Principles/Social Media Mining

CIS 600

Weeks 1-2: Python Tutorial

Edmund Yu, PhD
Associate Teaching Professor
esyu@syr.edu

August 27 & September 1, 2020

Python: The Origin

History [edit]

Main article: History of Python

Guido van Rossum at OSCON 2006

Python was conceived in the late 1980s[35] by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC language (itself inspired by SETL),[36] capable of exception handling and interfacing with the Amoeba operating system.[8] Its implementation began in December 1989.[37] Van Rossum shouldered sole responsibility for the project, as the lead developer, until 12 July 2018, when he announced his "permanent vacation" from his responsibilities as Python's *Benevolent Dictator For Life*, a title the Python community bestowed upon him to reflect his long-term commitment as the project's chief decision-maker.[38] He now shares his leadership as a member of a five-person steering council.[39][40][41] In January 2019, active Python core developers elected Brett Cannon, Nick Coghlan, Barry Warsaw, Carol Willing and Van Rossum to a five-member "Steering Council" to lead the project.[42]

Python 2.0 was released on 16 October 2000 with many major new features, including a cycle-detecting garbage collector and support for Unicode.[43]

Python 3.0 was released on 3 December 2008. It was a major revision of the language that is not completely backward-compatible.[44] Many of its major features were backported to Python 2.6.x[45] and 2.7.x version series. Releases of Python 3 include the `2to3` utility, which automates (at least partially) the translation of Python 2 code to Python 3.[46]

Python 2.7's end-of-life date was initially set at 2015 then postponed to 2020 out of concern that a large body of existing code could not easily be forward-ported to Python 3.[47][48]

Features and philosophy [edit]

Python is a multi-paradigm programming language. Object-oriented programming and structured programming are fully supported, and many of its features support functional programming and aspect-oriented programming (including by metaprogramming[49] and metaobjects (magic methods)).[50] Many other paradigms are supported via extensions, including design by contract[51][52] and logic programming.[53]

Python: The Origin

- Readability counts.

Rather than having all of its functionality built into its core, Python was designed to be highly extensible. This compact modularity has made it particularly popular as a means of adding programmable interfaces to existing applications. Van Rossum's vision of a small core language with a large standard library and easily extensible interpreter stemmed from his frustrations with ABC, which espoused the opposite approach.[35]

Python strives for a simpler, less-cluttered syntax and grammar while giving developers a choice in their coding methodology. In contrast to Perl's "there is more than one way to do it" motto, Python embraces a "there should be one—and preferably only one—obvious way to do it" design philosophy.[57] Alex Martelli, a Fellow at the Python Software Foundation and Python book author, writes that "To describe something as 'clever' is *not* considered a compliment in the Python culture."[58]

Python's developers strive to avoid premature optimization, and reject patches to non-critical parts of the CPython reference implementation that would offer marginal increases in speed at the cost of clarity.[59] When speed is important, a Python programmer can move time-critical functions to extension modules written in languages such as C, or use PyPy, a just-in-time compiler. Cython is also available, which translates a Python script into C and makes direct C-level API calls into the Python interpreter.

An important goal of Python's developers is keeping it fun to use. This is reflected in the language's name—a tribute to the British comedy group Monty Python[60]—and in occasionally playful approaches to tutorials and reference materials, such as examples that refer to spam and eggs (from a famous Monty Python sketch) instead of the standard foo and bar.[61][62]

A common neologism in the Python community is *pythonic*, which can have a wide range of meanings related to program style. To say that code is pythonic is to say that it uses Python idioms well, that it is natural or shows fluency in the language, that it conforms with Python's minimalist philosophy and emphasis on readability. In contrast, code that is difficult to understand or reads like a rough transcription from another programming language is called *unpythonic*.

Users and admirers of Python, especially those considered knowledgeable or experienced, are often referred to as *Pythonistas*.[63][64]

Syntax and semantics [edit]

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

In other projects
Wikimedia Commons

Print/export

Monty Python

From Wikipedia, the free encyclopedia

"Pythonesque" redirects here. For the play by Roy Smiles, see Pythonesque (play).
This article is about the comedy group. For their TV show frequently called Monty Python, see Monty Python's Flying Circus.

Monty Python (also collectively known as the **Pythons**)[2][3] are a British surreal comedy group who created the sketch comedy television show *Monty Python's Flying Circus*, which first aired on the BBC in 1969. Forty-five episodes were made over four series. The Python phenomenon developed from the television series into something larger in scope and impact, including touring stage shows, films, numerous albums, several books, and musicals. The Pythons' influence on comedy has been compared to the Beatles' influence on music.[4][5][6] Regarded as enduring icons of 1970s pop culture, their sketch show has been referred to as being "an important moment in the evolution of television comedy".[7]

Broadcast by the BBC between 1969 and 1974, *Monty Python's Flying Circus* was conceived, written, and performed by its members Graham Chapman, John Cleese, Terry Gilliam, Eric Idle, Terry Jones, and Michael Palin. Loosely structured as a sketch show, but with an innovative stream-of-consciousness approach aided by Gilliam's animation, it pushed the boundaries of what was acceptable in style and content.[8][9] A self-contained comedy team responsible for both writing and performing their work, the Pythons had creative control which allowed them to experiment with form and content, discarding rules of television comedy. Following their television work, they began making films, which include *Monty Python and the Holy Grail* (1975), *Life of Brian* (1979) and *The Meaning of Life* (1983). Their influence on British comedy has been apparent for years, while in North America, it has coloured the work of cult performers from the early editions of *Saturday Night Live* through to more recent absurdist trends in television comedy. "Pythonesque" has entered the English lexicon as a result.

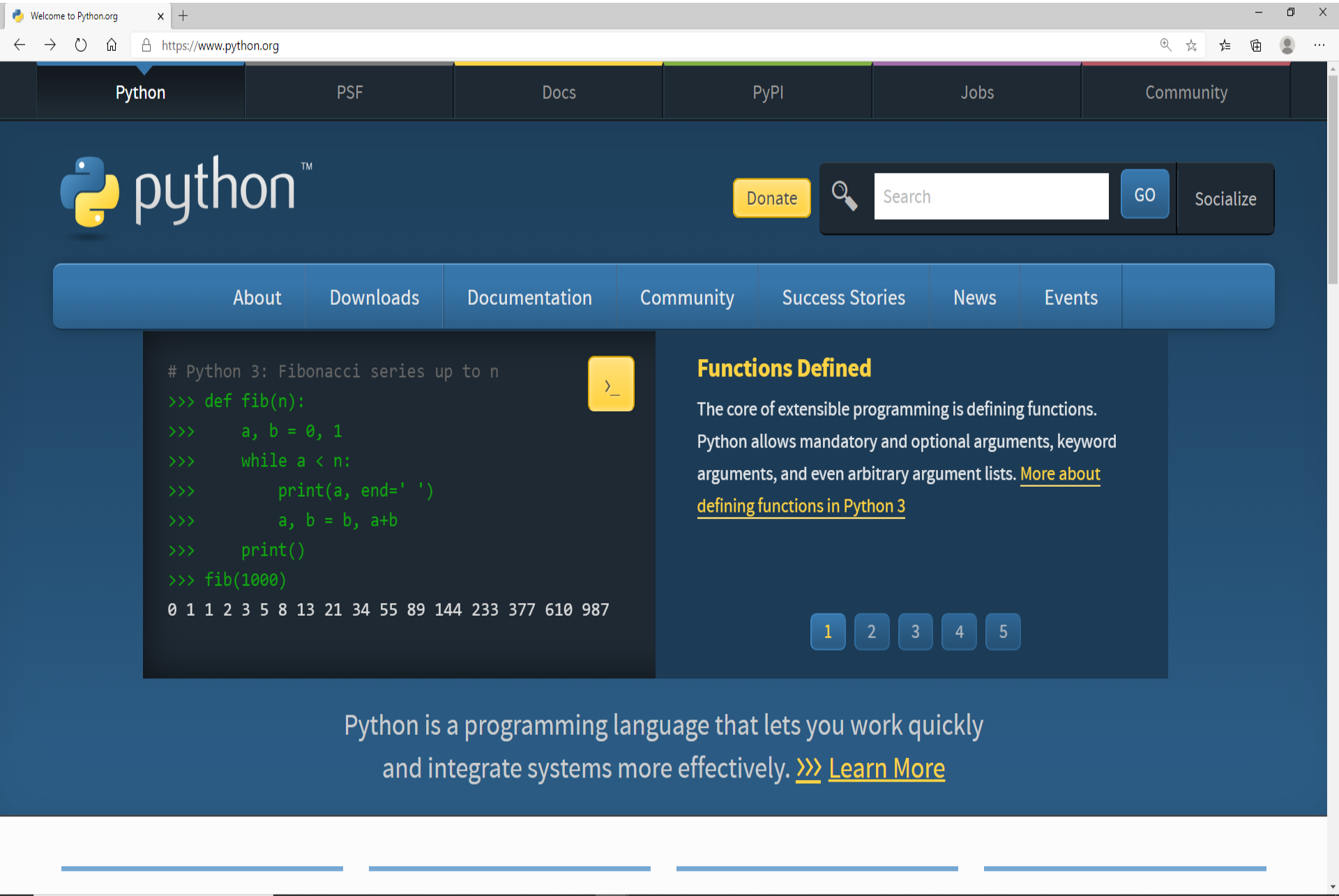
At the 41st British Academy Film Awards in 1988, Monty Python received the BAFTA Award for Outstanding British Contribution To Cinema. In 1998 they were awarded the AFI Star Award by the American Film Institute. Many sketches from their TV show and films are well-known and widely quoted. Both *Holy Grail* and *Life of Brian* are frequently ranked in lists of greatest comedy films. In a 2005 poll of over 300 comics, comedy writers, producers and directors throughout the English-speaking world to find "The Comedian's Comedian", three of the six Pythons

Monty Python

The Pythons in 1969:
Back row: Chapman, Idle, Gilliam
Front row: Jones, Cleese, Palin

Medium	Television · film · theatre · audio recordings · literature
Nationality	British[1]
Years active	1969–1983, 1989, 1998–1999, 2002, 2013–2014
Genres	Satire · surreal humour · black comedy · blue comedy
Notable works and roles	*Flying Circus* (1969–1974) *And Now for Something Completely Different* (1971) *Monty Python and the Holy*

Getting Started: python.org



Download Python | Python.org

https://www.python.org/downloads/

Python | PSF | Docs | PyPI | Jobs | Community

python™

Donate | Search | GO | Socialize

About | Downloads | Documentation | Community | Success Stories | News | Events

Download the latest version for Windows

Download Python 3.8.5

Looking for Python with a different OS? Python for Windows, Linux/UNIX, Mac OS X, Other

Want to help test development versions of Python? Prereleases, Docker images

Looking for Python 2.7? See below for specific releases

Active Python Releases

For more information visit the Python Developer's Guide.

Python version	Maintenance status	First released	End of support	Release schedule

Active Python Releases

For more information visit the Python Developer's Guide.

Python version	Maintenance status	First released	End of support	Release schedule
3.8	bugfix	2019-10-14	2024-10	PEP 569
3.7	security	2018-06-27	2023-06-27	PEP 537
3.6	security	2016-12-23	2021-12-23	PEP 494
3.5	security	2015-09-13	2020-09-13	PEP 478
2.7	end-of-life	2010-07-03	2020-01-01	PEP 373

Looking for a specific release?

Python releases by version number:

Release version	Release date		Click for more
Python 3.7.9	Aug. 17, 2020	Download	Release Notes
Python 3.6.12	Aug. 17, 2020	Download	Release Notes
Python 3.8.5	July 20, 2020	Download	Release Notes
Python 3.8.4	July 13, 2020	Download	Release Notes
Python 3.7.8	June 27, 2020	Download	Release Notes

Python 2 vs 3

- The most noticeable differences are:

Python 2	Python 3
print x	print(x)
4 / 3 = 1	4 / 3 = 1.33333 4 // 3 = 1
raw_input()	input()
file("my_file.txt")	open("my_file.txt")
xrange()	range()

Reading Input from the Keyboard

- Most programs need to read input from the user
- Built-in input function reads input from keyboard
  - Returns the data as a **string**
  - Format: *variable* = **input**(*prompt*)
    - prompt is typically a string instructing user to enter a value
  - Does not automatically display a space after the prompt

Reading Numbers with the *input* Function

- *input* function always returns a string
- Built-in functions convert between data types
  - **int**(*item*) converts *item* to an int
  - **float**(*item*) converts *item* to a float
  - Type conversion only works if item is valid numeric value, otherwise, throws exception

» Python2orPython3

FrontPage
RecentChanges
FindPage
HelpContents
Python2orPython3
Page
» Immutable Page
» Info
» Attachments
» More Actions:
User
» Login

Should I use Python 2 or Python 3 for my development activity?

You should use Python 3 going forward, and s of January 2020 Python 2 will be in EOL (End Of Life) status and receive no further official support. What's more, a large number of open source projects using Python are taking the same approach.

This page is largely out of date, but kept for historical reference as it provides value for maintaining legacy Python 2 systems.

Contents

What are the differences?

Short version: Python 2.x is legacy, Python 3.x is the present and future of the language

Python 3.0 was released in 2008. The final 2.x version 2.7 release came out in mid-2010, with a statement of extended support for this end-of-life release. The 2.x branch will see no new major releases after that. 3.x is under active development and has already seen over five years of stable

2to3 - Automated Python 2 to 3 code translation

2to3 is a Python program that reads Python 2.x source code and applies a series of *fixers* to transform it into valid Python 3.x code. The standard library contains a rich set of fixers that will handle almost all code. 2to3 supporting library `lib2to3` is, however, a flexible and generic library, so it is possible to write your own fixers for 2to3. `lib2to3` could also be adapted to custom applications in which Python code needs to be edited automatically.

Using 2to3

2to3 will usually be installed with the Python interpreter as a script. It is also located in the `Tools/scripts` directory of the Python root.

2to3's basic arguments are a list of files or directories to transform. The directories are recursively traversed for Python sources.

Here is a sample Python 2.x source file, `example.py`:

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

It can be converted to Python 3.x code via 2to3 on the command line:

Getting Started

- The standard installation includes
  - A Python Shell
  - An IDE called **IDLE** (see next slide)
  - 2 very convenient programs called, **easy_install,** and **pip**, which you can use to install the python packages you need.

Python IDE: IDLE

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

svm.py - C:\Users\Chris\Desktop\SMM-code\svm.py (3.6.0)

File Edit Format Run Options Window Help

```
from sklearn import svm
from sklearn import datasets

classifier = svm.SVC(gamma='auto')
iris = datasets.load_iris()

# print iris
X, y = iris.data, iris.target
print(X)
print(y)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20)
# classifier.fit(X_train, y_train)
# y_pred = classifier.predict(X_test)

# MLP
from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(hidden_layer_sizes=(10, 10, 10), max_iter=1000)
mlp.fit(X_train, y_train)
y_pred = mlp.predict(X_test)


from sklearn.metrics import accuracy_score
print(accuracy_score(y_pred, y_test))
print(accuracy_score(y_pred, y_test, normalize=False))

from sklearn.metrics import classification_report, confusion_matrix
```

Ln: 11 Col: 0

Python IDE: IDLE

- **IDLE** is included in the standard installation:

C:\[your python directory]\Lib\idlelib\idle.bat

Easy_Install

- **easy_install** is a package manager for the Python programming language that provides a standard format for distributing Python programs and libraries.
- It looks Python Package Index (PyPI):

  http://pypi.python.org/pypi

  for the desired packages and uses the metadata there to download and install the package and its dependencies.
- It allows you to effortlessly install Python packages, instead of downloading, building, and installing them from source:
  - easy_install networkx
  - easy_install twitter
  - easy_install nltk …

PIP

- **pip** is also a package manager for the Python programming language that provides a standard format for distributing Python programs and libraries.
- It looks Python Package Index (PyPI):

  http://pypi.python.org/pypi

  for the desired packages and uses the metadata there to download and install the package and its dependencies.
- It allows you to effortlessly install Python packages, instead of downloading, building, and installing them from source:
  - pip install networkx
  - pip install twitter
  - pip install nltk …

Python IDE: IDLE

- **IDLE** & **PIP** are also included in the standard installation:

C:\[your python directory]\Scripts\easy_install.exe

C:\[your python directory]\Scripts\pip.exe

pip vs easy_install

easy_install was released in 2004, as part of setuptools. It was notable at the time for installing packages from PyPI using requirement specifiers, and automatically installing dependencies.

pip came later in 2008, as alternative to *easy_install*, although still largely built on top of setuptools components. It was notable at the time for *not* installing packages as Eggs or from Eggs (but rather simply as 'flat' packages from sdists), and introducing the idea of Requirements Files, which gave users the power to easily replicate environments.

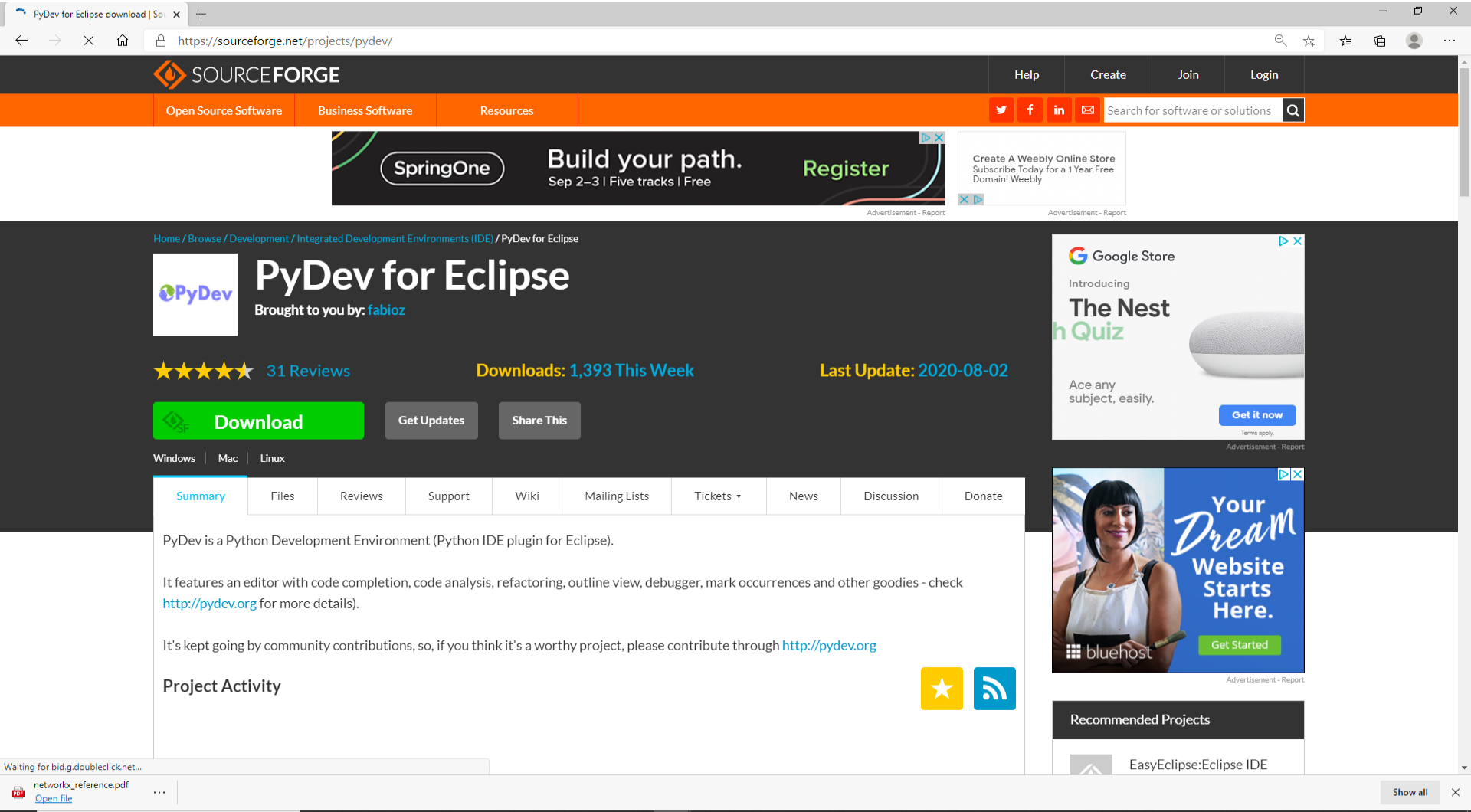
Here's a breakdown of the important differences between pip and easy_install now:

	pip	**easy_install**
Installs from Wheels	Yes	No
Uninstall Packages	Yes (`pip uninstall`)	No
Dependency Overrides	Yes (Requirements Files)	No
List Installed Packages	Yes (`pip list` and `pip freeze`)	No
PEP 438 Support	Yes	No
Installation format	'Flat' packages with *egg-info* metadata.	Encapsulated Egg format
sys.path modification	No	Yes
Installs from Eggs	No	Yes
pylauncher support	No	Yes [1]
Multi-version installs	No	Yes
Exclude scripts during install	No	Yes
per project index	Only in virtualenv	Yes, via setup.cfg

PyDev

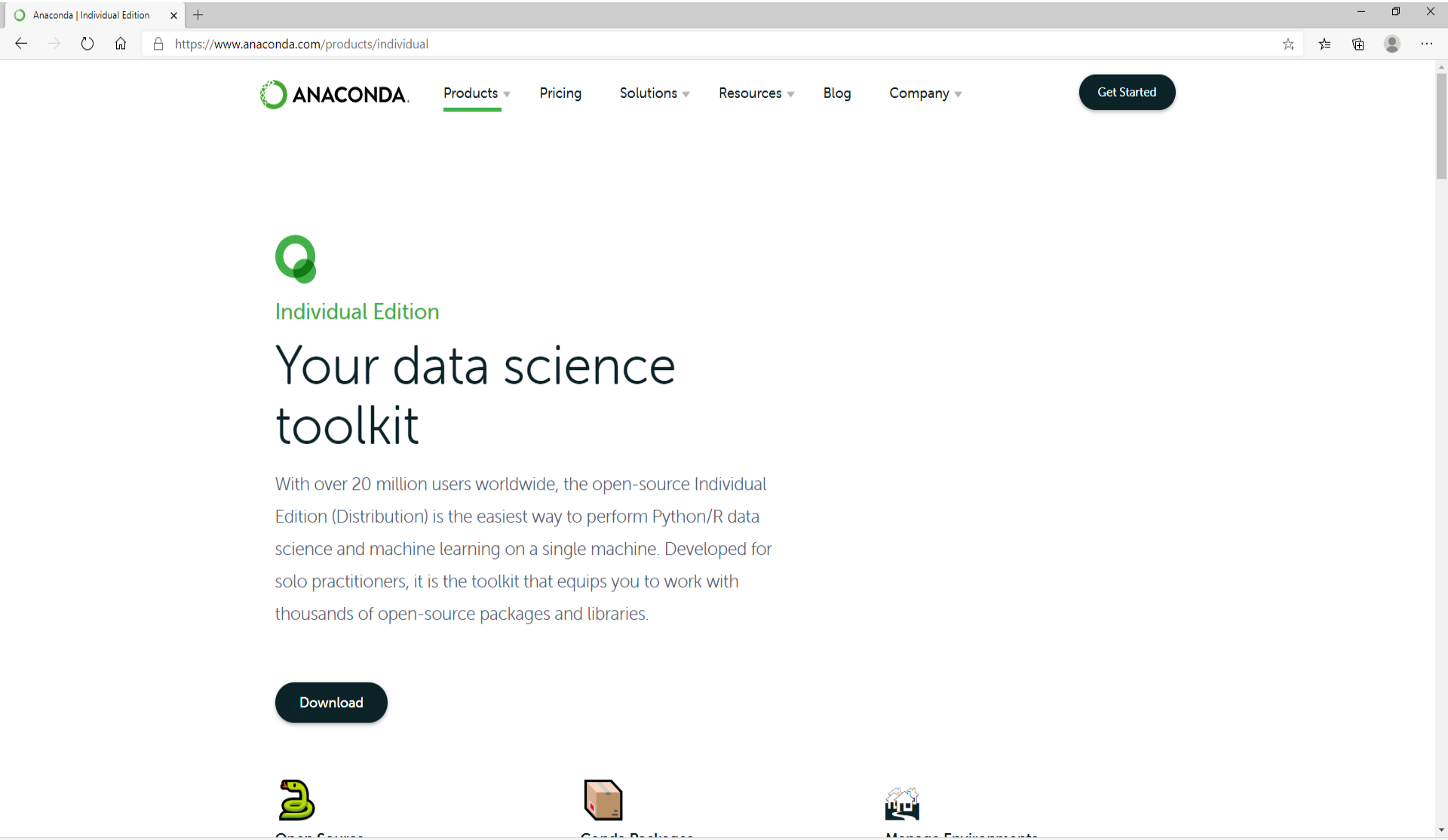
- Eclipse users can use PyDev for Eclipse:

http://sourceforge.net/projects/pydev/



Get Superpowers with Anaconda

- https://www.anaconda.com/download/



Enough to Understand the Code

- Assignment uses = and comparison uses ==
- For numbers + - * / % are as expected
  - Special use of + for string concatenation
  - Special use of **%** for string formatting (as with **printf** in C):

```
print("%i, %i, %i" % (1, 2, 3))
```

- Logical operators are words (**and**, **or**, **not**) not symbols
- The basic printing command is **print**
- The first assignment to a variable creates it
  - Variable types don't need to be declared
  - Python figures out the variable types on its own

The Exponent & Remainder Operators

- Remainder operator (**%**): Performs division and returns the remainder
  - a.k.a. modulus operator
  - e.g., 4%2=0, 5%2=1

- Exponent operator (**): Raises a number to a power

  x ** y = x^y

Variable Naming Rules

- Rules for naming variables in Python:
  - Variable name cannot be a Python keyword (see next slide)
  - Variable name cannot contain spaces
  - First character must be a letter or an underscore
  - After first character may use letters, digits, or underscores
  - Variable names are case sensitive

- Variable name should reflect its use

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'la
mbda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
>>>
```

Hello, World

- C:

```
#inlcude <stdio.h>
int main(int argc, char** argv[]) {
    printf("Hello, World!\n");
}
```

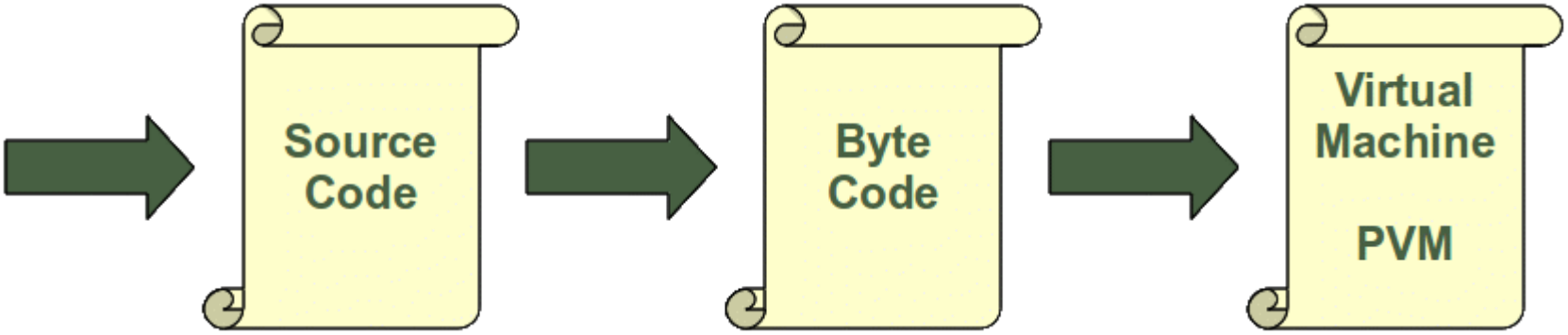
- Java:

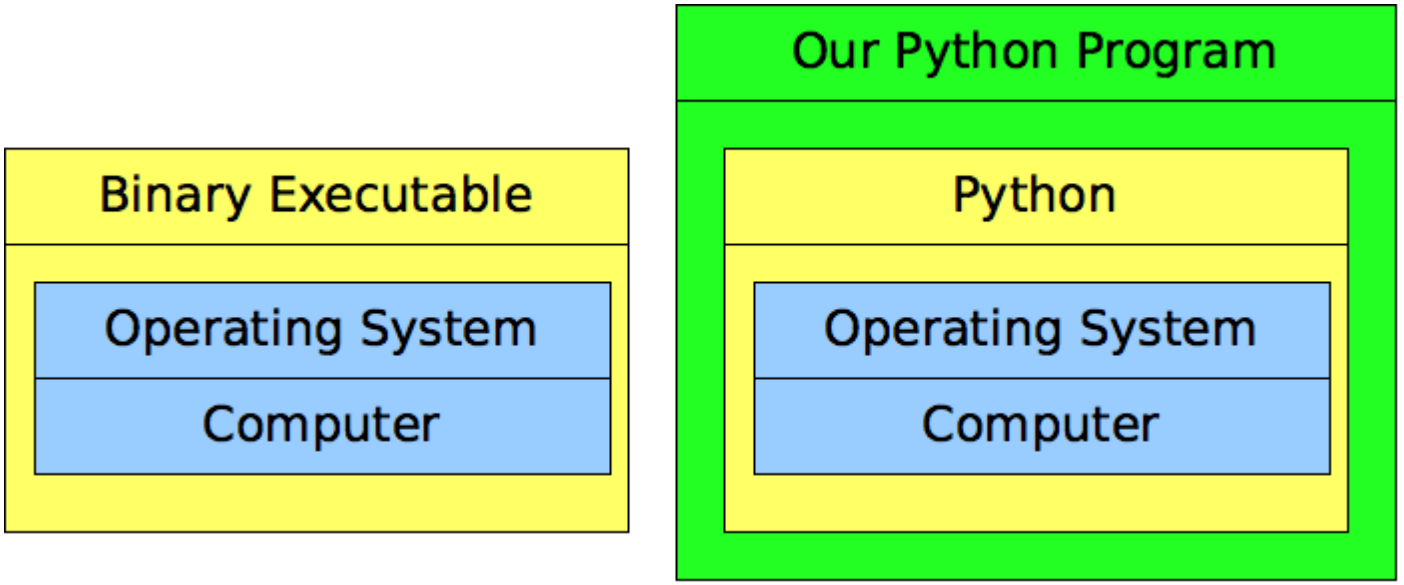
```
public class Hello {
    public static void main(String argv[]) {
        System.out.println("Hello, World!\n");
    }
}
```

- Python:

```
print("Hello, World!")
```

How Python Runs Programs





WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

py2exe

From Wikipedia, the free encyclopedia

py2exe is a Python extension which converts Python scripts (.py) into Microsoft Windows executables (.exe). These executables can run on a system without Python installed.[4] It is the most common tool for doing so.

py2exe was used to distribute the official BitTorrent client (prior to version 6.0) and is still used to distribute SpamBayes as well as other projects.

Since May 2014, there is a version of py2exe available for Python 3.[1] Before then, py2exe was made only for Python 2, and it was necessary to use an alternative like cx_Freeze for Python 3 code.

Although this program transforms a .py file to an .exe, it does not make it run faster as py2exe just bundles the Python bytecode rather than converting it to machine-code. It may even run slower than using the Python interpreter directly because of startup overhead.

py2exe

Stable release	0.9.2.2 for Python 3 / 21 October 2014[1] 0.6.9 for Python 2 / 15 November 2008[2]
Repository	sourceforge.net/p/py2exe/svn/HEAD/tree/
Written in	Python
Platform	Microsoft Windows
License	MIT[3]
Website	www.py2exe.org

References [edit]

1. ^ a b "py2exe 0.9.2.2 : Python Package Index". Retrieved 1 April 2015.
2. ^ "News". *py2exe.org*. Retrieved 2 February 2014.
3. ^ "Licence". *sourceforge.net May 2014*.
4. ^ "FrontPage". *py2exe.org*. Retrieved 2 February 2014.

External links [edit]

- Official website
- py2exe on SourceForge.net

py2exe 0.9.2.2

```
pip install py2exe
```

Latest version

Last released: Oct 21, 2014

Build standalone executables for Windows (python 3 version)

Navigation

Project description

Release history

Download files

Project description

py2exe is a distutils extension which allows to build standalone Windows executable programs (32-bit and 64-bit) from Python scripts; Python 3.3 and later are supported. It can build console executables, windows (GUI) executables, windows services, and DLL/EXE COM servers.

py2exe for Python 2 is still available at http://sourceforge.net/project/showfiles.php?group_id=15583.

Contents

How You Run Programs

- Python Interactive Shell:

  C:\Users\Edmund Yu>**python**

  Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900 32 bit (Intel)] on win32

  Type "help", "copyright", "credits" or "license" for more information.

  >>> print(“Hello World!”)

  Hello World!

  >>> print(2 ** 8)

  256

- Save the statements as a text file with a .py suffix (eg. hello.py), and then

  python hello.py

- Use IDLE: edit, save, run, debug…

Whitespace

- Whitespace is meaningful in Python: especially indentation and placement of newlines
- Use a newline to end a line of code
- Use \ when you must go to next line prematurely
- No braces { } to mark blocks of code in Python
  - Use consistent indentation instead.
  - The first line with more indentation starts a nested block.
  - The first line with less indentation is outside of the block.

```
# whitespace.py
for i in range(0, 10):
    for j in range(0, 10):
        print i, j
print 'done'
```

Basic Data Types

- **Integer**

```
>>> 2+2
4
>>> # This is a comment
...     2+2
4
>>> 2+2 # a comment on the same line
```

- **Floating point numbers**

```
>>> 3 * 3.75 / 1.5
7.5
```

- **Complex numbers**

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
```

Numeric Types — `int`, `float`, `complex`

There are three distinct numeric types: *integers*, *floating point numbers*, and *complex numbers*. In addition, Booleans are a subtype of integers. Integers have unlimited precision. Floating point numbers are usually implemented using `double` in C; information about the precision and internal representation of floating point numbers for the machine on which your program is running is available in `sys.float_info`. Complex numbers have a real and imaginary part, which are each a floating point number. To extract these parts from a complex number z, use `z.real` and `z.imag`. (The standard library includes additional numeric types, `fractions` that hold rationals, and `decimal` that hold floating-point numbers with user-definable precision.)

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including hex, octal and binary numbers) yield integers. Numeric literals containing a decimal point or an exponent sign yield floating point numbers. Appending `'j'` or `'J'` to a numeric literal yields an imaginary number (a complex number with a zero real part) which you can add to an integer or float to get a complex number with real and imaginary parts.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the “narrower” type is widened to that of the other, where integer is narrower than floating point, which is narrower than complex. Comparisons between numbers of mixed type use the same rule. [2] The constructors `int()`, `float()`, and `complex()` can be used to produce numbers of a specific type.

All numeric types (except complex) support the following operations, sorted by ascending priority (all numeric operations have a higher priority than comparison operations):

Operation	Result	Notes	Full documentation
`x + y`	sum of *x* and *y*		
`x - y`	difference of *x* and *y*		
`x * y`	product of *x* and *y*		
`x / y`	quotient of *x* and *y*		

Assignments

- Assignment:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
>>> x = y = z = 0 # Zero x, y and z
>>> x
0
>>> y
0
>>> z
0
```

- Multiple assignment:

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

Strings

```
>>> 'spam eggs'
'spam eggs'

>>> 'doesn't'
SyntaxError: invalid syntax

>>> 'doesn\'t'
"doesn't"

>>> '"Yes," he said.'
'"Yes," he said.'

>>> ""Yes, " he said"
SyntaxError: invalid syntax

>>> "\"Yes,\" he said."
'"Yes," he said.'
```

- Strings are immutable

- The built-in function **len()** returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Strings

❖ Concatenation

```
>>> word = 'Help' + ’A’
>>> word
'HelpA'

>>> '< ' + word*5 + ' > '
'<HelpAHelpAHelpAHelpAHelpA>'

>>> 'str' 'ing' # This is ok
'string'
>>> 'str'.strip() + 'ing' # This is ok
'string'
>>> ’str’.strip() 'ing' # <- This is invalid
File "<stdin>", line 1, in ?
’str’.strip() ’ing’
        ^
```

❖ Slicing

```
>>>  word[4]
’A’
>>>  word[0:2]
’He’
>>>  word[:2]
’He’
>>>  word[2:]
’lpA’
>>> word[-1] # The last character
’A’
>>> word[-2] # The last-but-one
’p’
```

❖ There is no separate <u>char</u> class

Unicode Strings

- They can be used to store and manipulate Unicode data (http://www.unicode.org/)
- Creating Unicode strings in Python is just as simple as creating normal strings:

  >>> **u'Hello World !'**

  u'Hello World !'

  The small 'u' in front of the quote indicates that a Unicode string is supposed to be created.
- If you want to include special characters in the string, you can do so by using the Python Unicode-Escape encoding:

  >>> **u'Hello\u0020World !'**

  u'Hello World !'
- You will see lots of Unicode data when harvesting data from Twitter

Table of Contents

Unicode HOWTO

Release:	1.12

This HOWTO discusses Python support for Unicode, and explains various problems that people commonly encounter when trying to work with Unicode.

Introduction to Unicode

History of Character Codes

In 1968, the American Standard Code for Information Interchange, better known by its acronym ASCII, was standardized. ASCII defined numeric codes for various characters, with the numeric values running from 0 to 127. For example, the lowercase letter 'a' is assigned 97 as its code value.

ASCII was an American-developed standard, so it only defined unaccented characters. There was an 'e', but no 'é' or 'Í'. This meant that languages which required accented characters couldn't be faithfully represented in ASCII. (Actually the missing accents matter for English, too, which contains words such as 'naïve' and 'café', and some publications have house styles which require spellings such as 'coöperate'.)

For a while people just wrote programs that didn't display accents. In the mid-1980s an Apple II BASIC program written by a French speaker might have lines like these:

```
PRINT "MISE A JOUR TERMINEE"
PRINT "PARAMETRES ENREGISTRES"
```

Unicode® character tat

unicode-table.com/en/#control-character

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0010	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0020		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0030	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0050	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0070	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
0080	XXX	XXX	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
0090	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	XXX	SCI	CSI	ST	OSC	PM	APC
00A0		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
00B0	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿

Ad closed by Google

Stop seeing this ad

Why this ad?

Control character

Open in separate page

Range: 0000— 001F

N

Click to highlight range

Follow us

Made by SA·Design

Lists

- Are comma-separated values (items) between **square** brackets
- List items need not all have the same type

>>> **a = [‘spam’, ‘eggs’, 100, 1234]**

>>> **a**

[‘spam’, ‘eggs’, 100, 1234]

- Slicing

>>> **a[0]**

‘spam’

>>> **a[3]**

1234

>>> **a[-2]**

100

>>> **a[1:-1]**

[‘eggs’, 100]

>>> **a[:2] + [‘bacon’, 2*2]**

[‘spam’, ‘eggs’, ‘bacon’, 4]

>>> **3*a[:3] + [‘Boo!’]**

[‘spam’, ‘eggs’, 100, ‘spam’, ‘eggs’, 100,
‘spam’, ‘eggs’, 100, ‘Boo!’]

Lists

❖ Lists, unlike strings, are mutable

```
>>> a
['spam', 'eggs', 100, 1234]

>>> a[2] = a[2] + 23
>>> a
[‘spam’, ‘eggs’, 123, 1234]
```

❖ The built-in function **len()** also applies to lists:

```
>>> a = [‘a’, ‘b’, ‘c’, ‘d’]
>>> len(a)
4
```

❖ Assignment

```
>>> # Replace some items:
...     a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
...     a[0:2] = []
>>> a
[123, 1234]
>>> # Insert some:
...     a[1:1] = [‘abc’, ‘xyz’]
>>> a
[123, ‘abc’, ‘xyz’, 1234]
```

Lists

- Methods associated with lists:
  - **append(x):** Add an item to the end of the list.
  - **extend(L):** Extend the list by appending all the items in the given list
  - **insert(i, x):** Insert an item at a given position
  - **remove(x):** Remove the first item from the list whose value is x.
  - **pop(i) or pop():** Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list.
  - **index(x):** Return the index in the list of the first item whose value is x.
  - **count(x):** Return the number of times x appears in the list.
  - **sort():** Sort the items of the list, in place. **Note: I'll discuss sorting in more detail later.**
  - **reverse():** Reverse the elements of the list

5. Data Structures

https://docs.python.org/3.8/tutorial/datastructures.html?highlight=data%20structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**(*x*)
Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**(*iterable*)
Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`.

`list.`**`insert`**(*i*, *x*)
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**`remove`**(*x*)
Remove the first item from the list whose value is equal to *x*. It raises a `ValueError` if there is no such item.

`list.`**`pop`**([*i*])
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

`list.`**`clear`**()
Remove all items from the list. Equivalent to `del a[:]`.

`list.`**`index`**(*x*[, *start*[, *end*]])
Return zero-based index in the list of the first item whose value is equal to *x*. Raises a `ValueError` if there is no such item.

The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

Sorting HOW TO

Author: Andrew Dalke and Raymond Hettinger
Release: 0.1

Python lists have a built-in `list.sort()` method that modifies the list in-place. There is also a `sorted()` built-in function that builds a new sorted list from an iterable.

In this document, we explore the various techniques for sorting data using Python.

Sorting Basics

A simple ascending sort is very easy: just call the `sorted()` function. It returns a new sorted list:

```
>>> sorted([5, 2, 3, 1, 4])
[1, 2, 3, 4, 5]
```

You can also use the `list.sort()` method. It modifies the list in-place (and returns `None` to avoid confusion). Usually it's less convenient than `sorted()` - but if you don't need the original list, it's slightly more efficient.

```
>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> a
[1, 2, 3, 4, 5]
```

Another difference is that the `list.sort()` method is only defined for lists. In contrast, the `sorted()` function accepts any iterable.

```
>>> sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})
[1, 2, 3, 4, 5]
```

Key Functions

Both `list.sort()` and `sorted()` have a *key* parameter to specify a function to be called on each list element prior to making comparisons.

For example, here's a case-insensitive string comparison:

String to List to String

- **join()** turns a list of strings into one string:
  <separator_string>.join(<some_list>)

```
>>> ";".join( ["abc", "def", "ghi"] )
```

“abc;def;ghi”

- **split()** turns one string into a list of strings.
  <some_string>.split(<separator_string>)

```
>>> "abc;def;ghi".split(";")
```

[“abc”, “def”, “ghi”]

- Note the **inversion** in the syntax

Convert Anything to a String

- The built-in **str()** function can convert an instance of any data type into a string:

```
>>> "Hello□" + str(2)
'Hello 2'

>>> str([1,2])
'[1, 2]'

>>> str(["abc", "def", "ghi"])
"['abc', 'def', 'ghi']"
```

Tuples

- Lists and strings are two examples of **sequence** data types, the third one being: tuples

- A tuple consists of a number of values separated by commas:

```
>>> t = 12345, 54321, “hello!”
>>> t[0]
12345
>>> t
(12345, 54321, ‘hello!’)
>>> # Tuples may be nested:
…    u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, ‘hello!’), (1, 2, 3, 4, 5))
```

Tuples

```
>>> # Tuples are immutable:
>>> t[0] = 88888
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> # but they can contain mutable objects:
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Multiple Assignment with Sequences

- Regular multiple assignment:

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

- It can applied to sequences as well, but the type and "shape" have to match:

```
>>> (x, y, (w, z)) = (2, 3, (4, 5))
>>> [x, y] = [4, 5]
```

Dictionaries

- Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by **keys**
- Keys can be any immutable type (e.g. strings and numbers):

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
```

Dictionaries

- The **keys()** method of a dictionary object returns a list of all the keys used in the dictionary, in arbitrary order

```
>>> tel.keys()
['jack', 'irv', 'guido']
```

- Similarly, the **values()** method of a dictionary object returns a list of all the values used in the dictionary, in arbitrary order

```
>>> tel.values()
[4098, 4127, 4127]
```

Dictionaries

- To check whether a single key is in the dictionary, use the **in** keyword.

  >>> ‘guido’ **in** tel

  True

- The **dict()** constructor builds dictionaries directly from lists of key-value pairs stored as tuples

  >>> **dict([(‘sape’, 4139), (‘guido’, 4127), (‘jack’, 4098)])**

  {‘sape’: 4139, ‘jack’: 4098, ‘guido’: 4127}

Mapping Types — dict

A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the *dictionary*. (For other containers see the built-in `list`, `set`, and `tuple` classes, and the `collections` module.)

A dictionary's keys are *almost* arbitrary values. Values that are not hashable, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (such as `1` and `1.0`) then they can be used interchangeably to index the same dictionary entry. (Note however, that since computers store floating-point numbers as approximations it is usually unwise to use them as dictionary keys.)

Dictionaries can be created by placing a comma-separated list of `key: value` pairs within braces, for example: `{'jack': 4098, 'sjoerd': 4127}` or `{4098: 'jack', 4127: 'sjoerd'}`, or by the `dict` constructor.

class **dict**(***kwarg*)
class **dict**(*mapping*, ***kwarg*)
class **dict**(*iterable*, ***kwarg*)

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it is a mapping object, a dictionary is created with the same key-value pairs as the mapping object. Otherwise, the positional argument must be an iterable object. Each item in the iterable must itself be an iterable with exactly two objects. The first object of each item becomes a key in the new dictionary, and the second object the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created

Sets

- A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.
- Curly braces or the **set()** function can be used to create sets. Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary, a data structure that we will discuss next.

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket) # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket # fast membership testing
True
>>> 'crabgrass' in basket
False
```

Sets

```
>>> # Demonstrate set operations on unique letters from two words
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b # letters in a but not in b – DIFFERENCE
{'r', 'd', 'b'}
>>> a | b # letters in a or b or both – OR/UNION
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b # letters in both a and b – AND/INTERSECTION
{'a', 'c'}
>>> a ^ b # letters in a or b but not both – Exclusive OR
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Files

- Open a file for reading or writing:

  >>> f = open('workfile', ‘w’) # also ‘r’, ‘r+’ ‘a’ & ‘b’

- Methods of File objects:
  - f.**read**() reads entire file into a string
  - f.**read**(*size*) reads *size* bytes of data into a string
  - f.**readline**() reads a single line from the file
  - f.**readlines**(*size*) returns a list containing all the lines of data in the file. If given an optional parameter *size*, it reads that many bytes from the file and enough more to complete a line, and returns the lines from that
  - f.**write**(string) writes the contents of string to the file, returning None
  - f.**tell**() returns an integer giving its current position (byte offset)
  - f.**seek**(offset, from_what) changes the file object’s position. ‘from_what’ is the reference point: 0 (default) means from the beginning of the file; 1 means the current file position; and 2 means the end of the file
  - f**.close**() closes the file

7.2. Reading and Writing Files

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

```
>>> f = open('workfile', 'w')
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. *mode* can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The *mode* argument is optional; `'r'` will be assumed if it's omitted.

Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If encoding is not specified, the default is platform dependent (see `open()`). `'b'` appended to the mode opens the file in *binary mode*: now the data is read and written in the form of bytes objects. This mode should be used for all files that don't contain text.

In text mode, the default when reading is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing in text mode, the default is to convert occurrences of `\n` back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in `JPEG` or `EXE` files. Be very careful to use binary mode when reading and writing such files.

It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using `with` is also much shorter than writing equivalent `try-finally` blocks:

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

Data Type Categories

Data Type	Category	Mutable?
Numbers	Numeric	No
Strings	Sequence	No
Lists	Sequence	Yes
Tuples	Sequence	No
Dictionaries	Mapping	Yes
Files	Extension	N/A

The pickle Module (Serialization)

- The pickle module saves programmers' time and energy in dealing with complicated data types.
  - It takes a Python object and converts it to a **string** representation; this process is called pickling.

- Reconstructing the object from the string representation is called unpickling.

```
>>> import pickle
>>> pickle.dump(x, f)    # pickle
>>> x = pickle.load(f)    # unpickle
```

The pickle Module, continued

- Python objects can be stored and reused by other programs or by a future invocation of the same program. (persistent objects).

- Because pickle is so widely used, many authors who write Python extensions take care to ensure that new data types such as matrices can be properly pickled and unpickled.

- **cPickle** is a faster version, renamed as **_pickle** in Python 3.

Pickling: An Example

```
tel = dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])

import pickle
f = open('dump.pickle', 'wb')
pickle.dump(tel, f)
f.close()

f = open('dump.pickle', 'rb')
x = pickle.load(f)
f.close()

print(x)
```

pickle — Python object serialization

Source code: Lib/pickle.py

The pickle module implements binary protocols for serializing and de-serializing a Python object structure. *"Pickling"* is the process whereby a Python object hierarchy is converted into a byte stream, and *"unpickling"* is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as "serialization", "marshalling," [1] or "flattening"; however, to avoid confusion, the terms used here are "pickling" and "unpickling".

Warning: The pickle module **is not secure**. Only unpickle data you trust.

It is possible to construct malicious pickle data which will **execute arbitrary code during unpickling**. Never unpickle data that could have come from an untrusted source, or that could have been tampered with.

Consider signing data with hmac if you need to ensure that it has not been tampered with.

Safer serialization formats such as json may be more appropriate if you are processing untrusted data. See Comparison with json.

Relationship to other Python modules

Comparison with marshal

Python has a more primitive serialization module called marshal, but in general pickle should always be the preferred way to serialize Python objects. marshal exists primarily to support Python's .pyc files.

The pickle module differs from marshal in several significant ways:

- The pickle module keeps track of the objects it has already serialized, so that later references to the same object won't be serialized again. marshal doesn't do this.

Saving Structured Data with json

- Python also allows you to use the popular data interchange format called JSON (JavaScript Object Notation).
- The standard module called **json** can take Python data hierarchies, and convert them to string representations;
- The JSON format is commonly used by modern applications to allow for data exchange. Many programmers are already familiar with it, which makes it a good choice for interoperability.

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


'[1, "simple", "list"]'
```

Saving Structured Data with json

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

```
x = '[1, "simple", "list"]'
json.dump(x, f)
```

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

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

- This simple serialization technique can handle lists and dictionaries, but serializing arbitrary class instances in JSON requires a bit of extra effort. The reference for the json module contains an explanation of this.

json — JSON encoder and decoder

Source code: Lib/json/__init__.py

JSON (JavaScript Object Notation), specified by **RFC 7159** (which obsoletes **RFC 4627**) and by ECMA-404, is a lightweight data interchange format inspired by JavaScript object literal syntax (although it is not a strict subset of JavaScript [1]).

json exposes an API familiar to users of the standard library marshal and pickle modules.

Encoding basic Python object hierarchies:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

Compact encoding:

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

str() vs. json.dumps(), Case 1

```
import json

tel = {'sape':4139, 'guido':4127, 'jack':4098}

print(json.dumps(tel))
```

Output: **{"sape": 4139, "guido": 4127, "jack": 4098}**

```
print(str(tel))
```

Output: **{'sape': 4139, 'guido': 4127, 'jack': 4098}**

str() vs. json.dumps(), Case 2

```
import json

tel = {"sape":4139, "guido":4127, "jack":4098}

print(json.dumps(tel))
```

Output: **{"sape": 4139, "guido": 4127, "jack": 4098}**

```
print(str(tel))
```

Output: **{'sape': 4139, 'guido': 4127, 'jack': 4098}**

str() vs. json.dumps(), More Cases

	Output:
print(json.dumps([1,2]))	[1, 2]
print(str([1,2]))	[1, 2]
print(json.dumps(True))	true
print(str(True))	True
print(json.dumps("This is a string."))	"This is a string."
print(str("This is a string."))	This is a string.

The if Statement

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

The while Loop

- The while Loop:
  - The while loop executes as long as the condition remains true.
  - In Python, like in C, any non-zero integer value is true; zero is false
  - The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false
  - The standard comparison operators are written the same as in C: < (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to) and != (not equal to)

```
>>> # Fibonacci series:
...     # the sum of two elements defines
...     # the next
>>> a, b = 0, 1
>>> while b < 10:
...         print (b)
...         a, b = b, a+b
...
1
1
2
3
5
8
```

The for Loop

❖ The **for** loop:

```
>>> a = [‘cat’, ‘window’, ‘defenestrate’]
>>> for x in a:
...         print(x, len(x))
...
cat 3
window 6
defenestrate 12
```

❖ The **range()** function

```
>>> for i in range(len(a)):
...         print(i, a[i])
...
0 cat
1 window
2 defenestrate
```

Ranges

https://docs.python.org/3/library/stdtypes.html?highlight=range#range

The range type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.

class **range**(*stop*)

class **range**(*start*, *stop*[, *step*])

The arguments to the range constructor must be integers (either built-in int or any object that implements the __index__ special method). If the *step* argument is omitted, it defaults to 1. If the *start* argument is omitted, it defaults to 0. If *step* is zero, ValueError is raised.

For a positive *step*, the contents of a range r are determined by the formula r[i] = start + step*i where i >= 0 and r[i] < stop.

For a negative *step*, the contents of the range are still determined by the formula r[i] = start + step*i, but the constraints are i >= 0 and r[i] > stop.

A range object will be empty if r[0] does not meet the value constraint. Ranges do support negative indices, but these are interpreted as indexing from the end of the sequence determined by the positive indices.

Ranges containing absolute values larger than sys.maxsize are permitted but some features (such as len()) may raise OverflowError.

Range examples:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
```

break

- **Break**: like in C, breaks out of the smallest enclosing for or while loop

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break

    else: # loop fell through
            print(n, 'is a prime number')
```

Output:

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

continue

- **Continue**: also borrowed from C, continues with the next iteration of the loop

```
for num in range(2, 10):
    if num % 2 == 0:
        print("Found an even number", num)
        continue
    print("Found a number", num)
```

Output:

```
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

pass

- The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action.

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

More Looping: iteritems()/items()

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

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

- **items()** replaces **iteritems()** in Python 3.

More Looping : enumerate()

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

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

More Looping: zip()

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

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

More Looping: sorted()

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

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

More Looping: sorted()

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

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

- Same as **sorted(basket, reverse=True)**

List Comprehensions

- List comprehensions provide a concise way of creating lists
- Python programmers use list comprehensions extensively. You'll see a lot of them in real code.
- Each list comprehension consists of an expression, which may contain a function, followed by a for clause, then zero or more for or if clauses:

  [expression for name in list]

  [expression for name in list if filter]

  [expression for name1 in list1 for name2 in list2]

- It creates a list, resulting from applying the function (or evaluating the expression) to every member of the original list, in the context of the for and if clauses.

List Comprehensions: Examples

❖ [expression for name in list]

>>> freshfruit = [‘□banana’, ‘□loganberry ’, ‘passion fruit ’]

>>> [weapon.**strip**() **for** weapon **in** freshfruit]

[‘banana’, ‘loganberry’, ‘passion fruit’]

>>> vec = [2, 4, 6]

>>> [3*x **for** x **in** vec]

[6, 12, 18]

List Comprehensions: Examples

❖ [expression for name in list if filter]

```
>>> [3*x for x in vec if x > 3]
[12, 18]
```

❖ [expression for name1 in list1 for name2 in list2]

```
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
```

Dictionary Comprehensions

- Just like a list comprehension, a dictionary comprehension iterates over a list of items and collects values (key/value pairs in this case) that are to be returned.
- Dictionary comprehensions provide a concise way of creating dictionaries

```
>>> { k : k*k for k in range(10) }
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

>>> freshfruit = [‘banana’, ‘loganberry’, ‘passion fruit’]
>>> {name : len(name) for name in freshfruit}
{'passion fruit': 13, 'banana': 6, 'loganberry': 10}

>>> {name : len(name) for name in freshfruit if len(name) > 10}
{'passion fruit': 13}
```

Defining Functions

No header file or declaration of types of function or arguments

Function definition begins with “def.”

Function name and its arguments.

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

Colon.

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

The keyword ‘return’ indicates the value to be sent back to the caller.

Defining Functions

An example:

```
def fib(n): # write Fibonacci series up to n
        a, b = 0, 1
        while a < n:
            print(a, end=' ')
            a, b = b, a+b

fib(2000)
```

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597

Defining Functions

- It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
f2000 = fib2(2000)          # call it
print(f2000)                # write the result
```

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597]

Defining Functions

- It is also possible to define functions with a variable number of arguments.

- There are three ways to do that, which can be combined:
  1. Default Argument Values
  2. Keyword Arguments
  3. Arbitrary Argument Lists

1. Default Argument Values

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
        while True:
                ok = input(prompt)
                if ok in ('y', 'ye', 'yes'):
                        return True
                if ok in ('n', 'no', 'nop', 'nope'):
                        return False
                retries = retries - 1
                if retries < 0:
                        raise ValueError('invalid user response')
                print(reminder)
ask_ok('Do you really want to quit?')
ask_ok('OK to overwrite the file?', 2)
ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')
```

2.Keyword Arguments:

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

It could be called in any of the following ways:

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

2.Keyword Arguments:

- but all the following calls would be invalid:

```
parrot() # required argument missing
parrot(voltage=5.0, 'dead') # non-keyword argument after a
                                        # keyword argument
parrot(110, voltage=220) # duplicate value for the same
argument
parrot(actor='John Cleese') # unknown keyword argument
```

3. Arbitrary Argument Lists

- The least frequently used option is to specify that a function can be called with an arbitrary number of arguments.
- These arguments will be wrapped up in a tuple.
- Before the variable number of arguments, zero or more normal arguments may occur.

```
>>> def f(*args): print(args)
>>> f(1, 2)
(1, 2)
```

Similarly for keyword arguments, which will be in a **dictionary**:

```
>>> def f(**args): print(args)
>>> f(a=1, b=2)
{‘a’: 1, ‘b’: 2}
```

Lambda Expressions

- Small anonymous functions can be created with the lambda keyword. The following example uses a lambda expression to return a function.

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


f = make_incrementor(42)
print(f(0))
42
print(f(1))
43
```

Lambda Expressions

- Another use is to pass a small function as an argument.

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

print(pairs)
```

[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]

Returning Multiple Values

- In Python, a function can return multiple values:
  - Format: **return** *expression1,expression2, etc.*

```
def get_name(): # Get the user's first and last names.
        first = input('Enter your first name: ‘)
        last = input('Enter your last name: ‘)
        # Return both names.
        return first, last
```

  - When you call such a function in an assignment statement, you need a separate variable on the left side of the = operator to receive each returned value:

```
first_name, last_name = get_name()
```

Modules

- If you quit from the Python interpreter, the definitions you have made (functions and variables) are lost.
- If you want to write a longer program, you should save it in a file. This is known as creating a **script**.
- As your program gets longer and longer, you may want to split it into several files for easier maintenance.
- To support this, Python allows you to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a **module**.
- The file name should be the module name with a suffix .py

Modules

- For example, create a file called **fibo.py** in the current directory with the following contents:

```
# Fibonacci numbers module
def  fib(n): # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print a,
        a, b = b, a+b
def  fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Importing Modules (dot notation)

- Three ways to import

```
>>> import fibo
>>> fibo.fib(100)
1 1 2 3 5 8 13 21 34 55 89

>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

>>> fib = fibo.fib
>>> fib(100)
1 1 2 3 5 8 13 21 34 55 89
```

```
>>> from fibo import fib, fib2
>>> fib(100)
1 1 2 3 5 8 13 21 34 55 89

>>> from fibo import *
>>> fib(100)
1 1 2 3 5 8 13 21 34 55 89
```

Executing Modules As Scripts

- By adding this code at the end of your module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

- You can then make the file usable as a script (i.e. runnable) as well as an importable module:

```
python fibo.py 100
1 1 2 3 5 8 13 21 34 55 89
```

Exception Handling

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called **exceptions**. Most exceptions are not handled by programs:

```
>>> 10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> 4 + spam*3
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
File "<stdin>", line 1, in
<module>
TypeError: Can't convert 'int'
object to str implicitly
```

Exception Handling

❖ The try statement with one except clause:

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
```

Exception Handling

❖ The try statement with more than one except clauses:

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    except:
       print("unexpected exception")
```

Exception Handling

❖ The try statement with else:

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero !")
    except:
       print("unexpected exception")
    else:
        print("result is", result)
```

Exception Handling

- The try statement with finally cleans up the action: (try.py)

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero !")
     except:
        print("unexpected exception")
    else:
        print("result is", result)
    finally:
        print("executing finally clause“)

divide(2, 1)
divide(2, 0)
divide("2", "1")
```

Exception Handling

- To learn more about how to throw/raise exceptions and define your own ‘exception’ classes, please see Chapter 8 of the official python tutorial:

  https://docs.python.org/3/tutorial/index.html

Defining Classes

- Class Definition Syntax:

```
class ClassName:
    <Statement 1>
    .
    .
    .
    <Statement-N>
```

- When a class is defined, a **class object** is created. Class objects support two kinds of operations:

  attribute references and **instantiation**

Attribute References

- Attribute references use the standard syntax used for all attribute references in Python: **obj.name**
- Valid attribute names are all the names that were in the class's namespace when the class object was created
- So, if the class definition looked like this:

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

- **MyClass.i** and **MyClass.f** are valid attribute references, returning an integer and a function object, respectively.

Attribute References

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

print(MyClass.i)        # output is: 12345
print(MyClass.f)        # output is: <function MyClass.f at 0x0154A8A0>

x = MyClass()
print(x.i)              # output is: 12345
print(x.f())            # output is: hello world
```

Instantiation

- Instantiation: x = MyClass() creates a new instance of the class and assigns this object to the local variable x. No new keyword.
- The instantiation operation creates an empty object.
- __init__() serves as a constructor for the class. Usually does some initialization work. The arguments passed to the class are given to __init__() :

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

x = Complex(3.0, -4.5)
print(x)          # ouput is: <__main__.Complex object at 0x014B36D0>
print(x.r, x.i)   # output is: 3.0, -4.5
```

Self

- The first argument of every method is a reference to the current instance of the class
  - By convention, it's named as self

```
class  myClass:
    def  myFunc(self, name):
        self.name = name
```

- In __init__, self refers to the object currently being created. (See previous slide)
- In other methods, it refers to the instance whose method was called
  - Similar to the keyword this in Java or C++.
  - But Python uses self more often than Java uses this
  - You don't include it when calling it. Python passes it for you automatically.

Class Variables vs Instance Variable

```
class Dog:
        kind = 'canine' # class variable shared by all instances
        def __init__(self, name):
            self.name = name # instance variable unique to each instance


d = Dog('Fido')
e = Dog('Buddy')
print(d.kind) # shared by all dogs
print(e.kind) # shared by all dogs
print(d.name) # unique to d
print(e.name) # unique to e
```

```
class Dog:

    tricks = []             # mistaken use of a class variable

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

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

Inheritance

❖ Inheritance: the syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <Statement 1>
    …
    <Statement N>
```

❖ When the base class is defined in another module:

```
class DerivedClassName(ModuleName.BaseClassName):
    <Statement 1>…
```

Multiple Inheritance

- Multiple inheritance: Python supports a limited form of multiple inheritance. A class definition with multiple base classes looks like this:

```
Class DerivedClassName(Base1, Base2, Base3):
  <Statement 1>
  ...
  <Statement N>
```

Deleting Instances

- When you are done with an object, you don't have to delete or free it explicitly.
  - Python has **automatic garbage collection**.
  - Python will automatically detect when all of the references to a piece of memory have gone out of scope, and then automatically frees that memory.
  - Few memory leaks.
  - No "destructor" method for classes.

Packages

- Packages are a way of structuring Python's module namespace by using "dotted module names"
- The module name A.B designates a module named B in a package named A.
- Just like using modules saves the authors of different modules from having to worry about each other's global variable names,
- Using dotted module names saves the authors of multi-module packages from having to worry about each other's module names.

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Use of Packages (dot notation)

1. **import** sound.effects.echo

   This loads the module sound.effects.echo. It must be referenced with its full name:

   sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)

2. **from** sound.effects **import** echo

   This also loads the module echo, and makes it available without its package prefix, so it can be used as follows:

   echo.echofilter(input, output, delay=0.7, atten=4)

3. **from** sound.effects.echo **import** echofilter

   Again, this loads the module echo, but this makes its function echofilter() directly available:

   echofilter(input, output, delay=0.7, atten=4)

The Standard Library

- The **os** module provides dozens of functions for interacting with the operating system
- The **shutil** (shell utilities) module provides a higher level interface for file and directory management tasks
- The **sys** module provides the *argv* attribute for command line arguments. It also has attributes for *stdin*, *stdout*, and *stderr*.
- The **re** module provides regular expression tools for string processing
- The **math** module gives access to the underlying C library functions for floating point math
- The **random** module provides tools for making random selections

Generating Random Numbers

- Random number are useful in a lot of programming tasks
- **random** module includes library functions for working with random numbers
- Use **dot** notation: notation for calling a function belonging to a module
  - Format: module_name.function_name()

randint

- **randint** function: generates a random number in the range provided by the arguments

Figure 5-21 The `random` function returns a value

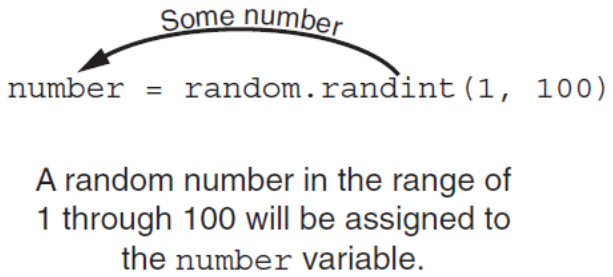
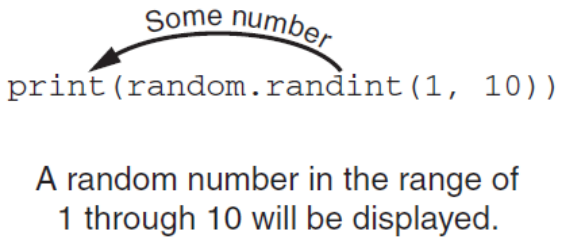


Figure 5-22 Displaying a random number



More Functions

- **randrange** function: similar to range function, but returns randomly selected integer from the resulting sequence
  - Same arguments as for the range function
- **random** function: returns a random float in the range of 0.0 and 1.0 (to be exact, **[0.0, 1.0)**)
  - Does not receive arguments
- **uniform** function: returns a random float but allows user to specify range

More Examples

```
import random

print(random.randrange(100))
print(random.randrange(1, 100, 2))

print(random.random())

print(random.uniform(1, 2))
print(random.uniform(1.5, 3.7))
```

`random` — Generate pseudo-random numbers

Source code: Lib/random.py

This module implements pseudo-random number generators for various distributions.

For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

Almost all module functions depend on the basic function `random()`, which generates a random float uniformly in the semi-open range [0.0, 1.0). Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The functions supplied by this module are actually bound methods of a hidden instance of the `random.Random` class. You can instantiate your own instances of `Random` to get generators that don't share state.

Class `Random` can also be subclassed if you want to use a different basic generator of your own devising: in that case, override the `random()`, `seed()`, `getstate()`, and `setstate()` methods. Optionally, a new generator can supply a `getrandbits()` method — this allows `randrange()` to produce selections over an arbitrarily large

The math Module

Table 5-2 Many of the functions in the `math` module

`math` Module Function	Description
`acos(x)`	Returns the arc cosine of `x`, in radians.
`asin(x)`	Returns the arc sine of `x`, in radians.
`atan(x)`	Returns the arc tangent of `x`, in radians.
`ceil(x)`	Returns the smallest integer that is greater than or equal to `x`.
`cos(x)`	Returns the cosine of `x` in radians.
`degrees(x)`	Assuming `x` is an angle in radians, the function returns the angle converted to degrees.
`exp(x)`	Returns e^x
`floor(x)`	Returns the largest integer that is less than or equal to `x`.
`hypot(x, y)`	Returns the length of a hypotenuse that extends from (0, 0) to (`x`, `y`).
`log(x)`	Returns the natural logarithm of `x`.
`log10(x)`	Returns the base-10 logarithm of `x`.
`radians(x)`	Assuming `x` is an angle in degrees, the function returns the angle converted to radians.
`sin(x)`	Returns the sine of `x` in radians.
`sqrt(x)`	Returns the square root of `x`.
`tan(x)`	Returns the tangent of `x` in radians.

The math Module

- The math module defines variables **pi** and **e** (natural exponential, next slide), which are assigned the mathematical values for *pi* and *e*
  - Can be used in equations that require these values, to get more accurate results
- Variables must also be called using the dot notation
  - Example:

```
import math
circle_area = math.pi * radius**2
```

What is a Natural Exponential Function?

exponent (variable)

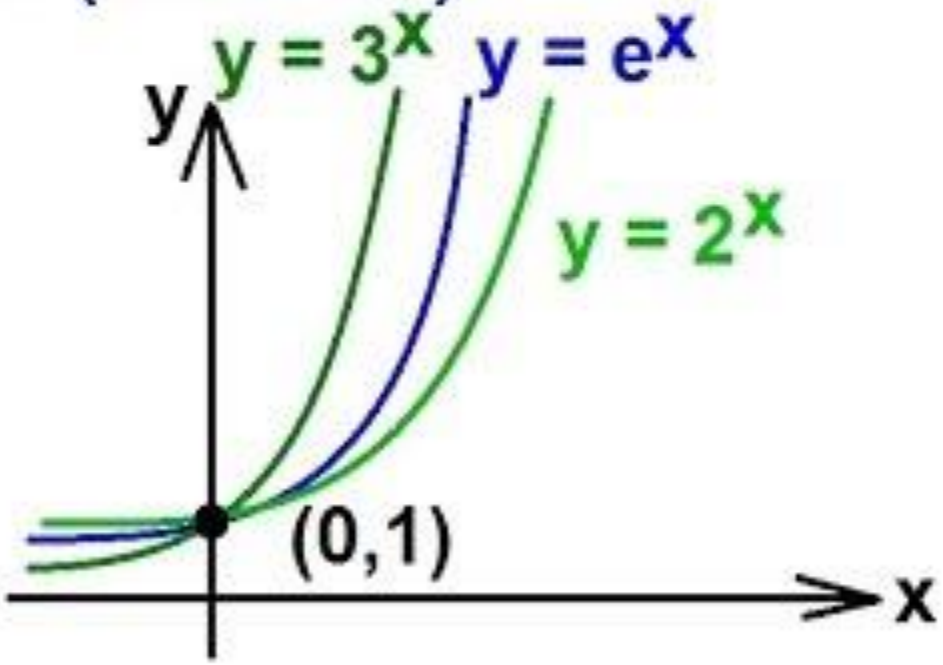
$$f(x) = e^{x}$$

base (constant)

$$e = ?$$

$$e = \lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^{n}$$

$$e = 2.71828\cdots$$



`math` — Mathematical functions

This module provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the `cmath` module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

Number-theoretic and representation functions

`math.`**`ceil`**(*x*)

Return the ceiling of *x*, the smallest integer greater than or equal to *x*. If *x* is not a float, delegates to `x.__ceil__()`, which should return an `Integral` value.

`math.`**`comb`**(*n*, *k*)

Return the number of ways to choose *k* items from *n* items without repetition and without order.

Evaluates to `n! / (k! * (n - k)!)` when `k <= n` and evaluates to zero when `k > n`.

Also called the binomial coefficient because it is equivalent to the coefficient of k-th term in polynomial ex

The Standard Library

- The **statistics** module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data.
- The **urllib.request** module is used for retrieving data from URLs. (The **requests** package is more popular, which uses urllib3.)
- The **smtplib** is used for sending mail
- The **datetime** module supplies classes for manipulating dates and times in both simple and complex ways.
- The **zlib**, **gzip**, **bz2**, **lzma**, **zipfile** and **tarfile** modules are used for data compression.
- The **timeit** module is used for measuring performance

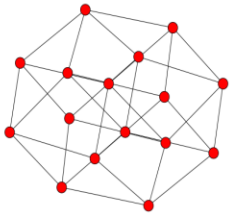
The Standard Library

- The **threading** module can run tasks in background while the main program continues to run:
- The **logging** module offers a full featured and flexible logging system.
- The **decimal** module offers a Decimal datatype for decimal floating point arithmetic.
- The **locale** module accesses a database of culture specific data formats.

- For more information, please see the official Python tutorial and related documentation.

Popular Python Packages

- **NumPy**, for high-performance and vectorized computations on multidimensional arrays: *http://www.numpy.org*
- **SciPy**, for advanced numerical algorithms: *http://www.scipy.org*
- **Matplotlib**, for plotting and interactive visualization: *http://matplotlib.org*
- **NetworkX**, for handling graphs: *http://networkx.lanl.gov*



- **Python Imaging Library** (PIL), for image-processing algorithms: *http://www.pythonware.com/products/pil* (python2, python3→pillow)
- **PySide/PyQt**, a wrapper around Qt for graphical user interfaces (GUIs): *http://qt-project.org/wiki/PySide* *http://www.riverbankcomputing.co.uk/software/pyqt/intro*
- **Cython**, for using C code in Python: *http://cython.org*
- **IPython**: *http://ipython.org*

View on GitHub

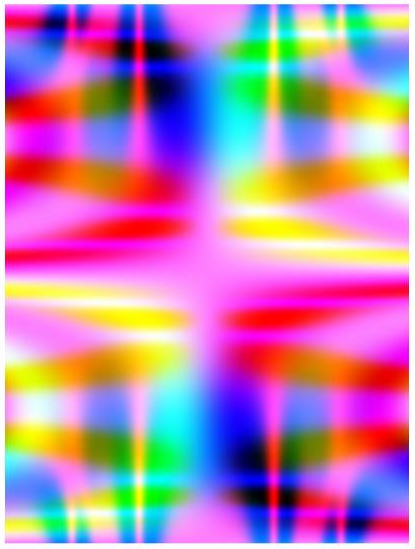
Pillow for enterprise

pillow

The friendly PIL fork

Welcome

This is the home of Pillow, the friendly PIL fork. PIL is the Python Imaging Library. If you have ever worried or wondered about the future of PIL, please stop. We're here to save the day.



Random psychedelic art made with PIL

Code

Our code is hosted on GitHub, tested on Travis CI, AppVeyor, Coveralls, Landscape and released on PyPI.

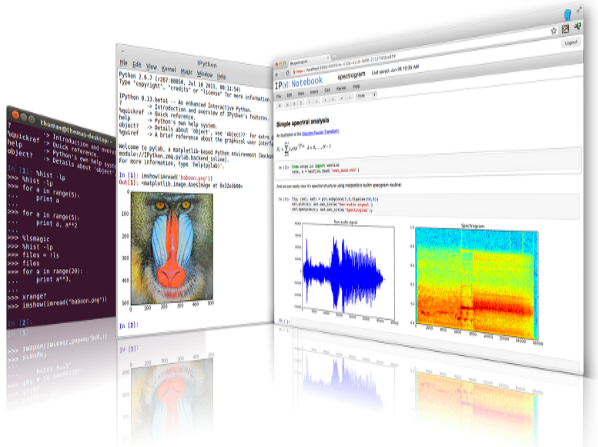
Documentation

Our documentation is hosted on

IP[y]: IPython
Interactive Computing

IPython provides a rich architecture for interactive computing with:

- A powerful interactive shell.
- A kernel for Jupyter.
- Support for interactive data visualization and use of GUI toolkits.
- Flexible, embeddable interpreters to load into your own projects.
- Easy to use, high performance tools for parallel computing.



To get started with IPython in the Jupyter Notebook, see our official example collection. Our notebook gallery is an excellent way to see the many things you can do with IPython while learning about a variety of topics, from basic programming to advanced statistics or quantum mechanics.

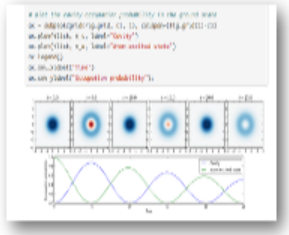
To learn more about IPython, you can download our talks and presentations, or read our extensive documentation. IPython is open source (BSD license), and is used by a range of other projects; add

JUPYTERCON

jupytercon

NOTEBOOK VIEWER

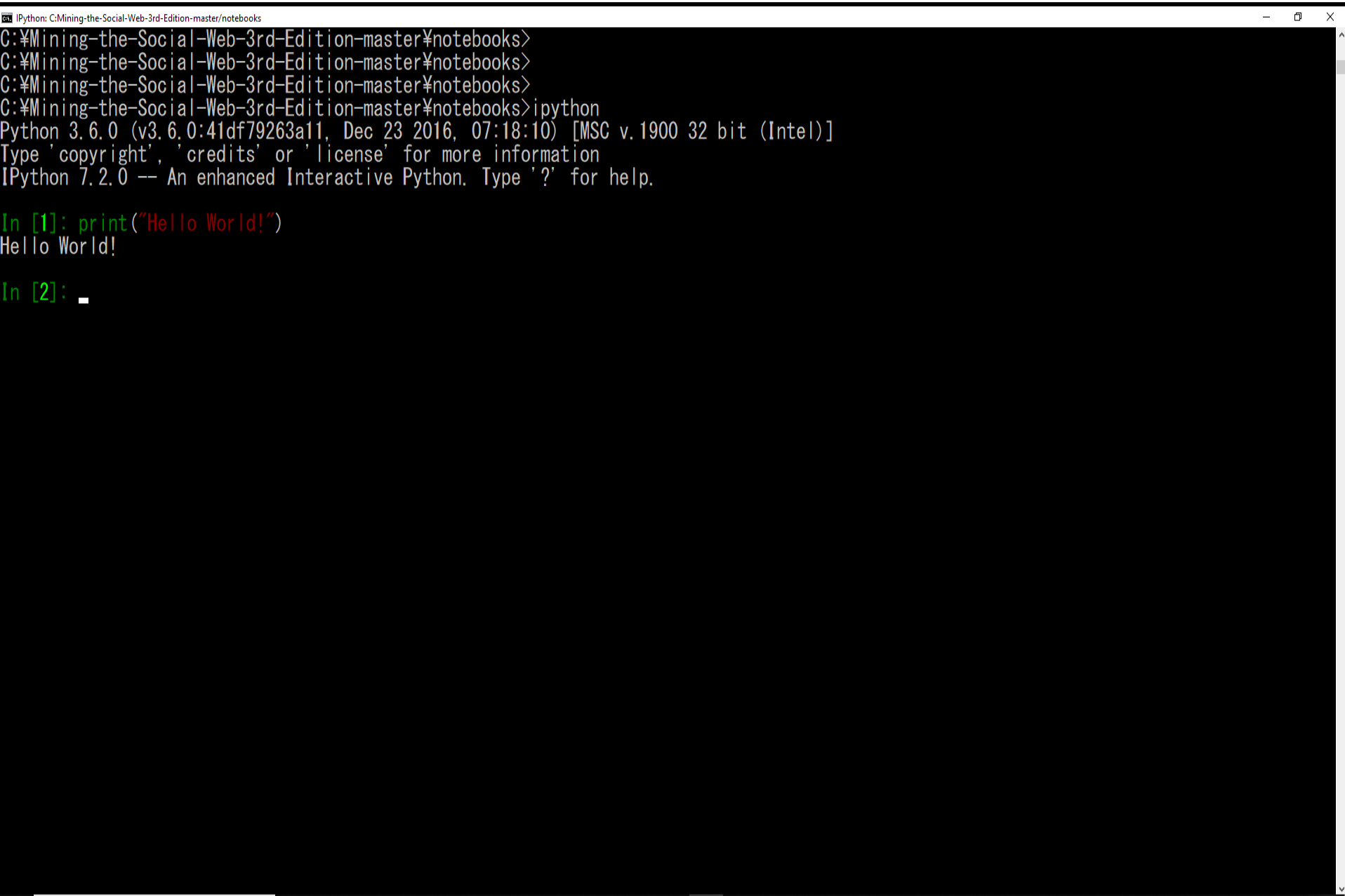
Share your notebooks



COMMUNITY

Stack Overflow
Mailing list
File a bug
Reddit

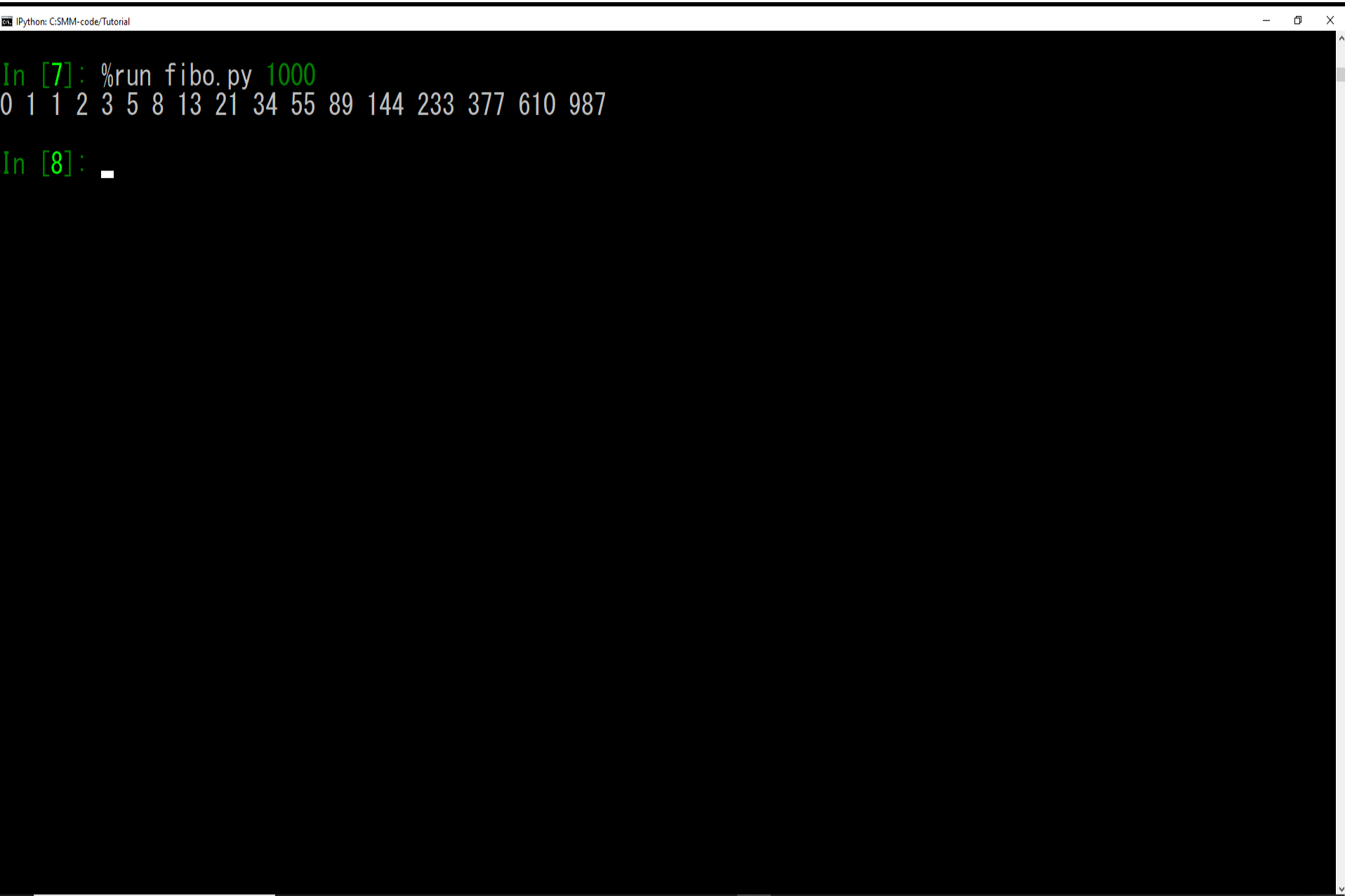
IPython


IPython: C:Mining-the-Social-Web-3rd-Edition-master/notebooks
C:¥Mining-the-Social-Web-3rd-Edition-master¥notebooks>
C:¥Mining-the-Social-Web-3rd-Edition-master¥notebooks>
C:¥Mining-the-Social-Web-3rd-Edition-master¥notebooks>
C:¥Mining-the-Social-Web-3rd-Edition-master¥notebooks>ipython
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 07:18:10) [MSC v.1900 32 bit (Intel)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.2.0 -- An enhanced Interactive Python. Type '?' for help.
In [1]: print("Hello World!")
Hello World!
In [2]:

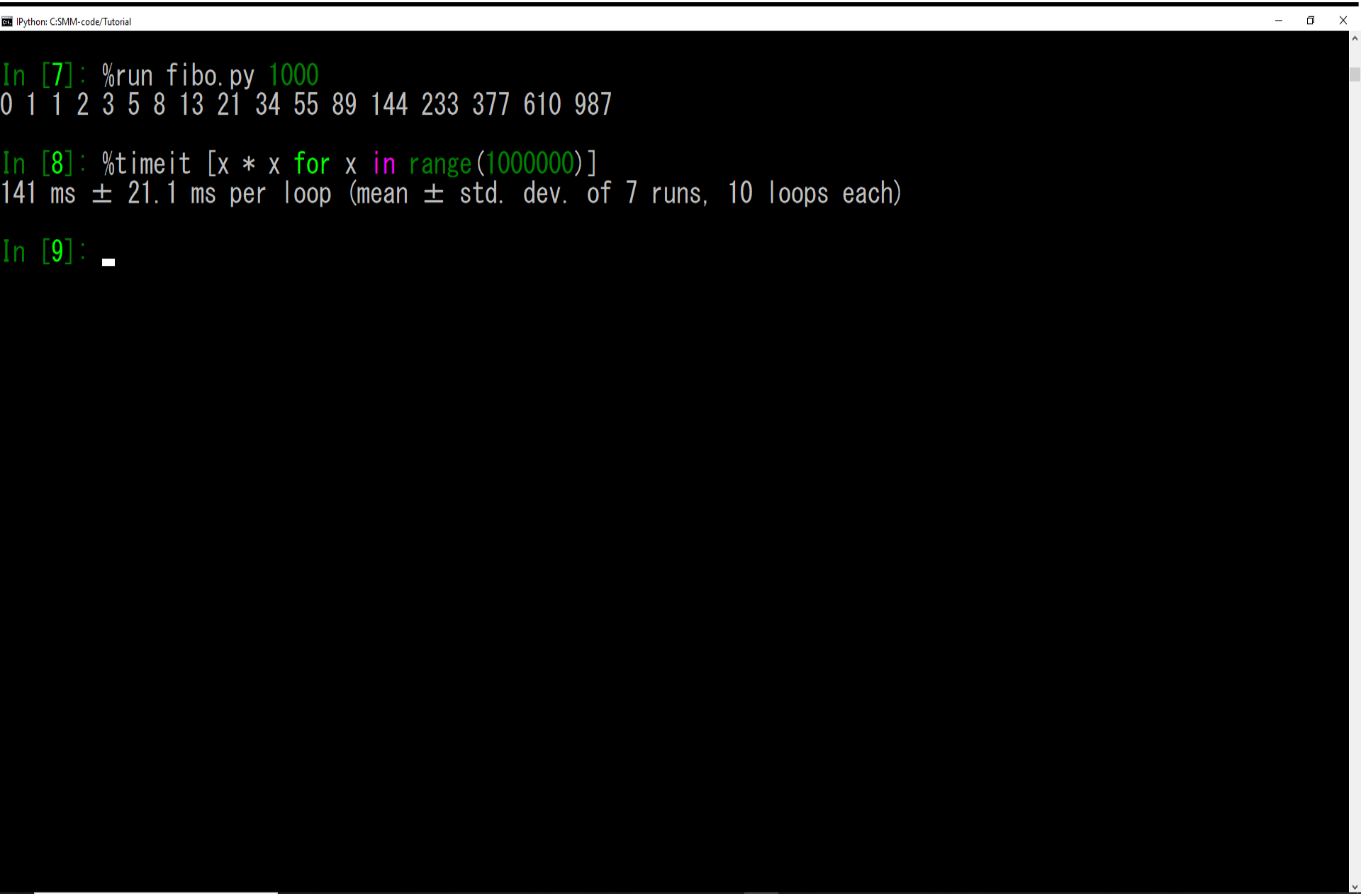
Using IPython as a system shell

```
IPython: C:Mining-the-Social-Web-3rd-Edition-master/notebooks
In [1]: print("Hello World!")
Hello World!

In [2]: pwd
Out[2]: 'C:¥¥Mining-the-Social-Web-3rd-Edition-master¥¥notebooks'

In [3]: ls
 Volume in drive C is Daiyousei
 Volume Serial Number is C2BB-CE8E

 Directory of C:¥Mining-the-Social-Web-3rd-Edition-master¥notebooks

01/18/2019  02:34 PM    <DIR>          .
01/18/2019  02:34 PM    <DIR>          ..
01/18/2019  01:36 PM    <DIR>          .ipynb_checkpoints
11/11/2018  03:27 PM            10,481 _Appendix A - Virtual Machine Experience.ipynb
11/11/2018  03:27 PM            19,629 _Appendix B - OAuth Primer.ipynb
11/11/2018  03:27 PM            27,939 _Appendix C - Python & Jupyter Notebook Tips.ipynb
11/11/2018  03:27 PM             7,196 Chapter 0 - Preface.ipynb
01/18/2019  01:32 PM           113,637 Chapter 1 - Mining Twitter.ipynb
11/11/2018  03:27 PM            18,106 Chapter 2 - Mining Facebook.ipynb
01/18/2019  01:36 PM            20,439 Chapter 3 - Mining Instagram.ipynb
11/11/2018  03:27 PM            33,010 Chapter 4 - Mining LinkedIn.ipynb
11/11/2018  03:27 PM            17,300 Chapter 5 - Mining Text Files.ipynb
12/13/2018  02:02 PM           449,468 Chapter 5 (Video Course) - Mining Google+.ipynb
11/11/2018  03:27 PM            29,408 Chapter 6 - Mining Web Pages.ipynb
11/11/2018  03:27 PM            19,082 Chapter 7 - Mining Mailboxes.ipynb
11/11/2018  03:27 PM            25,511 Chapter 8 - Mining GitHub.ipynb
01/18/2019  02:34 PM         9,032,889 Chapter 9 - Twitter Cookbook.ipynb
11/11/2018  03:27 PM    <DIR>          resources
01/18/2019  01:49 PM           440,254 search_results_CrossFit.pkl
              15 File(s)     10,264,349 bytes
               4 Dir(s)  507,511,222,272 bytes free

In [4]:
```

Executing a script with the %run



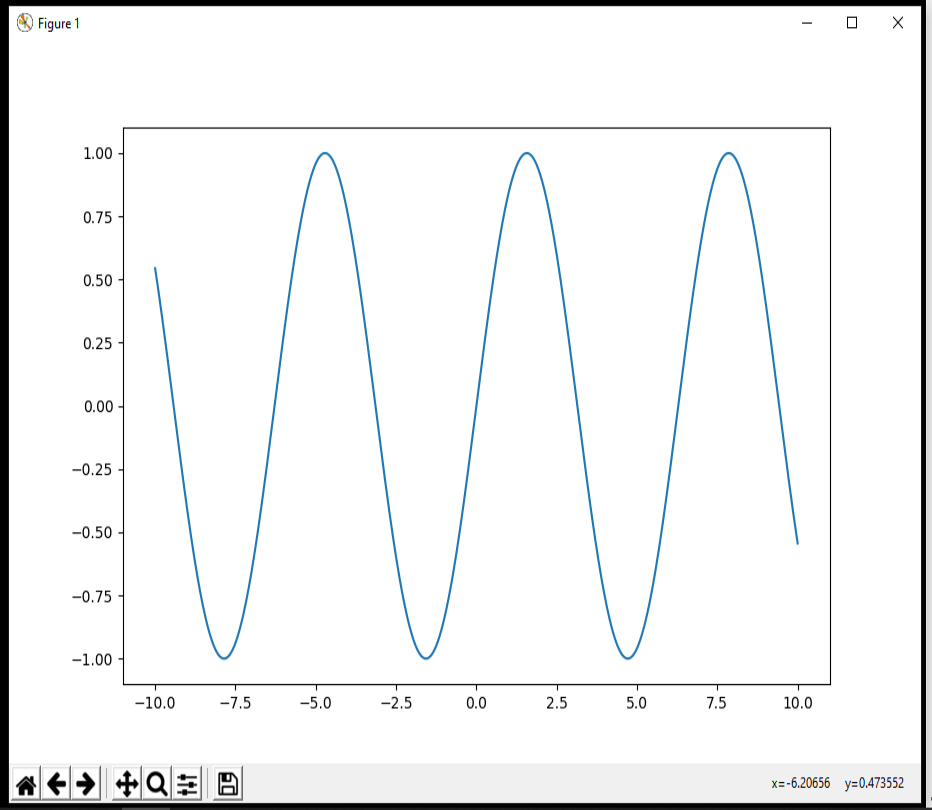
Quick benchmarking with the %timeit



Interactive computing with Pylab

IPython: C:SMM-code/Tutorial

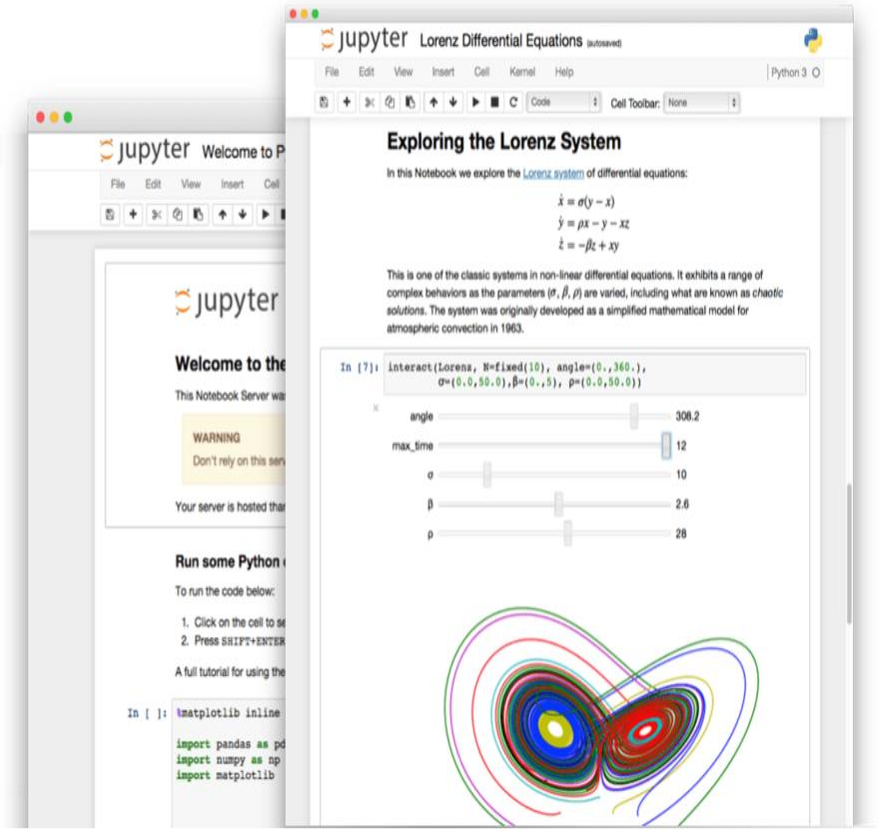
```
In [7]: %run fibo.py 1000
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

In [8]: %timeit [x * x for x in range(1000000)]
141 ms ± 21.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [9]: %pylab
Using matplotlib backend: TkAgg
Populating the interactive namespace from numpy and matplotlib

In [10]: x = linspace(-10, 10, 1000)

In [11]: plot(x, sin(x))
Out[11]: [<matplotlib.lines.Line2D at 0x737c550>]

In [12]:
```



Using the IPython Notebook

- The Notebook brings the functionality of IPython into the browser for multiline text editing features, interactive session reproducibility, and so on.
- It is a modern and powerful way of using Python in an interactive and reproducible way.
- To use the Notebook, call the **ipython (**or **jupyter) notebook** command in a shell.
  - This will launch a local web server on the default port **8888**.
  - Go to http://127.0.0.1:8888/ in a browser and view/create a new Notebook.

jupyter



The Jupyter Notebook

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

Try it in your browser

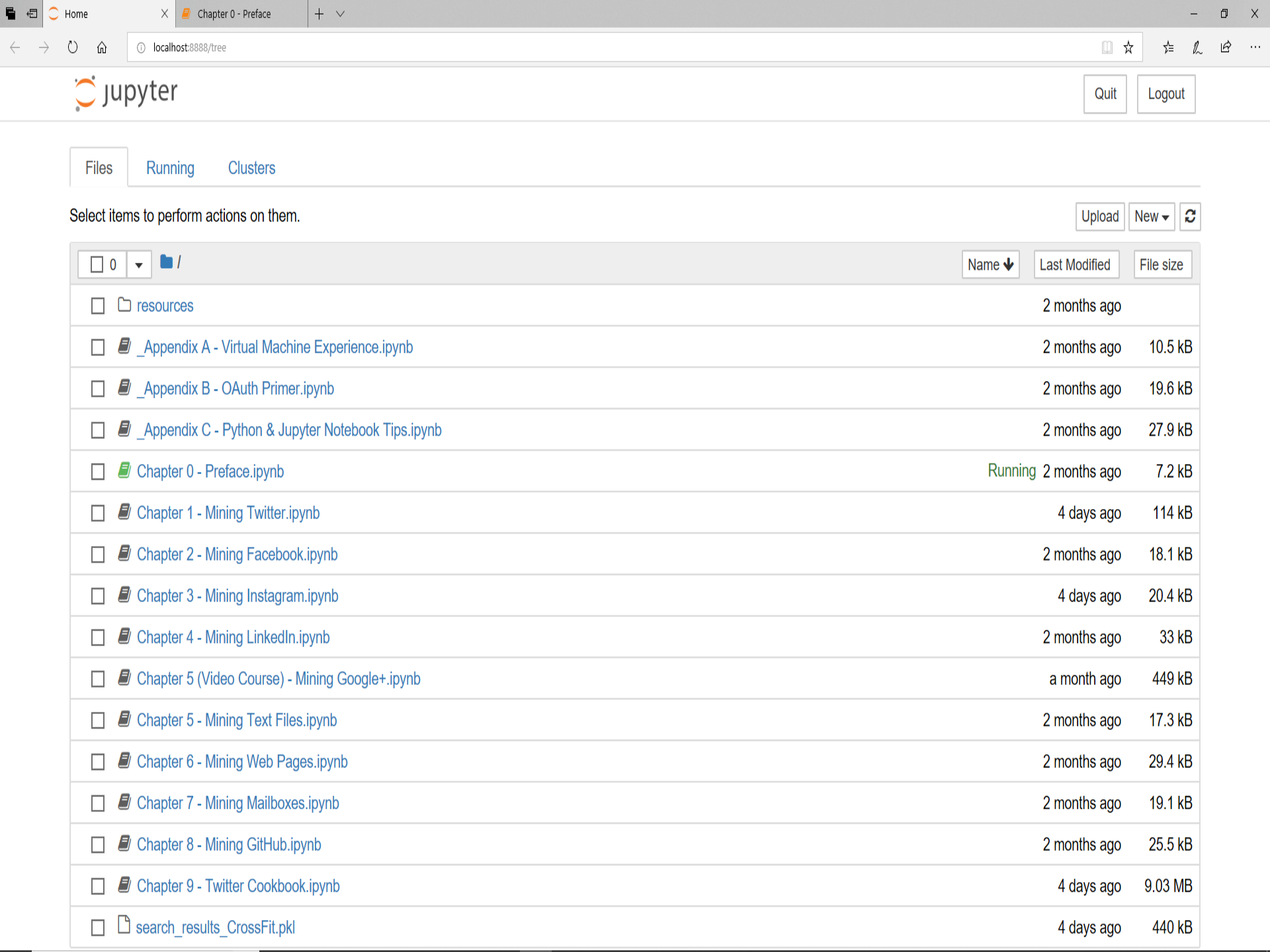
Install the Notebook

Language of choice

Share notebooks

Interactive output

Big data integration


Home
Chapter 0 - Preface
localhost:8888/tree
jupyter
Quit
Logout
Files
Running
Clusters
Select items to perform actions on them.
Upload
New
0
/
Name
Last Modified
File size
resources 2 months ago
_Appendix A - Virtual Machine Experience.ipynb 2 months ago 10.5 kB
_Appendix B - OAuth Primer.ipynb 2 months ago 19.6 kB
_Appendix C - Python & Jupyter Notebook Tips.ipynb 2 months ago 27.9 kB
Chapter 0 - Preface.ipynb Running 2 months ago 7.2 kB
Chapter 1 - Mining Twitter.ipynb 4 days ago 114 kB
Chapter 2 - Mining Facebook.ipynb 2 months ago 18.1 kB
Chapter 3 - Mining Instagram.ipynb 4 days ago 20.4 kB
Chapter 4 - Mining LinkedIn.ipynb 2 months ago 33 kB
Chapter 5 (Video Course) - Mining Google+.ipynb a month ago 449 kB
Chapter 5 - Mining Text Files.ipynb 2 months ago 17.3 kB
Chapter 6 - Mining Web Pages.ipynb 2 months ago 29.4 kB
Chapter 7 - Mining Mailboxes.ipynb 2 months ago 19.1 kB
Chapter 8 - Mining GitHub.ipynb 2 months ago 25.5 kB
Chapter 9 - Twitter Cookbook.ipynb 4 days ago 9.03 MB
search_results_CrossFit.pkl 4 days ago 440 kB

Mining the Social Web (3rd Edition) ¶

Preface

Welcome! Allow me to be the first to offer my congratulations on your decision to take an interest in *Mining the Social Web (3rd Edition)*! This collection of Jupyter Notebooks provides an interactive way to follow along with and explore the numbered examples from the book. Whereas many technical books require you type in the code examples one character at a time or download a source code archive (that may or may not be maintained by the author), this book reinforces the concepts from the sample code in a fun, convenient, and interactive way that really does make the learning experience superior to what you may have previously experienced, so even if you are skeptical, please give it try. I think you'll be pleasantly surprised at the amazing user experience that the Jupyter Notebook affords and just how much easier it is to follow along and adapt the code to your own particular needs.

In the somewhat unlikely event that you've somehow stumbled across this notebook outside of its context on GitHub, you can find the full source code repository here.)

The default expectation is that as a consumer of *Mining the Social Web* and its source code, you will take advantage of the incredible virtual machine experience that's been designed to enhance your pleasure in following along with this book and described in some more detail in the "Appendix A (Virtual Machine Experience)" notebook. The virtual machine experience is powered by Vagrant, an amazing abstraction that works on top of virtualization providers such as Virtualbox and VMWare, and simplifies the tedious and time-consuming configuration management that arise when trying to provide a consitent user experience across multiple platforms, various software versions, 3rd party dependencies that are continually updating, etc.

In short, the virtual machine experience for this book that's powered by Vagrant provides a well-tested "frozen" development environment for the source code in the book that frees you from dealing with myriad system administration complexities that you would otherwise be liable for figuring out to configure your development environment. The virtual machine experience is part of the source code repository and is maintained by the author (with notable contributions from others in the social web mining community.)

Assignment #1 (10 points): Due Sep. 11

- Create a Python module with a **__main__** , and at least 100 lines of code. You must use: **if __name__ == "__main__":**
- Define at least 1 **class**, and at least 1 **function** (method) for each class you have defined. Your **__main__** should instantiate objects of the classes you have designed, and use them to invoke the methods defined in those classes.
- Use **list comprehensions** to create **lists**.
- Use **dictionary comprehensions** to create **dictionaries**.
- Use at least 1 decision-making statement (if-elif)
- Use at least 1 looping statement (for or while).
- Use at least 1 try-except to catch some exceptions.
- Use the **input()** function, or command-line arguments, to get some user input
- Produce some, hopefully interesting, output
- Add comments to make your script easy to understand (not counted toward the 100-line requirement)
- Be creative, and make sure it runs before your submit.