation of the true value.
- You can use **int**() and **float**() to convert between int and float.

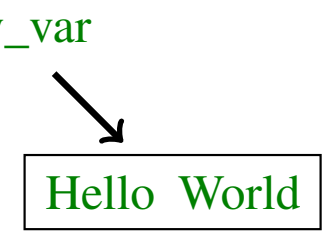
string

- Can use " or ' to specify.
- Use triple double-quotes for multi-line strings or string that contain " or ' inside of them.

Python Variables and Assignment

- Python allows us to attach “labels” to quantities

```
>>> my_var = 'Hello World'
```



- We may subsequently refer to these quantities by their labels

```
>>> print(my_var)
Hello World
```

- On reassignment the label is bound to a new object.
- Multiple assignments are supported

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

If Statements (as expected)

if_statement .py

```
5  x = 5
6  if x == 3:
7      print ('x equals 3.')
8  elif x == 2:
9      print ('x equals 2.')
10 else:
11     print ('x equals something else.')
12 print ("This is outside the 'if'.")
```

```
x equals something else.
This is outside the 'if'.
```

- Use of indentation for blocks
- Colon (:) after boolean expression

While Loops (as expected)

while_loops.py

```
5  x = 3
6  while x < 5:
7      print (x, 'still in first loop')
8      x = x + 1
9
10 x = 6
11 while x < 5:
12     print (x, 'still in second loop')
```

```
3 still in first loop
4 still in first loop
```

- You can use the keyword **break** inside a loop to leave the loop entirely.
- You can use the keyword **continue** inside a loop to stop processing the current iteration of the loop and immediately go on to the next one.

Functions

Function definition begins with **def**

Function name and its arguments.

```
def get_final_answer(filename):
    """Documentation String"""
    line1
    line2
    return total_counter
...
```

Colon.

First line with less indentation is considered to be outside of the function definition.

'return' indicates the value to be sent back to the caller.

No declaration of types of arguments or result

Functions II

Return value

- All functions in Python have a return value — even if no return line inside the code.
- Functions without a return return the special value None.
- None is a special constant in the language.
- None is used like null in Java.
- None is also logically equivalent to False.

Function overloading

- There is no function overloading in Python.
- Unlike Java, a Python function is specified by its name alone
  - The number, order, names, or types of its arguments cannot be used to distinguish between two functions with the same name.
  - Two different functions can't have the same name, even if they have different numbers of arguments.

Functions are First-Class Objects in Python

- Functions can be used just like any other data
- They can be
  - Arguments to function
  - Return values of functions
  - Assigned to variables
  - Parts of tuples, lists, etc ...

functions.py

```
5  def myfun(x):
6      return x*3
7
8  def apply(q, x):
9      return q(x)
10
11 print(apply(myfun, 7))
```

```
21
```

Function Arguments

Optional arguments

- You can provide default values for a function's arguments.
- These arguments are optional when the function is called.

functions.py

```
13 def myfun2(b, c=3, d=0):
14     return b + c + d
15 print(myfun2(5,3,9))
16 print(myfun2(5,3))
17 print(myfun2(5))
```

```
17
8
8
```

Keyword Arguments

- Functions can be called with arguments out of order.
- These arguments are specified in the call.
- Keyword arguments can be used for a final subset of the arguments.

functions.py

```
19 def myfun3(a, b, c):
20     return a-b*c
21
22 print(myfun3(2, 1, 4))
23 print(myfun3(c=4, b=1, a=2))
24 print(myfun3(2, c=4, b=1))
```

```
-2
-2
-2
```

Lambda Notation (anonymous functions)

- Functions can be defined without giving them names.
- This is most useful when passing a short function as an argument to another function.

```
>>> apply (lambda x: x*4,  7)
28
```

- The first argument to `apply()` is an unnamed function that takes one input and returns the input multiplied by four.
- Note: only single-expression functions can be defined using this lambda notation.

Sequence Types

Python supports three sequence types (tuples, strings, and lists) with similar syntax and functionality.

Tuple

- A simple immutable ordered sequence of items.
- Items can be of mixed types, including collection types

List

- Mutable ordered sequence of items of mixed types.

String

- Immutable
- Conceptually very much like a tuple
- Regular strings use 8-bit characters. Unicode strings use 2-byte characters. (This is changed in Python 3.)

Sequence Types — Defining . . .

Tuple

- Tuples are defined using parentheses (and commas).

```
>>> my_tuple = (23, 'abc', 4.56, (2,3), 'def')
```

- Note: need the comma if defining a tuple with only one element.

```
>>> no_mates = (23, )
```

List

- Lists are defined using square brackets (and commas).

```
>>> my_list = ['abc', 34, 4.34, 23]
```

String

- Strings are defined using quotes (', ", or """).

```
>>> st = 'Hello World'
>>> st = "Hello World"
>>> st = """This is a multi-line
string that uses triple quotes."""
```

Sequence Types — Assessing individual elements . . .

- We can access individual members of a tuple, list, or string using square bracket “array” notation.
- Note that all are 0 based . . .

Tuple

```
>>> my_tuple = (23, 'abc', 4.56, (2,3), 'def')
>>> my_tuple[1]
'abc'
```

List

```
>>> my_list = ['abc', 34, 4.34, 23] >>> my_list[1]
34
```

String

```
>>> st = 'Hello World'
'e'
```

Negative Indices

```
>>> my_tuple = (23, 'abc', 4.56, (2,3), 'def')
```

- Positive index: count from the left, starting with 0.

```
>>> my_tuple[1]
'abc'
```

- Negative lookup: count from right, starting with -1.

```
>>> my_tuple[-3]
4.56
```

Slicing: Return Copy of a Subset I

Slicing

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying **before** the second index.

```
>>> my_tuple = (23, 'abc', 4.56, (2,3), 'def')
```

- Return a copy of the container with a subset of the original members.

```
>>> my_tuple[1:4]
('abc', 4.56, (2,3))
```

- You can also use negative indices when slicing.

```
>>> my_tuple[1:-1]
('abc', 4.56, (2,3))
```

- Optional argument allows selection of every n^{th} item.

```
>>> my_tuple[1:-1:2]
('abc', (2,3))
```

Python used semi-open intervals

Slicing: Return Copy of a Subset II

```
>>> my_tuple = (23, 'abc', 4.56, (2,3), 'def')
```

- Omit the first index to make a copy starting from the beginning of the container.

```
>>> my_tuple[:2]
(23, 'abc')
```

- Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> my_tuple[2:]
(4.56, (2,3), 'def')
```

- To make a copy of an entire sequence, you can use `[:]`.

```
>>> my_tuple[:]
(23, 'abc', 4.56, (2,3), 'def')
```

- Note the difference between these two lines for mutable sequences:

```
>>> list2 = list1
```

which results in two names referring to one object

```
>>> list2 = list1 [:]
```

which is two independent copies.

The `in` and + Operators

- The `in` operator test whethers a value is inside a collection (often called a container in Python:

list and tuples

```
>>> t =[1,2,4,5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

strings

Tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- The + produces a **new** tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
```

Further List Operations

append

```
>>> li = [1, 11, 3, 4, 5]
>>> li.append('a')
>>> li
[1, 11, 3, 4, 5, 'a']
```

insert

```
>>> li.insert(2,'i')
>>> li
[1, 11, 'i', 3, 4, 5, 'a']
```

index

Index of first occurrence

```
>>> li.index('a')
6
```

extend

```
>>> li.extend([9,8,7])
>>> li
[1, 11, 'i', 3, 4, 5, 'a',9,8,7]
```

Operator extend is like add in Java; it operates on the list in place.
In contrast operator + creates a fresh list (with a new memory reference)

Also: remove, reverse, sort

For loop

- We often want to write a loop where the variables ranges over some sequence of numbers. The **range**() function returns a list of numbers from 0 up to but not including the number we pass to it.

for_loops.py

```
3 from __future__ import print_function
4
5 # print integers 0, 1, 2, 3, 4
```

- Before version 2.7 the **range** function returned a list — so was unsuitable for large ranges. Instead **xrange**, which returns an iterator, was used.
- Don't use **range**() to iterate over a sequence solely to have the index and elements available at the same time.

for_loops.py

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

for_loops.py

```
14 for i,v in enumerate(mylist):
15     print (i, v)
```

Dictionaries

- Like maps in Java
- Dictionaries store a mapping between a set of keys and a set of values.
- Keys can be any immutable type.
- Values can be any type.

Creating and accessing dictionaries

- An empty dictionary ...

```
>>> d ={}
```

- Initialising with some key-value pairs...

```
>>> d ={'user':'bozo', 'pswd':1234}
```

Assessing and Updating Dictionaries

```
>>> d ={'user':'bozo', 'pswd':1234}
```

Accessing

```
>>> d['user']
'bozo'
```

Keys must exist . . .

```
>>> d['pswd']
1234
>>> d['bozo']
Traceback (innermost last):
   File '<interactive input>' line 1, in ?
KeyError: bozo
```

Updating

```
>>> d['pswd']='monkey'
>>> d['pswd']
'monkey'
>>> d['id']='45'
>>> d
{'user':'bozo', 'pswd':'monkey','id':45}
```

Accessor Methods

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}
```

- List of current keys

```
>>> d.keys()
['user', 'p', 'i']
```

- List of current values

```
>>> d.values()
['bozo', 1234, 34]
```

- List of item tuples

```
>>> d.items()
[('user','bozo'), ('p',1234), ('i',34)]
```

Looping over Dictionaries

looping_over_dict .py

```
1  #!/usr/bin/env python
2
3  from __future__ import print_function
4
5  d = {'user':'bozo', 'pswd':'boo', 'id':45}
6
7  print('\nLooping over keys ...')
8  for k in d:
9      print (k)
10
11 print('\nLooping over items ...')
12 for k, v in d.items():
13     print(k, '-->', v)
```

```
Looping over keys ...
pswd
user
id

Looping over items ...
pswd --> boo
user --> bozo
id --> 45
```