

VINOD KUMAR VERMA, PGT(CS), KV OEF KANPUR &
SACHIN BHARDWAJ, PGT(CS), KV NO.1 TEZPUR

USING PYTHON LIBRARIES

COLLECTION OF MODULES

Introduction

- As our program become larger and more complex the need to organize our code becomes greater. We have already learnt in ***Function chapter that large and complex program should be divided into functions*** that perform a specific task. As we write more and more ***functions in a program, we should consider organizing of functions by storing them in modules***
- A module is simply a file that contains Python code. When we break a program into modules, each modules should contain functions that perform related tasks.
- Commonly used modules that contains source code for generic needs are called ***Libraries***.

Introduction

- When we speak about working with libraries in Python, we are, in fact, working with modules that are created inside Library or Packages. Thus a Python program comprises three main components:
 - Library or Package
 - Module
 - Function/Sub-routine

Relationship between Module, Package and Library in Python

- A **Module** is a file containing Python definitions (docstrings) , functions, variables, classes and statements
- Python package is simply a directory of Python module(s)
- Library is a collection of various packages. Conceptually there is no difference between package and Python library. In Python a library is used to loosely describe a collection of core or main modules

Commonly used Python libraries

STANDARD LIBRARY

math module	Provides mathematical functions
cmath module	Provides function for complex numbers
random module	For generating random numbers
Statistics module	Functions for statistical operation
Urllib	Provides URL handling functions so that you can access websites from within your program.
NumPy library	This library provides some advance math functionalities along with tools to create and manipulate numeric arrays
SciPy library	Another useful library that offers algorithmic and mathematical tools for scientific calculation
Tkinter library	Provides traditional user interface toolkit and helps you to create user friendly GUI interface for different types of applications.
Matplotlib library	Provides functions and tools to produce quality output in variety of formats such as plot, charts, graph etc,

What is module?

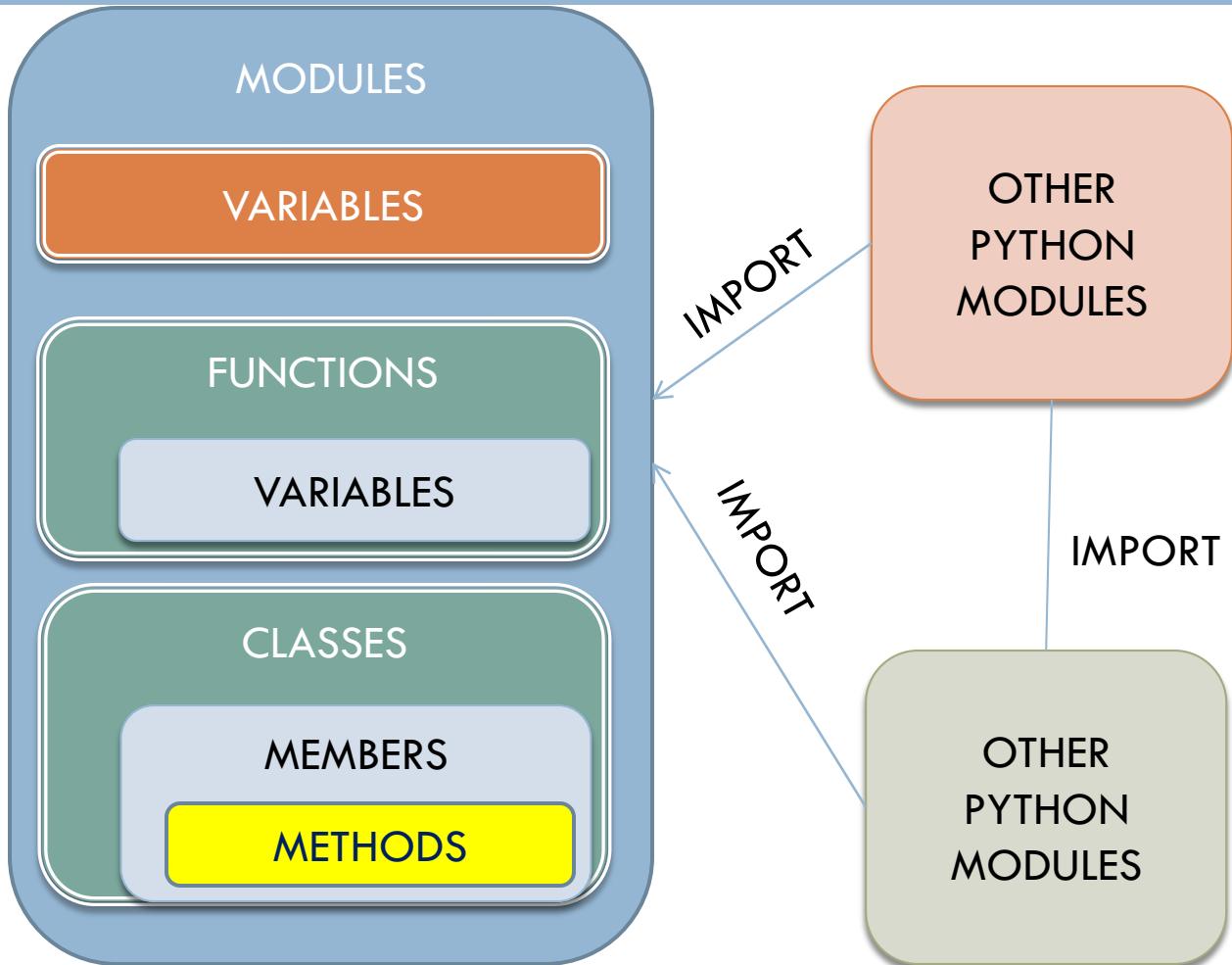
- Act of partitioning a program into individual components(modules) is called modularity. A module is a separate unit in itself.
 - It reduces its complexity to some degree
 - It creates numbers of well-defined, documented boundaries within program.
 - Its contents can be reused in other program, without having to rewrite or recreate them.

Structure of Python module

- A python module is simply a normal python file(.py) and contains functions, constants and other elements.
- Python module may contains following objects:

docstring	Triple quoted comments. Useful for documentation purpose
Variables and constants	For storing values
Classes	To create blueprint of any object
Objects	Object is an instance of class. It represent class in real world
Statements	Instruction
Functions	Group of statements

Composition/Structure of python module



Importing Python modules

- To import entire module
 - **import <module name>**
 - **Example:** `import math`
- To import specific function/object from module:
 - **from <module_name> import <function_name>**
 - **Example:** `from math import sqrt`
- **import *** : can be used to import all names from module into current calling module

Accessing function/constant of imported module

- To use function/constant/variable of imported module we have to specify module name and function name separated by dot(.). This format is known as dot notation.
 - <module_name>.<function_name>
 - **Example:** `print(math.sqrt(25))`

Example : import module_name

The screenshot shows a Python IDE window titled "mod1 - C:/Python27/mod1". The menu bar includes File, Edit, Format, Run, Options, Windows, and Help. The code editor contains the following Python code:

```
import math
print(math.sqrt(25))
print(math.log10(100))
```

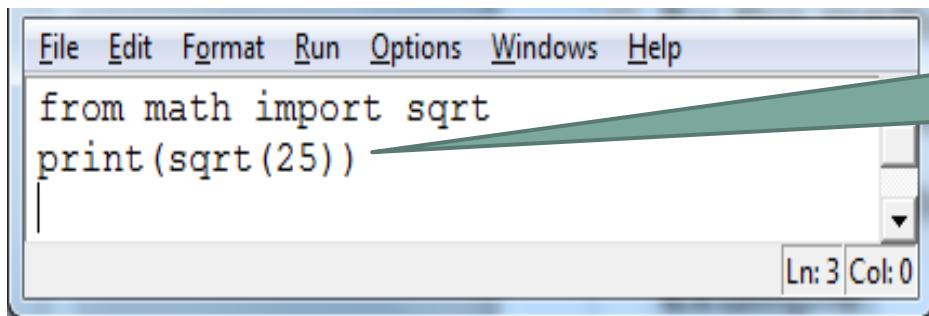
The output window below shows the results of the execution:

```
5.0
2.0
>>> |
```

Here we can see that after **import math**, we have to qualify the name of function with name of package using dot(.) notation

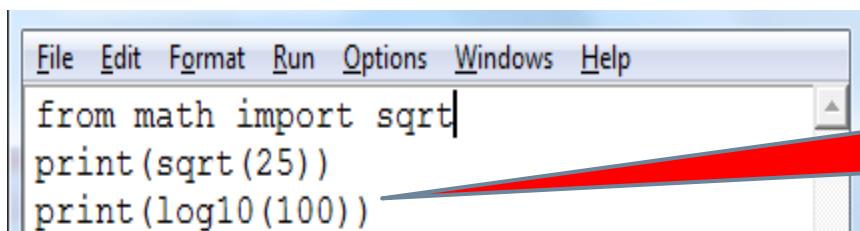
Example: from module import function

- By this method only particular method will be added to our current program. We need not to qualify name of method with name of module. Or example:



```
File Edit Format Run Options Windows Help
from math import sqrt
print(sqrt(25))
Ln: 3 Col: 0
```

Here function
sqrt() is
directly written

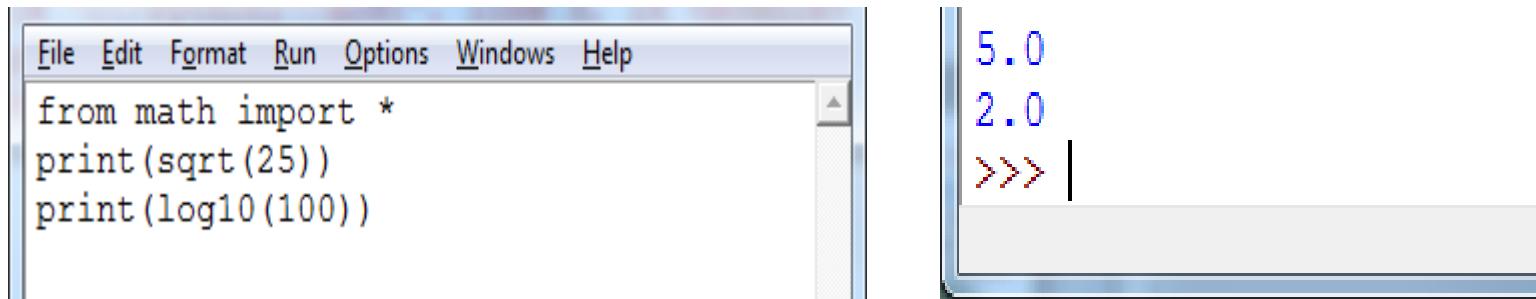


```
File Edit Format Run Options Windows Help
from math import sqrt
print(sqrt(25))
print(log10(100))
```

This line will not
be executed and
gives an error

Example: from module import *

- It is similar to importing the entire package as “import package” but by this method qualifying each function with module name is not required.



The image shows a screenshot of a Python IDLE window. On the left, the code editor pane contains the following Python code:

```
File Edit Format Run Options Windows Help
from math import *
print(sqrt(25))
print(log10(100))
```

On the right, the interactive shell pane shows the output:

```
5.0
2.0
>>> |
```

We can also import multiple elements of module as :
from math import sqrt, log10

Creating our own Module

- Create new python file(.py) and type the following code as:

```
import math
# the is my first module
mynum=100
def area_rect(length,breadth):
    return length*breadth

def area_square(side):
    return side*side

def area_circle(rad):
    return math.pi*rad*rad
```

Execute the following code to import and use your own module

```
>>> import area
>>> print(area.area_rect(7,8))
56
>>> print(area.mynum)
100
>>> from area import area_square
>>> area_square(8)
64
>>> area_circle(8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'area_circle' is not defined
>>> 
```

Save this file are “area.py”

help() function

- Is used to get detailed information about any module like : name of module, functions inside module, variables inside module and name of file etc.

```
>>> help(area)
Help on module area:

NAME
    area

FUNCTIONS
    area_circle(rad)

    area_rect(length, breadth)

    area_square(side)

DATA
    mynum = 100

FILE
    c:\users\vin\area.py

>>>
```

Namespace

- Is a space that holds a bunch of names. Consider an example:
 - In a CCA competition of vidyalaya, there are students from different classes having similar names, say there are three POOJA GUPTA, one from class X, one from XI and one from XII
 - As long as they are in their class there is no confusion, since in X there is only one POOJA GUPTA, and same with XI and XII
 - But problem arises when the students from X, XI, XII are sitting together, now calling just POOJA GUPTA would create confusion-which class's POOJA GUPTA. So one need to qualify the name as class X's POOJA GUPTA, or XI's or XII's and so on.

Namespace

- From the previous example, we can say that class X has its own namespace where there no two names as POOJA GUPTA; same holds for XI and XII
- In Python terms, namespace can be thought of as a named environment holding logical group of related objects.
- For every python module(.py), Python creates a namespace having its name similar to that of module's name. That is, namespace of module AREA is also AREA.
- **When 2 namespace comes together, to resolve any kind of object name dispute, Python asks you to qualify the name of object as <modulename>.<objectname>**

Processing of **import <module>**

- The code of import module is interpreted and executed
- Defined functions and variables in the module are now available to program in **new namespace created by the name of module**
- **For example, if the imported module is area, now you want to call the function area_circle(), it would be called as area.area_circle()**

Processing of `from module import object`

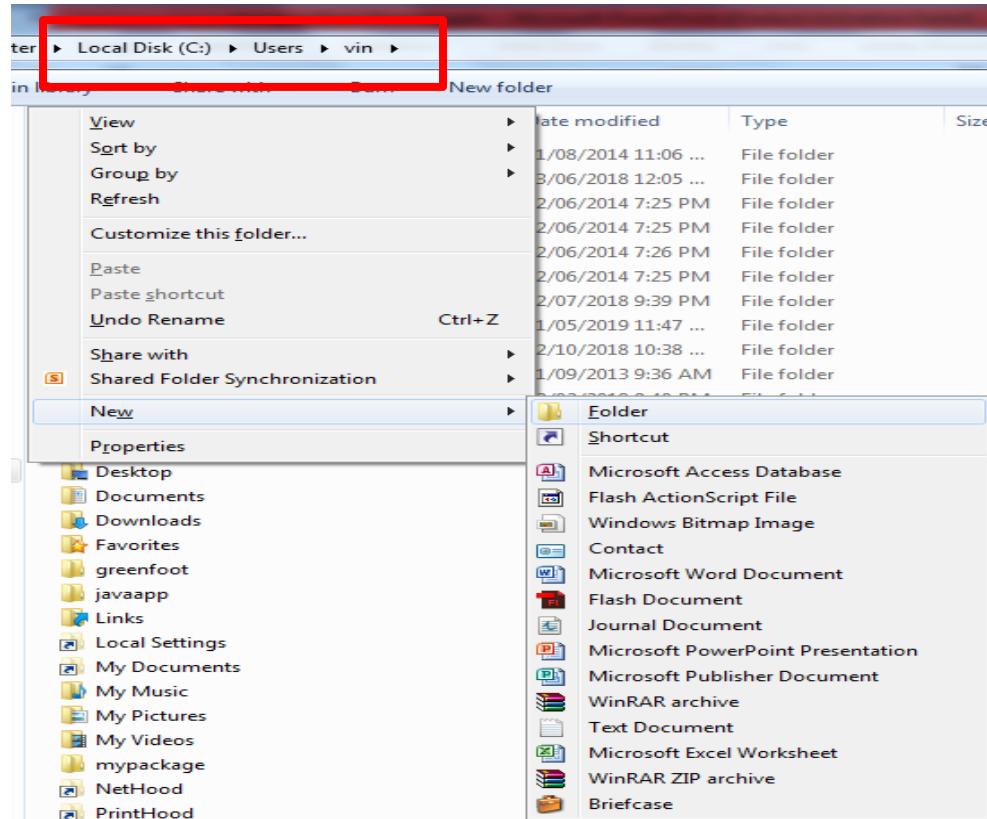
- When we issue `from module import object` command:
 - The code of imported module is interpreted and executed
 - Only the asked function and variables from module are now available in the current namespace i.e. **no new namespace is created that's why we can call object of imported module without qualifying the module name**
 - For example:

```
from math import sqrt
print(sqrt(25))
```
 - *However if the same function name is available in current namespace then local function will hide the imported module's function*
 - *Same will be apply for `from math import *` method*

Creating Package

□ Step 1

- Create a new folder which you want to act as package. The name of folder will be the name of your package



IN THE C:\USERS\VIN
A new Folder “mypackage” is
created.

**Note: you can create folder in
any desired location**

Creating Package

- Step 2: Create modules (.py) and save it in “mypackage” folder

area.py

```
import math
# the is my first module
mynum=100
def area_rect(length,breadth):
    return length*breadth

def area_square(side):
    return side*side

def area_circle(rad):
    return math.pi*rad*rad
```

numcheck.py

```
import math
def even(num):
    if num % 2 == 0:
        return 1
    else:
        return 0

def isprime(num):
    for i in range(2,int(math.sqrt(num)+1)):
        if num % i == 0:
            return 0
    return 1
def palindrome(num):
    mynum = num
    n = 0
    while num!=0:
        r = num % 10
        n = n*10 + r
        num = num // 10
    if mynum == n:
        return 1
    else:
        return 0
```

Creating Package

- Step 2: importing package and modules in python program

```
import mypackage.numcheck  
n = int(input("Enter number "))  
if(mypackage.numcheck.isprime(n)):  
    print("Number is Prime")  
else:  
    print("Number is Composite")
```

Save this file by
“anyname.py” outside
the package folder

RUN THE PROGRAM

Creating Alias of Package/module

- Alias is the another name for imported package/module. It can be used to shorten the package/module name

```
import mypackage.numcheck as mpack
n = int(input("Enter number "))
if(mpack.isprime(n)):
    print("Number is Prime")
else:
    print("Number is Composite")
```

Save this file by
“anyname.py” outside
the package folder

RUN THE PROGRAM

__init__.py file

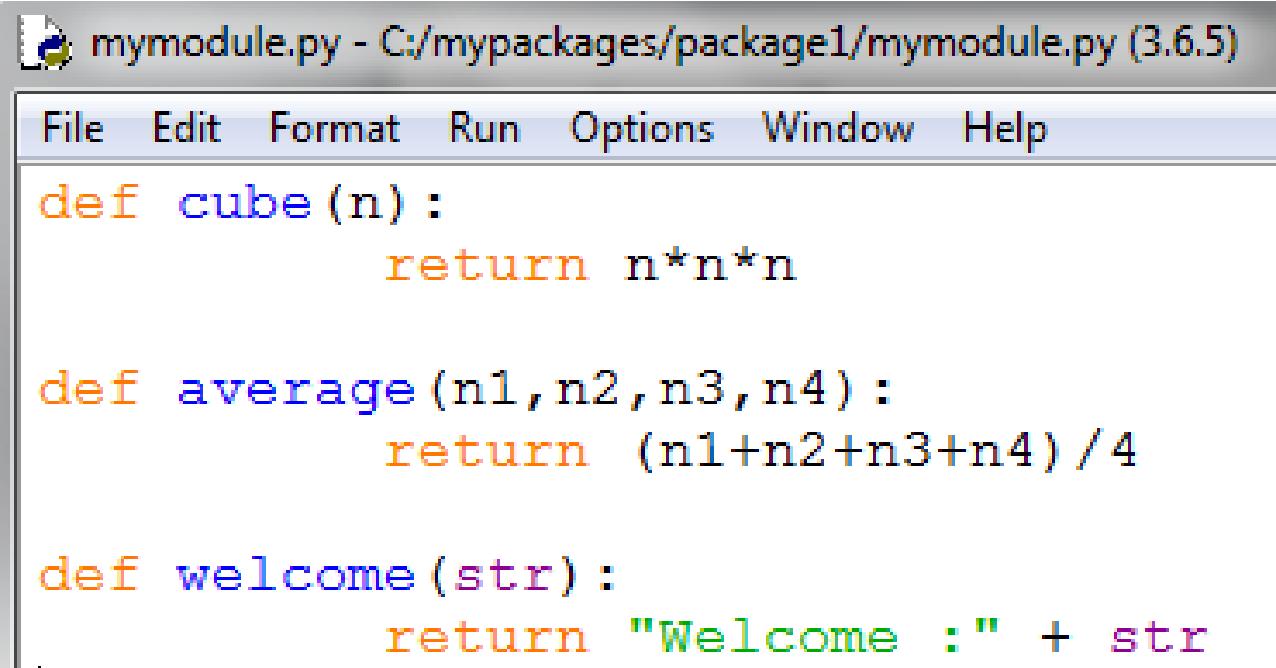
- **__init__.py** (double underscore is prefixed and suffixed) : This file is required to make Python treat directories containing the file as packages.
- **__init__.py** can be just empty file, but it can also execute initialization code for the package.
- **Note: Python 2 requires __init__.py to be inside a folder in order for the folder to be considered a package and made importable but in Python 3.3 and above, it support implicit namespace packages, all folders are packages regardless of the presence of a __init__.py file**
- **So from Python 3.3 it is optional to create __init__.py file**

Example - creating and using package and module and `__init__.py`

- Create a folder to act as a package
- For. e.g. In C: (C Drive) a folder “mypackages” is created
- Create `__init__.py` file (do not write any thing in it) in this folder “mypackages”
- Now create another folder “package1” inside “mypackages”
- Create `__init__.py` file inside “package1” also
- Create a module(.py) for e.g. “mymodule.py” file in “package1” to have some functions in it.

Another example of creating and using package and module and `__init__.py`

□ `mymodule.py`



The screenshot shows a code editor window titled "mymodule.py - C:/mypackages/package1/mymodule.py (3.6.5)". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code itself contains three functions:

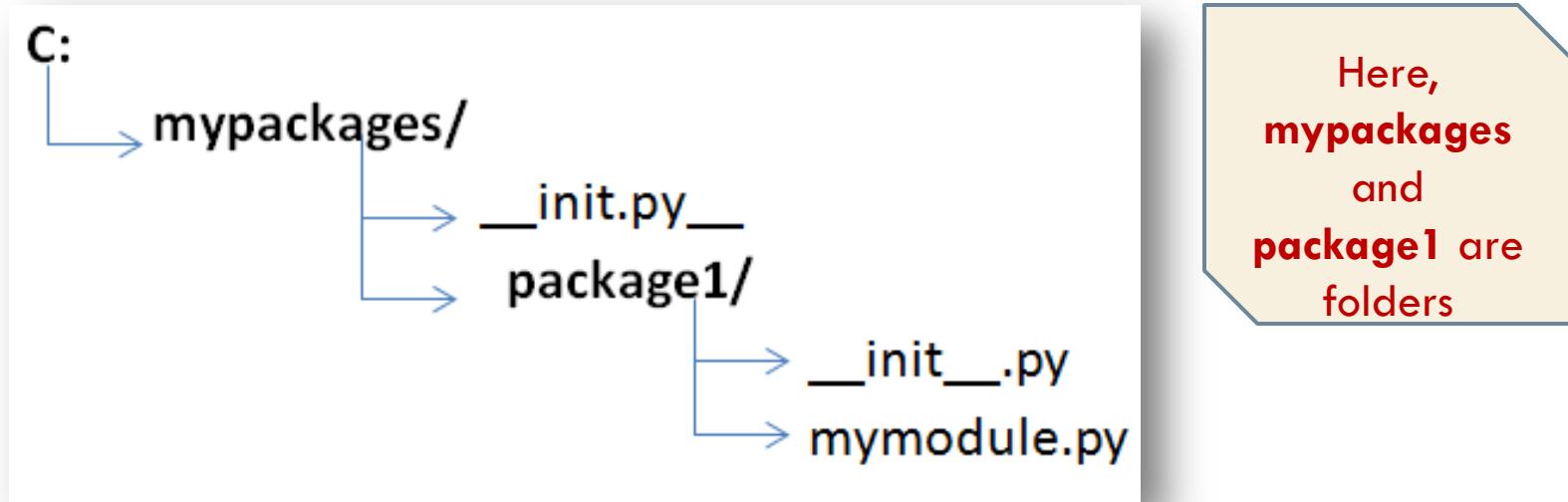
```
def cube(n):
    return n*n*n

def average(n1, n2, n3, n4):
    return (n1+n2+n3+n4) / 4

def welcome(str):
    return "Welcome :" + str
```

Another example of creating and using package and module and `__init__.py`

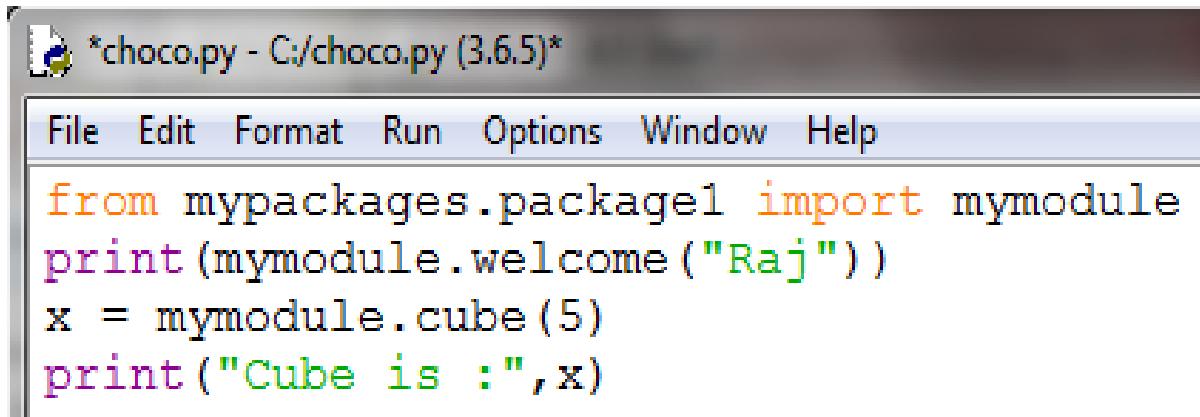
- Now our entire contents will be like this:



- Now in C: (C Drive), create a file for e.g. “choco.py” in which we will import module “mymodule”

Another example of creating and using package and module and `__init__.py`

- To import the packages, we can either use **Absolute addressing** or **Relative Addressing**. (already discussed in data file handling chapter)
- Absolute means following the complete address whereas Relative means with relation to current folder by using Single dot(.)

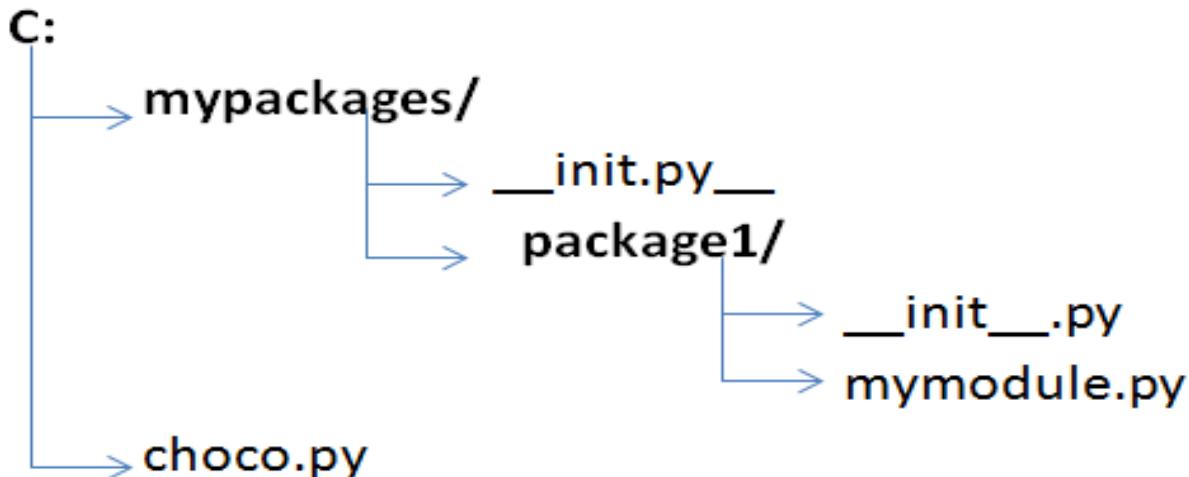


```
*choco.py - C:/choco.py (3.6.5)*
File Edit Format Run Options Window Help
from mypackages.package1 import mymodule
print(mymodule.welcome("Raj"))
x = mymodule(cube(5))
print("Cube is :", x)
```

Absolute path of package

Another example of creating and using package and module and `__init__.py`

- Now our entire content structure will be as:



- Run the `choco.py` file and you will get the output.

TIP : to use package from any location

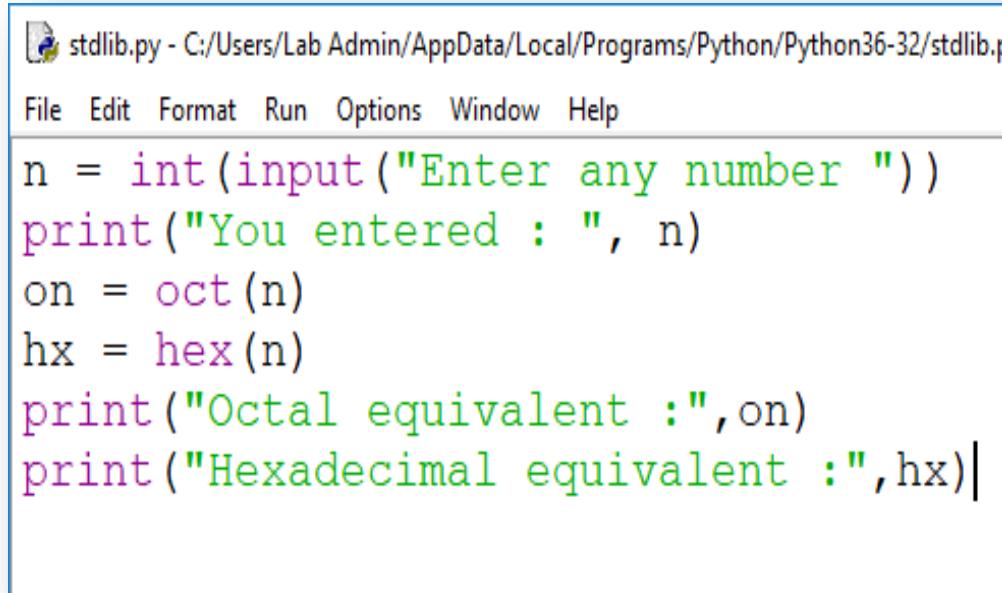
- At this time, the package “mypackages” will be accessible from its location only i.e. the file which wants to import this package must be in same folder/drive where “mypackages” is.
- To enable “mypackages” to be used from any location, Copy this mypackages to Python’s **site-packages** folder inside **Lib** folder of Python’s installation folder. (*Try with copying to Lib folder also*)
- Path of site-packages is :
C:\Users\vin\AppData\Local\Programs\Python\Python 36-32\Lib\site-packages
- After this you can import this package “mypackages” in any python file.

Using Python's Built-in Function

- Python's standard library is very extensive that offers many built-in functions that **we can use without having to import any library.**
- Using Python's Built-in functions
 - `Function_name()`

Mathematical and String functions

- `oct(int)` : return octal string for given number by prefixing “0o”
- `hex(int)` : return octal string for given number by prefixing “0x”



```
stdlib.py - C:/Users/Lab Admin/AppData/Local/Programs/Python/Python36-32/stdlib.py
File Edit Format Run Options Window Help
n = int(input("Enter any number "))
print("You entered : ", n)
on = oct(n)
hx = hex(n)
print("Octal equivalent :",on)
print("Hexadecimal equivalent :",hx)|
```

```
Enter any number 12
You entered : 12
Octal equivalent : 0o14
Hexadecimal equivalent : 0xc
```

Mathematical and String functions

- `int(number)` : function convert the fractional number to integer
- `int(string)` : convert the given string to integer
- `round(number,[nDIGIT])` : return number rounded to nDIGIT after decimal points. If nDIGIT is not given, it returns nearest integer to its input.
- Examples: (next slide)

Mathematical and String functions

```
n1 = float(input("Enter first number "))
n2 = float(input("Enter second number "))
n3 = n1/n2
print("Result :",n3)
```

Enter first number 5
Enter second number 3
Result : 1.6666666666666667

```
n1 = float(input("Enter first number "))
n2 = float(input("Enter second number "))
n3 = n1/n2
print("Result :",round(n3,2))
```

Enter first number 5
Enter second number 3
Result : 1.67

```
n1 = float(input("Enter first number "))
n2 = float(input("Enter second number "))
n3 = n1/n2
print("Result :",round(n3))
```

Enter first number 5
Enter second number 3
Result : 2

Other String function

- We have already used many string function in class XI, here are few new functions
 - **<string>.join()** : if the string based iterator is a string then the <string> is inserted after every character of the string.
 - If the string based iterator is a list or tuple of strings then, the given string/character is joined after each member of the list or tuple. **BUT the tuple or list must have all members as string otherwise Python will raise an error**
- Examples (next slide)

Other String function

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900
on win32
Type "copyright", "credits" or "license()" for more information.
>>> "#".join("INDIA")
'I#N#D#I#A'
>>> "$$".join(("IPL","CCL","FIFA"))
'IPL$$CCL$$FIFA'
>>> "$$".join(("IPL","FIFA",123))
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    "$$".join(("IPL","FIFA",123))
TypeError: sequence item 2: expected str instance, int found
>>>
```

Other String function

- We have already used many string function in class XI, here are few new functions
 - **<string>.split()** : allow to divide string in multiple parts and store it as a LIST. If you do not provide delimiter then by default string will be split using space otherwise using given character.
 - **<str>.replace()** : allows you to replace any part of string with another string.
 - **Example (NEXT SLIDE)**

Example (split() and replace())

```
msg = "KV OEF KANPUR"
msg1 = msg.split()      # by default split the string using space
print(msg1) # output will be in the form of list
print(type(msg1))
```

```
msg2="kv@gmail.com"
msg3=msg2.split("@")    #now split using '@'
print(msg3) # output will be in the form of list
print(type(msg3))
```

```
mymsg="I love Python Programming"
mymsg2=mymsg.replace("Python","C++") #replaces 'Python' with 'C++'
print(mysmg2)
```

```
['KV', 'OEF', 'KANPUR']
<class 'list'>
['kv', 'gmail.com']
<class 'list'>
>>>
RESTART: C:/Users/Lab A
Y
['KV', 'OEF', 'KANPUR']
<class 'list'>
['kv', 'gmail.com']
<class 'list'>
I love C++ Programming
```

Using **URLLIB** and **WEBBROWSER** modules

- Python provides **urllib** module to send http request and receive the result from within your program. To use **urllib** we have to first import **urllib** module.
- **urllib** module is a collection of sub-module like request, error, parse etc. following functions of **urllib** we can use: (**next slide**)

Functions of URLLIB

FUNCTION NAME	PURPOSE
urllib.request.urlopen(<url>)	Opens a website or network object denoted by URL for reading and return file like object using which other functions are often used
urlopen_object.read()	Return HTML or the source code of given url
urlopen_object..getcode()	Returns HTTP status code where 200 means 'all okay' 404 means url not found etc. 301, 302 means some kind of redirections happened
urlopen_object.headers	Stores metadata about the open URL
urlopen_object.info()	Returns same information as by headers
urlopen_object.geturl()	Return URL string

Example:

```
import urllib.request
x=urllib.request.urlopen('https://www.google.com/')
print("URL READ : ",x.geturl())
print("HEADERS : ",x.headers)
print("INFO : ",x.info())
print("CODE : ",x.getcode())
```

```
(base) C:\Users\vin>python use.py
URL READ : https://www.google.com/
HEADERS : Date: Thu, 16 May 2019 17:14:45 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: 1P_JAR=2019-05-16-17; expires=Sat, 15-Jun-2019 17:14:45 GMT; path=/;
domain=.google.com
Set-Cookie: NID=183=hBwj19cMG2hFqDHUVKRsoPE-JvwJRCcz2kSmxL6eWtJNoNozzUj_zwrM-Ia
uIr7XDb5XHUu8G4aM27IKMETI4GY9l0pA57buFyJIY_CZ2-tYtEeUk1sMB7aExBAigAq2XJgYmMmaF_
ufWmwVxbvpku2quvFSBIiFWj8NcqywQ; expires=Fri, 15-Nov-2019 17:14:45 GMT; path=/;
domain=.google.com; HttpOnly
Alt-Svc: quic=":443"; ma=2592000; v="46,44,43,39"
Accept-Ranges: none
Vary: Accept-Encoding
Connection: close
CODE : 200
```

Before execution of the above code, make sure PC is connected to working internet.

WEBBROWSER MODULE

- Provides functionality to open a website in a window or tab or web browser on your computer, from within your program.
- To use webbrowser module we must import the module as:

□ **import webbrowser**

```
import webbrowser
myurl = input("Enter url to open ")
webbrowser.open_new(myurl)
```

```
import webbrowser
text = input("Enter 'Text to Search Online ' :")
webbrowser.open_new("https://www.google.com?q="+text)
```

The Python Library Reference

Release 3.8.2

**Guido van Rossum
and the Python development team**

February 28, 2020

**Python Software Foundation
Email: docs@python.org**

CONTENTS

1	Introduction	3
1.1	Notes on availability	3
2	Built-in Functions	5
3	Built-in Constants	27
3.1	Constants added by the <code>site</code> module	28
4	Built-in Types	29
4.1	Truth Value Testing	29
4.2	Boolean Operations — <code>and</code> , <code>or</code> , <code>not</code>	29
4.3	Comparisons	30
4.4	Numeric Types — <code>int</code> , <code>float</code> , <code>complex</code>	30
4.5	Iterator Types	36
4.6	Sequence Types — <code>list</code> , <code>tuple</code> , <code>range</code>	37
4.7	Text Sequence Type — <code>str</code>	43
4.8	Binary Sequence Types — <code>bytes</code> , <code>bytearray</code> , <code>memoryview</code>	53
4.9	Set Types — <code>set</code> , <code>frozenset</code>	75
4.10	Mapping Types — <code>dict</code>	77
4.11	Context Manager Types	81
4.12	Other Built-in Types	82
4.13	Special Attributes	85
5	Built-in Exceptions	87
5.1	Base classes	88
5.2	Concrete exceptions	88
5.3	Warnings	94
5.4	Exception hierarchy	95
6	Text Processing Services	97
6.1	<code>string</code> — Common string operations	97
6.2	<code>re</code> — Regular expression operations	108
6.3	<code>difflib</code> — Helpers for computing deltas	127
6.4	<code>textwrap</code> — Text wrapping and filling	138
6.5	<code>unicodedata</code> — Unicode Database	142
6.6	<code>stringprep</code> — Internet String Preparation	144
6.7	<code>readline</code> — GNU readline interface	145
6.8	<code>rlcompleter</code> — Completion function for GNU readline	150
7	Binary Data Services	151
7.1	<code>struct</code> — Interpret bytes as packed binary data	151

7.2	<code>codecs</code> — Codec registry and base classes	157
8	Data Types	175
8.1	<code>datetime</code> — Basic date and time types	175
8.2	<code>calendar</code> — General calendar-related functions	210
8.3	<code>collections</code> — Container datatypes	215
8.4	<code>collections.abc</code> — Abstract Base Classes for Containers	232
8.5	<code>heapq</code> — Heap queue algorithm	236
8.6	<code>bisect</code> — Array bisection algorithm	240
8.7	<code>array</code> — Efficient arrays of numeric values	242
8.8	<code>weakref</code> — Weak references	245
8.9	<code>types</code> — Dynamic type creation and names for built-in types	253
8.10	<code>copy</code> — Shallow and deep copy operations	257
8.11	<code>pprint</code> — Data pretty printer	258
8.12	<code>reprlib</code> — Alternate <code>repr()</code> implementation	264
8.13	<code>enum</code> — Support for enumerations	266
9	Numeric and Mathematical Modules	285
9.1	<code>numbers</code> — Numeric abstract base classes	285
9.2	<code>math</code> — Mathematical functions	288
9.3	<code>cmath</code> — Mathematical functions for complex numbers	294
9.4	<code>decimal</code> — Decimal fixed point and floating point arithmetic	298
9.5	<code>fractions</code> — Rational numbers	325
9.6	<code>random</code> — Generate pseudo-random numbers	328
9.7	<code>statistics</code> — Mathematical statistics functions	334
10	Functional Programming Modules	345
10.1	<code>itertools</code> — Functions creating iterators for efficient looping	345
10.2	<code>functools</code> — Higher-order functions and operations on callable objects	360
10.3	<code>operator</code> — Standard operators as functions	368
11	File and Directory Access	377
11.1	<code>pathlib</code> — Object-oriented filesystem paths	377
11.2	<code>os.path</code> — Common pathname manipulations	393
11.3	<code>fileinput</code> — Iterate over lines from multiple input streams	398
11.4	<code>stat</code> — Interpreting <code>stat()</code> results	401
11.5	<code>filecmp</code> — File and Directory Comparisons	406
11.6	<code>tempfile</code> — Generate temporary files and directories	408
11.7	<code>glob</code> — Unix style pathname pattern expansion	412
11.8	<code>fnmatch</code> — Unix filename pattern matching	414
11.9	<code>linecache</code> — Random access to text lines	415
11.10	<code>shutil</code> — High-level file operations	415
12	Data Persistence	425
12.1	<code>pickle</code> — Python object serialization	425
12.2	<code>copyreg</code> — Register <code>pickle</code> support functions	442
12.3	<code>shelve</code> — Python object persistence	443
12.4	<code>marshal</code> — Internal Python object serialization	445
12.5	<code>dbm</code> — Interfaces to Unix “databases”	447
12.6	<code>sqlite3</code> — DB-API 2.0 interface for SQLite databases	451
13	Data Compression and Archiving	473
13.1	<code>zlib</code> — Compression compatible with <code>gzip</code>	473
13.2	<code>gzip</code> — Support for <code>gzip</code> files	477
13.3	<code>bz2</code> — Support for <code>bzip2</code> compression	480

13.4	<code>lzma</code> — Compression using the LZMA algorithm	484
13.5	<code>zipfile</code> — Work with ZIP archives	490
13.6	<code>tarfile</code> — Read and write tar archive files	499
14	File Formats	511
14.1	<code>csv</code> — CSV File Reading and Writing	511
14.2	<code>configparser</code> — Configuration file parser	518
14.3	<code>netrc</code> — netrc file processing	535
14.4	<code>xdrlib</code> — Encode and decode XDR data	536
14.5	<code>plistlib</code> — Generate and parse Mac OS X .plist files	539
15	Cryptographic Services	543
15.1	<code>hashlib</code> — Secure hashes and message digests	543
15.2	<code>hmac</code> — Keyed-Hashing for Message Authentication	553
15.3	<code>secrets</code> — Generate secure random numbers for managing secrets	555
16	Generic Operating System Services	559
16.1	<code>os</code> — Miscellaneous operating system interfaces	559
16.2	<code>io</code> — Core tools for working with streams	611
16.3	<code>time</code> — Time access and conversions	624
16.4	<code>argparse</code> — Parser for command-line options, arguments and sub-commands	633
16.5	<code> getopt</code> — C-style parser for command line options	666
16.6	<code>logging</code> — Logging facility for Python	668
16.7	<code>logging.config</code> — Logging configuration	684
16.8	<code>logging.handlers</code> — Logging handlers	694
16.9	<code>getpass</code> — Portable password input	707
16.10	<code>curses</code> — Terminal handling for character-cell displays	708
16.11	<code>curses.textpad</code> — Text input widget for curses programs	725
16.12	<code>curses.ascii</code> — Utilities for ASCII characters	727
16.13	<code>curses.panel</code> — A panel stack extension for curses	729
16.14	<code>platform</code> — Access to underlying platform's identifying data	730
16.15	<code>errno</code> — Standard errno system symbols	733
16.16	<code>ctypes</code> — A foreign function library for Python	739
17	Concurrent Execution	773
17.1	<code>threading</code> — Thread-based parallelism	773
17.2	<code>multiprocessing</code> — Process-based parallelism	786
17.3	<code>multiprocessing.shared_memory</code> — Provides shared memory for direct access across processes	829
17.4	<code>The concurrent package</code>	834
17.5	<code>concurrent.futures</code> — Launching parallel tasks	834
17.6	<code>subprocess</code> — Subprocess management	840
17.7	<code>sched</code> — Event scheduler	858
17.8	<code>queue</code> — A synchronized queue class	860
17.9	<code>_thread</code> — Low-level threading API	863
17.10	<code>_dummy_thread</code> — Drop-in replacement for the <code>_thread</code> module	866
17.11	<code>dummy_threading</code> — Drop-in replacement for the <code>threading</code> module	866
18	contextvars — Context Variables	867
18.1	Context Variables	867
18.2	Manual Context Management	868
18.3	asyncio support	870
19	Networking and Interprocess Communication	871
19.1	<code>asyncio</code> — Asynchronous I/O	871
19.2	<code>socket</code> — Low-level networking interface	959

19.3	<code>ssl</code> — TLS/SSL wrapper for socket objects	983
19.4	<code>select</code> — Waiting for I/O completion	1018
19.5	<code>selectors</code> — High-level I/O multiplexing	1025
19.6	<code>asyncore</code> — Asynchronous socket handler	1029
19.7	<code>asynchat</code> — Asynchronous socket command/response handler	1033
19.8	<code>signal</code> — Set handlers for asynchronous events	1035
19.9	<code>mmap</code> — Memory-mapped file support	1042
20	Internet Data Handling	1047
20.1	<code>email</code> — An email and MIME handling package	1047
20.2	<code>json</code> — JSON encoder and decoder	1105
20.3	<code>mailcap</code> — Mailcap file handling	1114
20.4	<code>mailbox</code> — Manipulate mailboxes in various formats	1115
20.5	<code>mimetypes</code> — Map filenames to MIME types	1133
20.6	<code>base64</code> — Base16, Base32, Base64, Base85 Data Encodings	1136
20.7	<code>binhex</code> — Encode and decode binhex4 files	1139
20.8	<code>binascii</code> — Convert between binary and ASCII	1140
20.9	<code>quopri</code> — Encode and decode MIME quoted-printable data	1142
20.10	<code>uu</code> — Encode and decode uuencode files	1143
21	Structured Markup Processing Tools	1145
21.1	<code>html</code> — HyperText Markup Language support	1145
21.2	<code>html.parser</code> — Simple HTML and XHTML parser	1145
21.3	<code>html.entities</code> — Definitions of HTML general entities	1150
21.4	<code>XML Processing Modules</code>	1151
21.5	<code>xml.etree.ElementTree</code> — The ElementTree XML API	1152
21.6	<code>xml.dom</code> — The Document Object Model API	1171
21.7	<code>xml.dom.minidom</code> — Minimal DOM implementation	1182
21.8	<code>xml.dom.pulldom</code> — Support for building partial DOM trees	1186
21.9	<code>xml.sax</code> — Support for SAX2 parsers	1188
21.10	<code>xml.sax.handler</code> — Base classes for SAX handlers	1190
21.11	<code>xml.sax.saxutils</code> — SAX Utilities	1195
21.12	<code>xml.sax.xmlreader</code> — Interface for XML parsers	1196
21.13	<code>xml.parsers.expat</code> — Fast XML parsing using Expat	1200
22	Internet Protocols and Support	1211
22.1	<code>webbrowser</code> — Convenient Web-browser controller	1211
22.2	<code>cgi</code> — Common Gateway Interface support	1213
22.3	<code>cgitb</code> — Traceback manager for CGI scripts	1220
22.4	<code>wsgiref</code> — WSGI Utilities and Reference Implementation	1221
22.5	<code>urllib</code> — URL handling modules	1231
22.6	<code>urllib.request</code> — Extensible library for opening URLs	1231
22.7	<code>urllib.response</code> — Response classes used by urllib	1249
22.8	<code>urllib.parse</code> — Parse URLs into components	1250
22.9	<code>urllib.error</code> — Exception classes raised by urllib.request	1258
22.10	<code>urllib.robotparser</code> — Parser for robots.txt	1258
22.11	<code>http</code> — HTTP modules	1260
22.12	<code>http.client</code> — HTTP protocol client	1262
22.13	<code>ftplib</code> — FTP protocol client	1269
22.14	<code>poplib</code> — POP3 protocol client	1274
22.15	<code>imaplib</code> — IMAP4 protocol client	1277
22.16	<code>nntplib</code> — NNTP protocol client	1283
22.17	<code>smtplib</code> — SMTP protocol client	1290
22.18	<code>smtpd</code> — SMTP Server	1297

22.19	<code>telnetlib</code> — Telnet client	1300
22.20	<code>uuid</code> — UUID objects according to RFC 4122	1303
22.21	<code>socketserver</code> — A framework for network servers	1306
22.22	<code>http.server</code> — HTTP servers	1315
22.23	<code>http.cookies</code> — HTTP state management	1320
22.24	<code>http.cookiejar</code> — Cookie handling for HTTP clients	1324
22.25	<code>xmlrpc</code> — XMLRPC server and client modules	1332
22.26	<code>xmlrpc.client</code> — XML-RPC client access	1333
22.27	<code>xmlrpc.server</code> — Basic XML-RPC servers	1341
22.28	<code>ipaddress</code> — IPv4/IPv6 manipulation library	1347
23	Multimedia Services	1361
23.1	<code>audioop</code> — Manipulate raw audio data	1361
23.2	<code>aifc</code> — Read and write AIFF and AIFC files	1364
23.3	<code>sunau</code> — Read and write Sun AU files	1366
23.4	<code>wave</code> — Read and write WAV files	1369
23.5	<code>chunk</code> — Read IFF chunked data	1372
23.6	<code>colorsyst</code> — Conversions between color systems	1373
23.7	<code>imghdr</code> — Determine the type of an image	1374
23.8	<code>sndhdr</code> — Determine type of sound file	1375
23.9	<code>osaudiodev</code> — Access to OSS-compatible audio devices	1375
24	Internationalization	1381
24.1	<code>gettext</code> — Multilingual internationalization services	1381
24.2	<code>locale</code> — Internationalization services	1390
25	Program Frameworks	1399
25.1	<code>turtle</code> — Turtle graphics	1399
25.2	<code>cmd</code> — Support for line-oriented command interpreters	1434
25.3	<code>shlex</code> — Simple lexical analysis	1439
26	Graphical User Interfaces with Tk	1445
26.1	<code>tkinter</code> — Python interface to Tcl/Tk	1445
26.2	<code>tkinter.ttk</code> — Tk themed widgets	1456
26.3	<code>tkinter.tix</code> — Extension widgets for Tk	1474
26.4	<code>tkinter.scrolledtext</code> — Scrolled Text Widget	1479
26.5	<code>IDLE</code>	1479
26.6	Other Graphical User Interface Packages	1490
27	Development Tools	1493
27.1	<code>typing</code> — Support for type hints	1493
27.2	<code>pydoc</code> — Documentation generator and online help system	1513
27.3	<code>doctest</code> — Test interactive Python examples	1514
27.4	<code>unittest</code> — Unit testing framework	1537
27.5	<code>unittest.mock</code> — mock object library	1567
27.6	<code>unittest.mock</code> — getting started	1608
27.7	<code>2to3</code> - Automated Python 2 to 3 code translation	1628
27.8	<code>test</code> — Regression tests package for Python	1634
27.9	<code>test.support</code> — Utilities for the Python test suite	1636
27.10	<code>test.support.script_helper</code> — Utilities for the Python execution tests	1649
28	Debugging and Profiling	1651
28.1	<code>Audit</code> events table	1651
28.2	<code>bdb</code> — Debugger framework	1654
28.3	<code>faulthandler</code> — Dump the Python traceback	1659

28.4	<code>pdb</code> — The Python Debugger	1661
28.5	<code>The Python Profilers</code>	1667
28.6	<code>timeit</code> — Measure execution time of small code snippets	1676
28.7	<code>trace</code> — Trace or track Python statement execution	1681
28.8	<code>tracemalloc</code> — Trace memory allocations	1683
29	Software Packaging and Distribution	1695
29.1	<code>distutils</code> — Building and installing Python modules	1695
29.2	<code>ensurepip</code> — Bootstrapping the <code>pip</code> installer	1696
29.3	<code>venv</code> — Creation of virtual environments	1697
29.4	<code>zipapp</code> — Manage executable Python zip archives	1706
30	Python Runtime Services	1713
30.1	<code>sys</code> — System-specific parameters and functions	1713
30.2	<code>sysconfig</code> — Provide access to Python’s configuration information	1732
30.3	<code>builtins</code> — Built-in objects	1736
30.4	<code>__main__</code> — Top-level script environment	1736
30.5	<code>warnings</code> — Warning control	1737
30.6	<code>dataclasses</code> — Data Classes	1743
30.7	<code>contextlib</code> — Utilities for with-statement contexts	1751
30.8	<code>abc</code> — Abstract Base Classes	1764
30.9	<code>atexit</code> — Exit handlers	1768
30.10	<code>traceback</code> — Print or retrieve a stack traceback	1770
30.11	<code>__future__</code> — Future statement definitions	1776
30.12	<code>gc</code> — Garbage Collector interface	1777
30.13	<code>inspect</code> — Inspect live objects	1780
30.14	<code>site</code> — Site-specific configuration hook	1796
31	Custom Python Interpreters	1801
31.1	<code>code</code> — Interpreter base classes	1801
31.2	<code>codeop</code> — Compile Python code	1803
32	Importing Modules	1805
32.1	<code>zipimport</code> — Import modules from Zip archives	1805
32.2	<code>pkgutil</code> — Package extension utility	1807
32.3	<code>modulefinder</code> — Find modules used by a script	1810
32.4	<code>runpy</code> — Locating and executing Python modules	1811
32.5	<code>importlib</code> — The implementation of <code>import</code>	1813
32.6	Using <code>importlib.metadata</code>	1833
33	Python Language Services	1837
33.1	<code>parser</code> — Access Python parse trees	1837
33.2	<code>ast</code> — Abstract Syntax Trees	1841
33.3	<code>symtable</code> — Access to the compiler’s symbol tables	1847
33.4	<code>symbol</code> — Constants used with Python parse trees	1850
33.5	<code>token</code> — Constants used with Python parse trees	1850
33.6	<code>keyword</code> — Testing for Python keywords	1854
33.7	<code>tokenize</code> — Tokenizer for Python source	1854
33.8	<code>tabnanny</code> — Detection of ambiguous indentation	1858
33.9	<code>pyclbr</code> — Python module browser support	1859
33.10	<code>py_compile</code> — Compile Python source files	1860
33.11	<code>compileall</code> — Byte-compile Python libraries	1862
33.12	<code>dis</code> — Disassembler for Python bytecode	1865
33.13	<code>pickletools</code> — Tools for pickle developers	1879

34 Miscellaneous Services	1881
34.1 <code>formatter</code> — Generic output formatting	1881
35 MS Windows Specific Services	1885
35.1 <code>msilib</code> — Read and write Microsoft Installer files	1885
35.2 <code>msvcrt</code> — Useful routines from the MS VC++ runtime	1891
35.3 <code>winreg</code> — Windows registry access	1893
35.4 <code>winsound</code> — Sound-playing interface for Windows	1902
36 Unix Specific Services	1905
36.1 <code>posix</code> — The most common POSIX system calls	1905
36.2 <code>pwd</code> — The password database	1906
36.3 <code>spwd</code> — The shadow password database	1907
36.4 <code>grp</code> — The group database	1908
36.5 <code>crypt</code> — Function to check Unix passwords	1908
36.6 <code>termios</code> — POSIX style tty control	1910
36.7 <code>tty</code> — Terminal control functions	1912
36.8 <code>pty</code> — Pseudo-terminal utilities	1912
36.9 <code>fcntl</code> — The <code>fcntl</code> and <code>ioctl</code> system calls	1914
36.10 <code>pipes</code> — Interface to shell pipelines	1916
36.11 <code>resource</code> — Resource usage information	1917
36.12 <code>nis</code> — Interface to Sun's NIS (Yellow Pages)	1921
36.13 <code>syslog</code> — Unix syslog library routines	1922
37 Superseded Modules	1925
37.1 <code>optparse</code> — Parser for command line options	1925
37.2 <code>imp</code> — Access the import internals	1952
38 Undocumented Modules	1957
38.1 Platform specific modules	1957
A Glossary	1959
B About these documents	1973
B.1 Contributors to the Python Documentation	1973
C History and License	1975
C.1 History of the software	1975
C.2 Terms and conditions for accessing or otherwise using Python	1976
C.3 Licenses and Acknowledgements for Incorporated Software	1979
D Copyright	1993
Bibliography	1995
Python Module Index	1997
Index	2001

While `reference-index` describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions.

Python’s standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

The Python installers for the Windows platform usually include the entire standard library and often also include many additional components. For Unix-like operating systems Python is normally provided as a collection of packages, so it may be necessary to use the packaging tools provided with the operating system to obtain some or all of the optional components.

In addition to the standard library, there is a growing collection of several thousand components (from individual programs and modules to packages and entire application development frameworks), available from the [Python Package Index](#).

**CHAPTER
ONE**

INTRODUCTION

The “Python library” contains several different kinds of components.

It contains data types that would normally be considered part of the “core” of a language, such as numbers and lists. For these types, the Python language core defines the form of literals and places some constraints on their semantics, but does not fully define the semantics. (On the other hand, the language core does define syntactic properties like the spelling and priorities of operators.)

The library also contains built-in functions and exceptions — objects that can be used by all Python code without the need of an `import` statement. Some of these are defined by the core language, but many are not essential for the core semantics and are only described here.

The bulk of the library, however, consists of a collection of modules. There are many ways to dissect this collection. Some modules are written in C and built in to the Python interpreter; others are written in Python and imported in source form. Some modules provide interfaces that are highly specific to Python, like printing a stack trace; some provide interfaces that are specific to particular operating systems, such as access to specific hardware; others provide interfaces that are specific to a particular application domain, like the World Wide Web. Some modules are available in all versions and ports of Python; others are only available when the underlying system supports or requires them; yet others are available only when a particular configuration option was chosen at the time when Python was compiled and installed.

This manual is organized “from the inside out:” it first describes the built-in functions, data types and exceptions, and finally the modules, grouped in chapters of related modules.

This means that if you start reading this manual from the start, and skip to the next chapter when you get bored, you will get a reasonable overview of the available modules and application areas that are supported by the Python library. Of course, you don’t *have* to read it like a novel — you can also browse the table of contents (in front of the manual), or look for a specific function, module or term in the index (in the back). And finally, if you enjoy learning about random subjects, you choose a random page number (see module `random`) and read a section or two. Regardless of the order in which you read the sections of this manual, it helps to start with chapter *Built-in Functions*, as the remainder of the manual assumes familiarity with this material.

Let the show begin!

1.1 Notes on availability

- An “Availability: Unix” note means that this function is commonly found on Unix systems. It does not make any claims about its existence on a specific operating system.
- If not separately noted, all functions that claim “Availability: Unix” are supported on Mac OS X, which builds on a Unix core.

CHAPTER TWO

BUILT-IN FUNCTIONS

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

Built-in Functions				
<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	

`abs` (*x*)

Return the absolute value of a number. The argument may be an integer or a floating point number. If the argument is a complex number, its magnitude is returned. If *x* defines `__abs__()`, `abs(x)` returns *x*.`__abs__()`.

`all` (*iterable*)

Return True if all elements of the *iterable* are true (or if the iterable is empty). Equivalent to:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

`any` (*iterable*)

Return True if any element of the *iterable* is true. If the iterable is empty, return False. Equivalent to:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

`ascii(object)`

As `repr()`, return a string containing a printable representation of an object, but escape the non-ASCII characters in the string returned by `repr()` using `\x`, `\u` or `\U` escapes. This generates a string similar to that returned by `repr()` in Python 2.

`bin(x)`

Convert an integer number to a binary string prefixed with “0b”. The result is a valid Python expression. If `x` is not a Python `int` object, it has to define an `__index__()` method that returns an integer. Some examples:

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

If prefix “0b” is desired or not, you can use either of the following ways.

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

See also `format()` for more information.

`class bool([x])`

Return a Boolean value, i.e. one of `True` or `False`. `x` is converted using the standard *truth testing procedure*. If `x` is false or omitted, this returns `False`; otherwise it returns `True`. The `bool` class is a subclass of `int` (see *Numeric Types — int, float, complex*). It cannot be subclassed further. Its only instances are `False` and `True` (see *Boolean Values*).

Changed in version 3.7: `x` is now a positional-only parameter.

`breakpoint(*args, **kws)`

This function drops you into the debugger at the call site. Specifically, it calls `sys.breakpointhook()`, passing `args` and `kws` straight through. By default, `sys.breakpointhook()` calls `pdb.set_trace()` expecting no arguments. In this case, it is purely a convenience function so you don’t have to explicitly import `pdb` or type as much code to enter the debugger. However, `sys.breakpointhook()` can be set to some other function and `breakpoint()` will automatically call that, allowing you to drop into the debugger of choice.

Raises an *auditing event* `builtins.breakpoint` with argument `breakpointhook`.

New in version 3.7.

`class bytearray([source[, encoding[, errors]]])`

Return a new array of bytes. The `bytearray` class is a mutable sequence of integers in the range $0 \leq x < 256$. It has most of the usual methods of mutable sequences, described in *Mutable Sequence Types*, as well as most methods that the `bytes` type has, see *Bytes and Bytearray Operations*.

The optional `source` parameter can be used to initialize the array in a few different ways:

- If it is a `string`, you must also give the `encoding` (and optionally, `errors`) parameters; `bytearray()` then converts the string to bytes using `str.encode()`.
- If it is an `integer`, the array will have that size and will be initialized with null bytes.
- If it is an object conforming to the `buffer` interface, a read-only buffer of the object will be used to initialize the bytes array.
- If it is an `iterable`, it must be an iterable of integers in the range $0 \leq x < 256$, which are used as the initial contents of the array.

Without an argument, an array of size 0 is created.

See also [Binary Sequence Types — bytes, bytearray, memoryview](#) and [Bytearray Objects](#).

`class bytes ([source[, encoding[, errors]]])`

Return a new “bytes” object, which is an immutable sequence of integers in the range $0 \leq x < 256$. `bytes` is an immutable version of `bytearray` – it has the same non-mutating methods and the same indexing and slicing behavior.

Accordingly, constructor arguments are interpreted as for `bytearray()`.

Bytes objects can also be created with literals, see strings.

See also [Binary Sequence Types — bytes, bytearray, memoryview, Bytes Objects](#), and [Bytes and Bytearray Operations](#).

`callable (object)`

Return `True` if the `object` argument appears callable, `False` if not. If this returns `True`, it is still possible that a call fails, but if it is `False`, calling `object` will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a `__call__()` method.

New in version 3.2: This function was first removed in Python 3.0 and then brought back in Python 3.2.

`chr (i)`

Return the string representing a character whose Unicode code point is the integer `i`. For example, `chr(97)` returns the string '`a`', while `chr(8364)` returns the string '`€`'. This is the inverse of `ord()`.

The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16). `ValueError` will be raised if `i` is outside that range.

`@classmethod`

Transform a method into a class method.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C:
    @classmethod
    def f(cls, arg1, arg2, ...): ...
```

The `@classmethod` form is a function `decorator` – see function for details.

A class method can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see `staticmethod()`.

For more information on class methods, see types.

`compile (source, filename, mode=0, dont_inherit=False, optimize=-1)`

Compile the `source` into a code or AST object. Code objects can be executed by `exec()` or `eval()`. `source` can either be a normal string, a byte string, or an AST object. Refer to the `ast` module documentation for information on how to work with AST objects.

The `filename` argument should give the file from which the code was read; pass some recognizable value if it wasn't read from a file ('<string>' is commonly used).

The `mode` argument specifies what kind of code must be compiled; it can be '`exec`' if `source` consists of a sequence of statements, '`eval`' if it consists of a single expression, or '`single`' if it consists of a single interactive statement (in the latter case, expression statements that evaluate to something other than `None` will be printed).

The optional arguments `flags` and `dont_inherit` control which future statements affect the compilation of `source`. If neither is present (or both are zero) the code is compiled with those future statements that are in effect in the code that is calling `compile()`. If the `flags` argument is given and `dont_inherit` is not (or is zero) then the future

statements specified by the *flags* argument are used in addition to those that would be used anyway. If *dont_inherit* is a non-zero integer then the *flags* argument is it – the future statements in effect around the call to `compile` are ignored.

Future statements are specified by bits which can be bitwise ORed together to specify multiple statements. The bitfield required to specify a given feature can be found as the `compiler_flag` attribute on the `_Feature` instance in the `__future__` module.

The optional argument *flags* also controls whether the compiled source is allowed to contain top-level `await`, `async for` and `async with`. When the bit `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` is set, the return code object has `CO_COROUTINE` set in `co_code`, and can be interactively executed via `await eval(code_object)`.

The argument *optimize* specifies the optimization level of the compiler; the default value of `-1` selects the optimization level of the interpreter as given by `-O` options. Explicit levels are `0` (no optimization; `__debug__` is true), `1` (asserts are removed, `__debug__` is false) or `2` (docstrings are removed too).

This function raises `SyntaxError` if the compiled source is invalid, and `ValueError` if the source contains null bytes.

If you want to parse Python code into its AST representation, see `ast.parse()`.

Raises an *auditing event* `compile` with arguments `source` and `filename`. This event may also be raised by implicit compilation.

Note: When compiling a string with multi-line code in '`'single'`' or '`'eval'`' mode, input must be terminated by at least one newline character. This is to facilitate detection of incomplete and complete statements in the `code` module.

Warning: It is possible to crash the Python interpreter with a sufficiently large/complex string when compiling to an AST object due to stack depth limitations in Python's AST compiler.

Changed in version 3.2: Allowed use of Windows and Mac newlines. Also input in '`'exec'`' mode does not have to end in a newline anymore. Added the *optimize* parameter.

Changed in version 3.5: Previously, `TypeError` was raised when null bytes were encountered in `source`.

New in version 3.8: `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` can now be passed in `flags` to enable support for top-level `await`, `async for`, and `async with`.

```
class complex([real[, imag]])
```

Return a complex number with the value `real + imag*1j` or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including `complex`). If `imag` is omitted, it defaults to zero and the constructor serves as a numeric conversion like `int` and `float`. If both arguments are omitted, returns `0j`.

For a general Python object `x`, `complex(x)` delegates to `x.__complex__()`. If `__complex__()` is not defined then it falls back to `__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

Note: When converting from a string, the string must not contain whitespace around the central `+` or `-` operator. For example, `complex('1+2j')` is fine, but `complex('1 + 2j')` raises `ValueError`.

The complex type is described in [Numeric Types — `int`, `float`, `complex`](#).

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.8: Falls back to `__index__()` if `__complex__()` and `__float__()` are not defined.

`delattr(object, name)`

This is a relative of `setattr()`. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(x, 'foobar')` is equivalent to `del x foobar`.

```
class dict(**kwargs)
class dict(mapping, **kwargs)
class dict(iterable, **kwargs)
```

Create a new dictionary. The `dict` object is the dictionary class. See `dict` and *Mapping Types — dict* for documentation about this class.

For other containers see the built-in `list`, `set`, and `tuple` classes, as well as the `collections` module.

`dir([object])`

Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object.

If the object has a method named `__dir__()`, this method will be called and must return the list of attributes. This allows objects that implement a custom `__getattr__()` or `__getattribute__()` function to customize the way `dir()` reports their attributes.

If the object does not provide `__dir__()`, the function tries its best to gather information from the object's `__dict__` attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom `__getattr__()`.

The default `dir()` mechanism behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete, information:

- If the object is a module object, the list contains the names of the module's attributes.
- If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
- Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes.

The resulting list is sorted alphabetically. For example:

```
>>> import struct
>>> dir()    # show the names in the module namespace
['__builtins__', '__name__', 'struct']
>>> dir(struct)  # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '__clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
>>> s = Shape()
>>> dir(s)
['area', 'location', 'perimeter']
```

Note: Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its

detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

divmod(*a, b*)

Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division. With mixed operand types, the rules for binary arithmetic operators apply. For integers, the result is the same as (*a* // *b*, *a* % *b*). For floating point numbers the result is (*q*, *a* % *b*), where *q* is usually `math.floor(a / b)` but may be 1 less than that. In any case *q* * *b* + *a* % *b* is very close to *a*, if *a* % *b* is non-zero it has the same sign as *b*, and $0 \leq \text{abs}(a \% b) < \text{abs}(b)$.

enumerate(*iterable*, *start*=0)

Return an enumerate object. *iterable* must be a sequence, an [iterator](#), or some other object which supports iteration. The `__next__()` method of the iterator returned by `enumerate()` returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *iterable*.

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Equivalent to:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

eval(*expression*[, *globals*[, *locals*]])

The arguments are a string and optional *globals* and *locals*. If provided, *globals* must be a dictionary. If provided, *locals* can be any mapping object.

The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* dictionaries as global and local namespace. If the *globals* dictionary is present and does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key before *expression* is parsed. This means that *expression* normally has full access to the standard `builtins` module and restricted environments are propagated. If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed with the *globals* and *locals* in the environment where `eval()` is called. Note, `eval()` does not have access to the *nested scopes* (non-locals) in the enclosing environment.

The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```
>>> x = 1
>>> eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as those created by `compile()`). In this case pass a code object instead of a string. If the code object has been compiled with 'exec' as the *mode* argument, `eval()`'s return value will be `None`.

Hints: dynamic execution of statements is supported by the `exec()` function. The `globals()` and `locals()` functions returns the current global and local dictionary, respectively, which may be useful to pass around for use by `eval()` or `exec()`.

See `ast.literal_eval()` for a function that can safely evaluate strings with expressions containing only literals.

Raises an *auditing event* `exec` with the code object as the argument. Code compilation events may also be raised.

`exec(object[, globals[, locals]])`

This function supports dynamic execution of Python code. *object* must be either a string or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs).¹ If it is a code object, it is simply executed. In all cases, the code that's executed is expected to be valid as file input (see the section “File input” in the Reference Manual). Be aware that the `return` and `yield` statements may not be used outside of function definitions even within the context of code passed to the `exec()` function. The return value is `None`.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only *globals* is provided, it must be a dictionary (and not a subclass of dictionary), which will be used for both the global and the local variables. If *globals* and *locals* are given, they are used for the global and local variables, respectively. If provided, *locals* can be any mapping object. Remember that at module level, *globals* and *locals* are the same dictionary. If `exec` gets two separate objects as *globals* and *locals*, the code will be executed as if it were embedded in a class definition.

If the *globals* dictionary does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key. That way you can control what builtins are available to the executed code by inserting your own `__builtins__` dictionary into *globals* before passing it to `exec()`.

Raises an *auditing event* `exec` with the code object as the argument. Code compilation events may also be raised.

Note: The built-in functions `globals()` and `locals()` return the current global and local dictionary, respectively, which may be useful to pass around for use as the second and third argument to `exec()`.

Note: The default *locals* act as described for function `locals()` below: modifications to the default *locals* dictionary should not be attempted. Pass an explicit *locals* dictionary if you need to see effects of the code on *locals* after function `exec()` returns.

`filter(function, iterable)`

Construct an iterator from those elements of *iterable* for which *function* returns true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *function* is `None`, the identity function is assumed, that is, all elements of *iterable* that are false are removed.

Note that `filter(function, iterable)` is equivalent to the generator expression `(item for item in iterable if function(item))` if *function* is not `None` and `(item for item in iterable if item)` if *function* is `None`.

See `itertools.filterfalse()` for the complementary function that returns elements of *iterable* for which *function* returns false.

`class float([x])`

Return a floating point number constructed from a number or string *x*.

If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be '+' or '-'; a '+' sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or a positive or negative infinity. More precisely, the input must conform to the following grammar after leading and trailing whitespace characters are removed:

```
sign ::= "+" | "-"
```

¹ Note that the parser only accepts the Unix-style end of line convention. If you are reading the code from a file, make sure to use newline conversion mode to convert Windows or Mac-style newlines.

```

infinity      ::=  "Infinity" | "inf"
nan          ::=  "nan"
numeric_value ::=  floatnumber | infinity | nan
numeric_string ::=  [sign] numeric_value

```

Here `floatnumber` is the form of a Python floating-point literal, described in [floating](#). Case is not significant, so, for example, “`inf`”, “`Inf`”, “`INFINITY`” and “`iNfINity`” are all acceptable spellings for positive infinity.

Otherwise, if the argument is an integer or a floating point number, a floating point number with the same value (within Python’s floating point precision) is returned. If the argument is outside the range of a Python float, an [`OverflowError`](#) will be raised.

For a general Python object `x`, `float(x)` delegates to `x.__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

If no argument is given, `0.0` is returned.

Examples:

```

>>> float('+1.23')
1.23
>>> float(' -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf

```

The float type is described in [Numeric Types — `int`, `float`, `complex`](#).

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.7: `x` is now a positional-only parameter.

Changed in version 3.8: Falls back to `__index__()` if `__float__()` is not defined.

`format(value[, format_spec])`

Convert a `value` to a “formatted” representation, as controlled by `format_spec`. The interpretation of `format_spec` will depend on the type of the `value` argument, however there is a standard formatting syntax that is used by most built-in types: [Format Specification Mini-Language](#).

The default `format_spec` is an empty string which usually gives the same effect as calling `str(value)`.

A call to `format(value, format_spec)` is translated to `type(value).__format__(value, format_spec)` which bypasses the instance dictionary when searching for the value’s `__format__()` method. A [`TypeError`](#) exception is raised if the method search reaches `object` and the `format_spec` is non-empty, or if either the `format_spec` or the return value are not strings.

Changed in version 3.4: `object().__format__(format_spec)` raises [`TypeError`](#) if `format_spec` is not an empty string.

`class frozenset([iterable])`

Return a new `frozenset` object, optionally with elements taken from `iterable`. `frozenset` is a built-in class. See `frozenset` and [Set Types — `set`, `frozenset`](#) for documentation about this class.

For other containers see the built-in `set`, `list`, `tuple`, and `dict` classes, as well as the `collections` module.

`getattr(object, name[, default])`

Return the value of the named attribute of `object`. `name` must be a string. If the string is the name of one of the ob-

ject's attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribute does not exist, `default` is returned if provided, otherwise `AttributeError` is raised.

`globals()`

Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

`hasattr(object, name)`

The arguments are an object and a string. The result is `True` if the string is the name of one of the object's attributes, `False` if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an `AttributeError` or not.)

`hash(object)`

Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

Note: For objects with custom `__hash__()` methods, note that `hash()` truncates the return value based on the bit width of the host machine. See `__hash__()` for details.

`help([object])`

Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated.

Note that if a slash(/) appears in the parameter list of a function, when invoking `help()`, it means that the parameters prior to the slash are positional-only. For more info, see the FAQ entry on positional-only parameters.

This function is added to the built-in namespace by the `site` module.

Changed in version 3.4: Changes to `pydoc` and `inspect` mean that the reported signatures for callables are now more comprehensive and consistent.

`hex(x)`

Convert an integer number to a lowercase hexadecimal string prefixed with "0x". If `x` is not a Python `int` object, it has to define an `__index__()` method that returns an integer. Some examples:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

If you want to convert an integer number to an uppercase or lower hexadecimal string with prefix or not, you can use either of the following ways:

```
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
>>> f'{255:#x}', f'{255:x}', f'{255:X}'
('0xff', 'ff', 'FF')
```

See also `format()` for more information.

See also `int()` for converting a hexadecimal string to an integer using a base of 16.

Note: To obtain a hexadecimal string representation for a float, use the `float.hex()` method.

`id`(*object*)

Return the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.

CPython implementation detail: This is the address of the object in memory.

`input`(*[prompt]*)

If the `prompt` argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, `EOFError` is raised. Example:

```
>>> s = input('---> ')
---> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the `readline` module was loaded, then `input()` will use it to provide elaborate line editing and history features.

Raises an *auditing event* `builtins.input` with argument `prompt` before reading input

Raises an auditing event `builtins.input/result` with the result after successfully reading input.

`class int([x])`**`class int(x, base=10)`**

Return an integer object constructed from a number or string *x*, or return 0 if no arguments are given. If *x* defines `__int__()`, `int(x)` returns *x*.`__int__()`. If *x* defines `__index__()`, it returns *x*.`__index__()`. If *x* defines `__trunc__()`, it returns *x*.`__trunc__()`. For floating point numbers, this truncates towards zero.

If *x* is not a number or if *base* is given, then *x* must be a string, `bytes`, or `bytearray` instance representing an integer literal in radix *base*. Optionally, the literal can be preceded by + or - (with no space in between) and surrounded by whitespace. A base-n literal consists of the digits 0 to n-1, with a to z (or A to Z) having values 10 to 35. The default *base* is 10. The allowed values are 0 and 2–36. Base-2, -8, and -16 literals can be optionally prefixed with 0b/0B, 0o/0O, or 0x/0X, as with integer literals in code. Base 0 means to interpret exactly as a code literal, so that the actual base is 2, 8, 10, or 16, and so that `int('010', 0)` is not legal, while `int('010')` is, as well as `int('010', 8)`.

The integer type is described in *Numeric Types — int, float, complex*.

Changed in version 3.4: If *base* is not an instance of `int` and the *base* object has a `base.__index__` method, that method is called to obtain an integer for the base. Previous versions used `base.__int__` instead of `base.__index__`.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.7: *x* is now a positional-only parameter.

Changed in version 3.8: Falls back to `__index__()` if `__int__()` is not defined.

`isinstance`(*object, classinfo*)

Return `True` if the *object* argument is an instance of the *classinfo* argument, or of a (direct, indirect or *virtual*) subclass thereof. If *object* is not an object of the given type, the function always returns `False`. If *classinfo* is a tuple of type objects (or recursively, other such tuples), return `True` if *object* is an instance of any of the types. If *classinfo* is not a type or tuple of types and such tuples, a `TypeError` exception is raised.

`issubclass`(*class, classinfo*)

Return `True` if *class* is a subclass (direct, indirect or *virtual*) of *classinfo*. A class is considered a subclass of itself.

classinfo may be a tuple of class objects, in which case every entry in *classinfo* will be checked. In any other case, a *TypeError* exception is raised.

`iter(object[, sentinel])`

Return an *iterator* object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, *object* must be a collection object which supports the iteration protocol (the `__iter__()` method), or it must support the sequence protocol (the `__getitem__()` method with integer arguments starting at 0). If it does not support either of those protocols, *TypeError* is raised. If the second argument, *sentinel*, is given, then *object* must be a callable object. The iterator created in this case will call *object* with no arguments for each call to its `__next__()` method; if the value returned is equal to *sentinel*, *StopIteration* will be raised, otherwise the value will be returned.

See also *Iterator Types*.

One useful application of the second form of `iter()` is to build a block-reader. For example, reading fixed-width blocks from a binary database file until the end of file is reached:

```
from functools import partial
with open('mydata.db', 'rb') as f:
    for block in iter(partial(f.read, 64), b''):
        process_block(block)
```

`len(s)`

Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

`class list([iterable])`

Rather than being a function, `list` is actually a mutable sequence type, as documented in *Lists and Sequence Types — list, tuple, range*.

`locals()`

Update and return a dictionary representing the current local symbol table. Free variables are returned by `locals()` when it is called in function blocks, but not in class blocks. Note that at the module level, `locals()` and `globals()` are the same dictionary.

Note: The contents of this dictionary should not be modified; changes may not affect the values of local and free variables used by the interpreter.

`map(function, iterable, ...)`

Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see `itertools.starmap()`.

`max(iterable, *[, key, default])`

`max(arg1, arg2, *args[, key])`

Return the largest item in an iterable or the largest of two or more arguments.

If one positional argument is provided, it should be an *iterable*. The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for `list.sort()`. The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a *ValueError* is raised.

If multiple items are maximal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc, reverse=True)[0]` and `heapq.nlargest(1, iterable, key=keyfunc)`.

New in version 3.4: The *default* keyword-only argument.

Changed in version 3.8: The *key* can be None.

class memoryview(*obj*)

Return a “memory view” object created from the given argument. See [Memory Views](#) for more information.

min(*iterable*, *[, *key*, *default*])

min(*arg1*, *arg2*, **args*[, *key*])

Return the smallest item in an iterable or the smallest of two or more arguments.

If one positional argument is provided, it should be an *iterable*. The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for [*list.sort\(\)*](#). The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a [ValueError](#) is raised.

If multiple items are minimal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as [*sorted\(iterable, key=keyfunc\)\[0\]*](#) and [*heapq.nsmallest\(1, iterable, key=keyfunc\)*](#).

New in version 3.4: The *default* keyword-only argument.

Changed in version 3.8: The *key* can be None.

next(*iterator*[, *default*])

Retrieve the next item from the *iterator* by calling its [*__next__\(\)*](#) method. If *default* is given, it is returned if the iterator is exhausted, otherwise [StopIteration](#) is raised.

class object

Return a new featureless object. [*object*](#) is a base for all classes. It has the methods that are common to all instances of Python classes. This function does not accept any arguments.

Note: [*object*](#) does *not* have a [*__dict__*](#), so you can't assign arbitrary attributes to an instance of the [*object*](#) class.

oct(*x*)

Convert an integer number to an octal string prefixed with “0o”. The result is a valid Python expression. If *x* is not a Python [*int*](#) object, it has to define an [*__index__\(\)*](#) method that returns an integer. For example:

```
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'
```

If you want to convert an integer number to octal string either with prefix “0o” or not, you can use either of the following ways.

```
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
>>> f'{10:#o}', f'{10:o}'
('0o12', '12')
```

See also [*format\(\)*](#) for more information.

open (*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

Open *file* and return a corresponding *file object*. If the file cannot be opened, an *OSError* is raised.

file is a *path-like object* giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless *closefd* is set to *False*.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to '*r*' which means open for reading in text mode. Other common values are '*w*' for writing (truncating the file if it already exists), '*x*' for exclusive creation and '*a*' for appending (which on *some* Unix systems, means that *all* writes append to the end of the file regardless of the current seek position). In text mode, if *encoding* is not specified the encoding used is platform dependent: `locale.getpreferredencoding(False)` is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave *encoding* unspecified.) The available modes are:

Character	Meaning
' <i>r</i> '	open for reading (default)
' <i>w</i> '	open for writing, truncating the file first
' <i>x</i> '	open for exclusive creation, failing if the file already exists
' <i>a</i> '	open for writing, appending to the end of the file if it exists
' <i>b</i> '	binary mode
' <i>t</i> '	text mode (default)
'+'	open for updating (reading and writing)

The default mode is '*r*' (open for reading text, synonym of '*rt*'). Modes '*w+*' and '*w+b*' open and truncate the file. Modes '*r+*' and '*r+b*' open the file with no truncation.

As mentioned in the *Overview*, Python distinguishes between binary and text I/O. Files opened in binary mode (including '*b*' in the *mode* argument) return contents as *bytes* objects without any decoding. In text mode (the default, or when '*t*' is included in the *mode* argument), the contents of the file are returned as *str*, the bytes having been first decoded using a platform-dependent encoding or using the specified *encoding* if given.

There is an additional mode character permitted, '*U*', which no longer has any effect, and is considered deprecated. It previously enabled *universal newlines* in text mode, which became the default behaviour in Python 3.0. Refer to the documentation of the *newline* parameter for further details.

Note: Python doesn't depend on the underlying operating system's notion of text files; all the processing is done by Python itself, and is therefore platform-independent.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size in bytes of a fixed-size chunk buffer. When no *buffering* argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on `io.DEFAULT_BUFFER_SIZE`. On many systems, the buffer will typically be 4096 or 8192 bytes long.
- "Interactive" text files (files for which `isatty()` returns `True`) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever `locale.getpreferredencoding()` returns), but any *text encoding* supported by Python can be used. See the `codecs` module for the list of supported encodings.

errors is an optional string that specifies how encoding and decoding errors are to be handled—this cannot be used in binary mode. A variety of standard error handlers are available (listed under *Error Handlers*), though any error

handling name that has been registered with `codecs.register_error()` is also valid. The standard names include:

- 'strict' to raise a `ValueError` exception if there is an encoding error. The default value of `None` has the same effect.
- 'ignore' ignores errors. Note that ignoring encoding errors can lead to data loss.
- 'replace' causes a replacement marker (such as '?) to be inserted where there is malformed data.
- 'surrogateescape' will represent any incorrect bytes as code points in the Unicode Private Use Area ranging from U+DC80 to U+DCFF. These private code points will then be turned back into the same bytes when the `surrogateescape` error handler is used when writing data. This is useful for processing files in an unknown encoding.
- 'xmlcharrefreplace' is only supported when writing to a file. Characters not supported by the encoding are replaced with the appropriate XML character reference &#nnn;.
- 'backslashreplace' replaces malformed data by Python's backslashed escape sequences.
- 'namereplace' (also only supported when writing) replaces unsupported characters with \N{...} escape sequences.

`newline` controls how *universal newlines* mode works (it only applies to text mode). It can be `None`, '`'`', '`\n`', '`\r`', and '`\r\n`'. It works as follows:

- When reading input from the stream, if `newline` is `None`, universal newlines mode is enabled. Lines in the input can end in '`\n`', '`\r`', or '`\r\n`', and these are translated into '`\n`' before being returned to the caller. If it is '`'`', universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- When writing output to the stream, if `newline` is `None`, any '`\n`' characters written are translated to the system default line separator, `os.linesep`. If `newline` is '`'` or '`\n`', no translation takes place. If `newline` is any of the other legal values, any '`\n`' characters written are translated to the given string.

If `closefd` is `False` and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given `closefd` must be `True` (the default) otherwise an error will be raised.

A custom opener can be used by passing a callable as `opener`. The underlying file descriptor for the file object is then obtained by calling `opener` with `(file, flags)`. `opener` must return an open file descriptor (passing `os.open` as `opener` results in functionality similar to passing `None`).

The newly created file is *non-inheritable*.

The following example uses the `dir_fd` parameter of the `os.open()` function to open a file relative to a given directory:

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spamspam.txt', file=f)
...
>>> os.close(dir_fd) # don't leak a file descriptor
```

The type of `file object` returned by the `open()` function depends on the mode. When `open()` is used to open a file in a text mode ('`w`', '`r`', '`wt`', '`rt`', etc.), it returns a subclass of `io.TextIOWrapper` (specifically `io.TextIOWrapper`). When used to open a file in a binary mode with buffering, the returned class is a subclass of

`io.BufferedIOBase`. The exact class varies: in read binary mode, it returns an `io.BufferedReader`; in write binary and append binary modes, it returns an `io.BufferedWriter`, and in read/write mode, it returns an `io.BufferedRandom`. When buffering is disabled, the raw stream, a subclass of `io.RawIOBase`, `io.FileIO`, is returned.

See also the file handling modules, such as, `fileinput`, `io` (where `open()` is declared), `os`, `os.path`, `tempfile`, and `shutil`.

Raises an *auditing event* open with arguments `file`, `mode`, `flags`.

The `mode` and `flags` arguments may have been modified or inferred from the original call.

Changed in version 3.3:

- The `opener` parameter was added.
- The '`x`' mode was added.
- `IOError` used to be raised, it is now an alias of `OSError`.
- `FileExistsError` is now raised if the file opened in exclusive creation mode ('`x`') already exists.

Changed in version 3.4:

- The file is now non-inheritable.

Deprecated since version 3.4, will be removed in version 3.9: The '`U`' mode.

Changed in version 3.5:

- If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see [PEP 475](#) for the rationale).
- The '`namereplace`' error handler was added.

Changed in version 3.6:

- Support added to accept objects implementing `os.PathLike`.
- On Windows, opening a console buffer may return a subclass of `io.RawIOBase` other than `io.FileIO`.

`ord(c)`

Given a string representing one Unicode character, return an integer representing the Unicode code point of that character. For example, `ord('a')` returns the integer 97 and `ord('€')` (Euro sign) returns 8364. This is the inverse of `chr()`.

`pow(base, exp[, mod])`

Return `base` to the power `exp`; if `mod` is present, return `base` to the power `exp`, modulo `mod` (computed more efficiently than `pow(base, exp) % mod`). The two-argument form `pow(base, exp)` is equivalent to using the power operator: `base**exp`.

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For `int` operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns 100, but `10**-2` returns 0.01.

For `int` operands `base` and `exp`, if `mod` is present, `mod` must also be of integer type and `mod` must be nonzero. If `mod` is present and `exp` is negative, `base` must be relatively prime to `mod`. In that case, `pow(inv_base, -exp, mod)` is returned, where `inv_base` is an inverse to `base` modulo `mod`.

Here's an example of computing an inverse for 38 modulo 97:

```
>>> pow(38, -1, mod=97)
23
>>> 23 * 38 % 97 == 1
True
```

Changed in version 3.8: For `int` operands, the three-argument form of `pow` now allows the second argument to be negative, permitting computation of modular inverses.

Changed in version 3.9: Allow keyword arguments. Formerly, only positional arguments were supported.

`print(*objects, sep='', end='\n', file=sys.stdout, flush=False)`

Print `objects` to the text stream `file`, separated by `sep` and followed by `end`. `sep`, `end`, `file` and `flush`, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by `sep` and followed by `end`. Both `sep` and `end` must be strings; they can also be `None`, which means to use the default values. If no `objects` are given, `print()` will just write `end`.

The `file` argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Whether output is buffered is usually determined by `file`, but if the `flush` keyword argument is true, the stream is forcibly flushed.

Changed in version 3.3: Added the `flush` keyword argument.

`class property(fget=None, fset=None, fdel=None, doc=None)`

Return a property attribute.

`fget` is a function for getting an attribute value. `fset` is a function for setting an attribute value. `fdel` is a function for deleting an attribute value. And `doc` creates a docstring for the attribute.

A typical use is to define a managed attribute `x`:

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

x = property(getx, setx, delx, "I'm the 'x' property.")
```

If `c` is an instance of `C`, `c.x` will invoke the getter, `c.x = value` will invoke the setter and `del c.x` the deleter.

If given, `doc` will be the docstring of the property attribute. Otherwise, the property will copy `fget`'s docstring (if it exists). This makes it possible to create read-only properties easily using `property()` as a `decorator`:

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

(continues on next page)

(continued from previous page)

```

self._voltage = 100000

@property
def voltage(self):
    """Get the current voltage."""
    return self._voltage

```

The `@property` decorator turns the `voltage()` method into a “getter” for a read-only attribute with the same name, and it sets the docstring for `voltage` to “Get the current voltage.”

A property object has `getter`, `setter`, and `deleter` methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```

class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x

```

This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original property (`x` in this case.)

The returned property object also has the attributes `fget`, `fset`, and `fdel` corresponding to the constructor arguments.

Changed in version 3.5: The docstrings of property objects are now writeable.

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

Rather than being a function, `range` is actually an immutable sequence type, as documented in [Ranges and Sequence Types — list, tuple, range](#).

`repr(object)`

Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`, otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a `__repr__()` method.

`reversed(seq)`

Return a reverse [iterator](#). `seq` must be an object which has a `__reversed__()` method or supports the sequence protocol (the `__len__()` method and the `__getitem__(index)` method with integer arguments starting at 0).

`round(number[, ndigits])`

Return `number` rounded to `ndigits` precision after the decimal point. If `ndigits` is omitted or is `None`, it returns the nearest integer to its input.

For the built-in types supporting `round()`, values are rounded to the closest multiple of 10 to the power minus `ndigits`; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are 0, and `round(1.5)` is 2). Any integer value is valid for `ndigits` (positive, zero, or negative). The return value is an integer if `ndigits` is omitted or `None`. Otherwise the return value has the same type as `number`.

For a general Python object `number`, `round` delegates to `number.__round__`.

Note: The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives 2.67 instead of the expected 2.68. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See [tut-fp-issues](#) for more information.

class `set`([*iterable*])

Return a new `set` object, optionally with elements taken from *iterable*. `set` is a built-in class. See [set](#) and [Set Types — `set`, `frozenset`](#) for documentation about this class.

For other containers see the built-in `frozenset`, `list`, `tuple`, and `dict` classes, as well as the `collections` module.

`setattr`(*object*, *name*, *value*)

This is the counterpart of `getattr()`. The arguments are an object, a string and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

class `slice`(*stop*)

class `slice`(*start*, *stop*[, *step*])

Return a `slice` object representing the set of indices specified by range (`start`, `stop`, `step`). The `start` and `step` arguments default to `None`. Slice objects have read-only data attributes `start`, `stop` and `step` which merely return the argument values (or their default). They have no other explicit functionality; however they are used by Numerical Python and other third party extensions. Slice objects are also generated when extended indexing syntax is used. For example: `a[start:stop:step]` or `a[start:stop, i]`. See [itertools.islice\(\)](#) for an alternate version that returns an iterator.

`sorted`(*iterable*, *, *key=None*, *reverse=False*)

Return a new sorted list from the items in *iterable*.

Has two optional arguments which must be specified as keyword arguments.

key specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

Use `functools.cmp_to_key()` to convert an old-style `cmp` function to a `key` function.

The built-in `sorted()` function is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

For sorting examples and a brief sorting tutorial, see [sortinghowto](#).

`@staticmethod`

Transform a method into a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...
```

The `@staticmethod` form is a function *decorator* – see function for details.

A static method can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`).

Static methods in Python are similar to those found in Java or C++. Also see `classmethod()` for a variant that is useful for creating alternate class constructors.

Like all decorators, it is also possible to call `staticmethod` as a regular function and do something with its result. This is needed in some cases where you need a reference to a function from a class body and you want to avoid the automatic transformation to instance method. For these cases, use this idiom:

```
class C:
    builtin_open = staticmethod(open)
```

For more information on static methods, see types.

```
class str(object=’’)
class str(object=b”, encoding=’utf-8’, errors=’strict’)
    Return a str version of object. See str() for details.
```

`str` is the built-in string `class`. For general information about strings, see *Text Sequence Type — str*.

`sum(iterable, /, start=0)`

Sums `start` and the items of an `iterable` from left to right and returns the total. The `iterable`'s items are normally numbers, and the `start` value is not allowed to be a string.

For some use cases, there are good alternatives to `sum()`. The preferred, fast way to concatenate a sequence of strings is by calling `''.join(sequence)`. To add floating point values with extended precision, see `math.fsum()`. To concatenate a series of iterables, consider using `itertools.chain()`.

Changed in version 3.8: The `start` parameter can be specified as a keyword argument.

`super([type[, object-or-type]])`

Return a proxy object that delegates method calls to a parent or sibling class of `type`. This is useful for accessing inherited methods that have been overridden in a class.

The `object-or-type` determines the *method resolution order* to be searched. The search starts from the class right after the `type`.

For example, if `__mro__` of `object-or-type` is `D -> B -> C -> A -> object` and the value of `type` is `B`, then `super()` searches `C -> A -> object`.

The `__mro__` attribute of the `object-or-type` lists the method resolution search order used by both `getattr()` and `super()`. The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

If the second argument is omitted, the super object returned is unbound. If the second argument is an object, `isinstance(obj, type)` must be true. If the second argument is a type, `issubclass(type2, type)` must be true (this is useful for classmethods).

There are two typical use cases for `super`. In a class hierarchy with single inheritance, `super` can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of `super` in other programming languages.

The second use case is to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement “diamond diagrams” where multiple base classes implement the same method. Good design dictates that this method have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

For both use cases, a typical superclass call looks like this:

```
class C(B):
    def method(self, arg):
        super().method(arg)      # This does the same thing as:
                                # super(C, self).method(arg)
```

In addition to method lookups, `super()` also works for attribute lookups. One possible use case for this is calling [descriptors](#) in a parent or sibling class.

Note that `super()` is implemented as part of the binding process for explicit dotted attribute lookups such as `super().__getitem__(name)`. It does so by implementing its own `__getattribute__()` method for searching classes in a predictable order that supports cooperative multiple inheritance. Accordingly, `super()` is undefined for implicit lookups using statements or operators such as `super()[name]`.

Also note that, aside from the zero argument form, `super()` is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references. The zero argument form only works inside a class definition, as the compiler fills in the necessary details to correctly retrieve the class being defined, as well as accessing the current instance for ordinary methods.

For practical suggestions on how to design cooperative classes using `super()`, see [guide to using super\(\)](#).

`class tuple([iterable])`

Rather than being a function, `tuple` is actually an immutable sequence type, as documented in [Tuples and Sequence Types — list, tuple, range](#).

`class type(object)`

`class type(name, bases, dict)`

With one argument, return the type of an `object`. The return value is a type object and generally the same object as returned by `object.__class__`.

The `isinstance()` built-in function is recommended for testing the type of an object, because it takes subclasses into account.

With three arguments, return a new type object. This is essentially a dynamic form of the `class` statement. The `name` string is the class name and becomes the `__name__` attribute; the `bases` tuple itemizes the base classes and becomes the `__bases__` attribute; and the `dict` dictionary is the namespace containing definitions for class body and is copied to a standard dictionary to become the `__dict__` attribute. For example, the following two statements create identical `type` objects:

```
>>> class X:
...     a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

See also [Type Objects](#).

Changed in version 3.6: Subclasses of `type` which don't override `type.__new__` may no longer use the one-argument form to get the type of an object.

`vars([object])`

Return the `__dict__` attribute for a module, class, instance, or any other object with a `__dict__` attribute.

Objects such as modules and instances have an updateable `__dict__` attribute; however, other objects may have write restrictions on their `__dict__` attributes (for example, classes use a `types.MappingProxyType` to prevent direct dictionary updates).

Without an argument, `vars()` acts like `locals()`. Note, the locals dictionary is only useful for reads since updates to the locals dictionary are ignored.

`zip(*iterables)`

Make an iterator that aggregates elements from each of the iterables.

Returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator. Equivalent to:

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using `zip(*[iter(s)]*n)`. This repeats the *same* iterator *n* times so that each output tuple has the result of *n* calls to the iterator. This has the effect of dividing the input into *n*-length chunks.

`zip()` should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables. If those values are important, use `itertools.zip_longest()` instead.

`zip()` in conjunction with the `*` operator can be used to unzip a list:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True
```

`__import__(name, globals=None, locals=None, fromlist=(), level=0)`

Note: This is an advanced function that is not needed in everyday Python programming, unlike `importlib.import_module()`.

This function is invoked by the `import` statement. It can be replaced (by importing the `builtins` module and assigning to `builtins.__import__()`) in order to change semantics of the `import` statement, but doing so is **strongly** discouraged as it is usually simpler to use import hooks (see [PEP 302](#)) to attain the same goals and does not cause issues with code which assumes the default import implementation is in use. Direct use of `__import__()` is also discouraged in favor of `importlib.import_module()`.

The function imports the module *name*, potentially using the given *globals* and *locals* to determine how to interpret the name in a package context. The *fromlist* gives the names of objects or submodules that should be imported from the module given by *name*. The standard implementation does not use its *locals* argument at all, and uses its *globals* only to determine the package context of the `import` statement.

level specifies whether to use absolute or relative imports. 0 (the default) means only perform absolute imports. Positive values for *level* indicate the number of parent directories to search relative to the directory of the module calling `__import__()` (see [PEP 328](#) for the details).

When the *name* variable is of the form `package.module`, normally, the top-level package (the name up till the

first dot) is returned, *not* the module named by *name*. However, when a non-empty *fromlist* argument is given, the module named by *name* is returned.

For example, the statement `import spam` results in bytecode resembling the following code:

```
spam = __import__('spam', globals(), locals(), [], 0)
```

The statement `import spam.ham` results in this call:

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

Note how `__import__()` returns the toplevel module here because this is the object that is bound to a name by the `import` statement.

On the other hand, the statement `from spam.ham import eggs, sausage as saus` results in

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], 0)
eggs = _temp.eggs
saus = _temp.sausage
```

Here, the `spam.ham` module is returned from `__import__()`. From this object, the names to import are retrieved and assigned to their respective names.

If you simply want to import a module (potentially within a package) by name, use `importlib.import_module()`.

Changed in version 3.3: Negative values for *level* are no longer supported (which also changes the default value to 0).

BUILT-IN CONSTANTS

A small number of constants live in the built-in namespace. They are:

False

The false value of the `bool` type. Assignments to `False` are illegal and raise a `SyntaxError`.

True

The true value of the `bool` type. Assignments to `True` are illegal and raise a `SyntaxError`.

None

The sole value of the type `NoneType`. `None` is frequently used to represent the absence of a value, as when default arguments are not passed to a function. Assignments to `None` are illegal and raise a `SyntaxError`.

NotImplemented

Special value which should be returned by the binary special methods (e.g. `__eq__()`, `__lt__()`, `__add__()`, `__rsub__()`, etc.) to indicate that the operation is not implemented with respect to the other type; may be returned by the in-place binary special methods (e.g. `__imul__()`, `__iand__()`, etc.) for the same purpose. Its truth value is true.

Note: When a binary (or in-place) method returns `NotImplemented` the interpreter will try the reflected operation on the other type (or some other fallback, depending on the operator). If all attempts return `NotImplemented`, the interpreter will raise an appropriate exception. Incorrectly returning `NotImplemented` will result in a misleading error message or the `NotImplemented` value being returned to Python code.

See [Implementing the arithmetic operations](#) for examples.

Note: `NotImplementedError` and `NotImplemented` are not interchangeable, even though they have similar names and purposes. See [NotImplementedError](#) for details on when to use it.

Ellipsis

The same as the ellipsis literal “`...`”. Special value used mostly in conjunction with extended slicing syntax for user-defined container data types.

`__debug__`

This constant is true if Python was not started with an `-O` option. See also the `assert` statement.

Note: The names `None`, `False`, `True` and `__debug__` cannot be reassigned (assignments to them, even as an attribute name, raise `SyntaxError`), so they can be considered “true” constants.

3.1 Constants added by the `site` module

The `site` module (which is imported automatically during startup, except if the `-S` command-line option is given) adds several constants to the built-in namespace. They are useful for the interactive interpreter shell and should not be used in programs.

`quit` (`code=None`)

`exit` (`code=None`)

Objects that when printed, print a message like “Use `quit()` or Ctrl-D (i.e. EOF) to exit”, and when called, raise `SystemExit` with the specified exit code.

`copyright`

`credits`

Objects that when printed or called, print the text of copyright or credits, respectively.

`license`

Object that when printed, prints the message “Type `license()` to see the full license text”, and when called, displays the full license text in a pager-like fashion (one screen at a time).

BUILT-IN TYPES

The following sections describe the standard types that are built into the interpreter.

The principal built-in types are numerics, sequences, mappings, classes, instances and exceptions.

Some collection classes are mutable. The methods that add, subtract, or rearrange their members in place, and don't return a specific item, never return the collection instance itself but `None`.

Some operations are supported by several object types; in particular, practically all objects can be compared for equality, tested for truth value, and converted to a string (with the `repr()` function or the slightly different `str()` function). The latter function is implicitly used when an object is written by the `print()` function.

4.1 Truth Value Testing

Any object can be tested for truth value, for use in an `if` or `while` condition or as operand of the Boolean operations below.

By default, an object is considered true unless its class defines either a `__bool__()` method that returns `False` or a `__len__()` method that returns zero, when called with the object.¹ Here are most of the built-in objects considered false:

- constants defined to be false: `None` and `False`.
- zero of any numeric type: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- empty sequences and collections: `' '`, `()`, `[]`, `{}`, `set()`, `range(0)`

Operations and built-in functions that have a Boolean result always return `0` or `False` for false and `1` or `True` for true, unless otherwise stated. (Important exception: the Boolean operations `or` and `and` always return one of their operands.)

4.2 Boolean Operations — `and`, `or`, `not`

These are the Boolean operations, ordered by ascending priority:

Operation	Result	Notes
<code>x or y</code>	if <code>x</code> is false, then <code>y</code> , else <code>x</code>	(1)
<code>x and y</code>	if <code>x</code> is false, then <code>x</code> , else <code>y</code>	(2)
<code>not x</code>	if <code>x</code> is false, then <code>True</code> , else <code>False</code>	(3)

¹ Additional information on these special methods may be found in the Python Reference Manual (customization).

Notes:

- (1) This is a short-circuit operator, so it only evaluates the second argument if the first one is false.
- (2) This is a short-circuit operator, so it only evaluates the second argument if the first one is true.
- (3) `not` has a lower priority than non-Boolean operators, so `not a == b` is interpreted as `not (a == b)`, and `a == not b` is a syntax error.

4.3 Comparisons

There are eight comparison operations in Python. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, `x < y <= z` is equivalent to `x < y` and `y <= z`, except that `y` is evaluated only once (but in both cases `z` is not evaluated at all when `x < y` is found to be false).

This table summarizes the comparison operations:

Operation	Meaning
<code><</code>	strictly less than
<code><=</code>	less than or equal
<code>></code>	strictly greater than
<code>>=</code>	greater than or equal
<code>==</code>	equal
<code>!=</code>	not equal
<code>is</code>	object identity
<code>is not</code>	negated object identity

Objects of different types, except different numeric types, never compare equal. The `==` operator is always defined but for some object types (for example, class objects) is equivalent to `is`. The `<`, `<=`, `>` and `>=` operators are only defined where they make sense; for example, they raise a `TypeError` exception when one of the arguments is a complex number.

Non-identical instances of a class normally compare as non-equal unless the class defines the `__eq__()` method.

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines enough of the methods `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` (in general, `__lt__()` and `__eq__()` are sufficient, if you want the conventional meanings of the comparison operators).

The behavior of the `is` and `is not` operators cannot be customized; also they can be applied to any two objects and never raise an exception.

Two more operations with the same syntactic priority, `in` and `not in`, are supported by types that are `iterable` or implement the `__contains__()` method.

4.4 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 the additional numeric types `fractions.Fraction`, for rationals, and `decimal.Decimal`, for 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 (for priorities of the operations, see operator-summary):

Operation	Result	Notes	Full documentation
<code>x + y</code>	sum of <code>x</code> and <code>y</code>		
<code>x - y</code>	difference of <code>x</code> and <code>y</code>		
<code>x * y</code>	product of <code>x</code> and <code>y</code>		
<code>x / y</code>	quotient of <code>x</code> and <code>y</code>		
<code>x // y</code>	floored quotient of <code>x</code> and <code>y</code>	(1)	
<code>x % y</code>	remainder of <code>x / y</code>	(2)	
<code>-x</code>	<code>x</code> negated		
<code>+x</code>	<code>x</code> unchanged		
<code>abs(x)</code>	absolute value or magnitude of <code>x</code>		<code>abs()</code>
<code>int(x)</code>	<code>x</code> converted to integer	(3)(6)	<code>int()</code>
<code>float(x)</code>	<code>x</code> converted to floating point	(4)(6)	<code>float()</code>
<code>complex(re, im)</code>	a complex number with real part <code>re</code> , imaginary part <code>im</code> . <code>im</code> defaults to zero.	(6)	<code>complex()</code>
<code>c.conjugate()</code>	conjugate of the complex number <code>c</code>		
<code>divmod(x, y)</code>	the pair <code>(x // y, x % y)</code>	(2)	<code>divmod()</code>
<code>pow(x, y)</code>	<code>x</code> to the power <code>y</code>	(5)	<code>pow()</code>
<code>x ** y</code>	<code>x</code> to the power <code>y</code>	(5)	

Notes:

- (1) Also referred to as integer division. The resultant value is a whole integer, though the result’s type is not necessarily `int`. The result is always rounded towards minus infinity: `1//2` is `0`, `(-1)//2` is `-1`, `1//(-2)` is `-1`, and `(-1)//(-2)` is `0`.
- (2) Not for complex numbers. Instead convert to floats using `abs()` if appropriate.
- (3) Conversion from floating point to integer may round or truncate as in C; see functions `math.floor()` and `math.ceil()` for well-defined conversions.
- (4) `float` also accepts the strings “`nan`” and “`inf`” with an optional prefix “`+`” or “`-`” for Not a Number (NaN) and positive or negative infinity.
- (5) Python defines `pow(0, 0)` and `0 ** 0` to be `1`, as is common for programming languages.
- (6) The numeric literals accepted include the digits 0 to 9 or any Unicode equivalent (code points with the `Nd` property). See <http://www.unicode.org/Public/12.1.0/ucd/extracted/DerivedNumericType.txt> for a complete list of code points with the `Nd` property.

All `numbers.Real` types (`int` and `float`) also include the following operations:

² As a consequence, the list `[1, 2]` is considered equal to `[1.0, 2.0]`, and similarly for tuples.

Operation	Result
<code>math.trunc(x)</code>	x truncated to <code>Integral</code>
<code>round(x[, n])</code>	x rounded to n digits, rounding half to even. If n is omitted, it defaults to 0.
<code>math.floor(x)</code>	the greatest <code>Integral</code> $\leq x$
<code>math.ceil(x)</code>	the least <code>Integral</code> $\geq x$

For additional numeric operations see the `math` and `cmath` modules.

4.4.1 Bitwise Operations on Integer Types

Bitwise operations only make sense for integers. The result of bitwise operations is calculated as though carried out in two's complement with an infinite number of sign bits.

The priorities of the binary bitwise operations are all lower than the numeric operations and higher than the comparisons; the unary operation `~` has the same priority as the other unary numeric operations (`+` and `-`).

This table lists the bitwise operations sorted in ascending priority:

Operation	Result	Notes
<code>x y</code>	bitwise <i>or</i> of x and y	(4)
<code>x ^ y</code>	bitwise <i>exclusive or</i> of x and y	(4)
<code>x & y</code>	bitwise <i>and</i> of x and y	(4)
<code>x << n</code>	x shifted left by n bits	(1)(2)
<code>x >> n</code>	x shifted right by n bits	(1)(3)
<code>~x</code>	the bits of x inverted	

Notes:

- (1) Negative shift counts are illegal and cause a `ValueError` to be raised.
- (2) A left shift by n bits is equivalent to multiplication by `pow(2, n)` without overflow check.
- (3) A right shift by n bits is equivalent to division by `pow(2, n)` without overflow check.
- (4) Performing these calculations with at least one extra sign extension bit in a finite two's complement representation (a working bit-width of $1 + \max(x.bit_length(), y.bit_length())$ or more) is sufficient to get the same result as if there were an infinite number of sign bits.

4.4.2 Additional Methods on Integer Types

The `int` type implements the `numbers.Integral` abstract base class. In addition, it provides a few more methods:

`int.bit_length()`

Return the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros:

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

More precisely, if x is nonzero, then `x.bit_length()` is the unique positive integer k such that $2^{k-1} \leq \text{abs}(x) < 2^k$. Equivalently, when `abs(x)` is small enough to have a correctly rounded logarithm, then $k = 1 + \text{int}(\log(\text{abs}(x), 2))$. If x is zero, then `x.bit_length()` returns 0.

Equivalent to:

```
def bit_length(self):
    s = bin(self)           # binary representation: bin(-37) --> '-0b100101'
    s = s.lstrip('-0b')     # remove leading zeros and minus sign
    return len(s)           # len('100101') --> 6
```

New in version 3.1.

`int.to_bytes(length, byteorder, *, signed=False)`

Return an array of bytes representing an integer.

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() + 7) // 8, byteorder='little')
b'\xe8\x03'
```

The integer is represented using *length* bytes. An *OverflowError* is raised if the integer is not representable with the given number of bytes.

The *byteorder* argument determines the byte order used to represent the integer. If *byteorder* is "big", the most significant byte is at the beginning of the byte array. If *byteorder* is "little", the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value.

The *signed* argument determines whether two's complement is used to represent the integer. If *signed* is False and a negative integer is given, an *OverflowError* is raised. The default value for *signed* is False.

New in version 3.2.

`classmethod int.from_bytes(bytes, byteorder, *, signed=False)`

Return the integer represented by the given array of bytes.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

The argument *bytes* must either be a *bytes-like object* or an iterable producing bytes.

The *byteorder* argument determines the byte order used to represent the integer. If *byteorder* is "big", the most significant byte is at the beginning of the byte array. If *byteorder* is "little", the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value.

The *signed* argument indicates whether two's complement is used to represent the integer.

New in version 3.2.

`int.as_integer_ratio()`

Return a pair of integers whose ratio is exactly equal to the original integer and with a positive denominator. The integer ratio of integers (whole numbers) is always the integer as the numerator and 1 as the denominator.

New in version 3.8.

4.4.3 Additional Methods on Float

The float type implements the `numbers.Real` abstract base class. float also has the following additional methods.

`float.as_integer_ratio()`

Return a pair of integers whose ratio is exactly equal to the original float and with a positive denominator. Raises `OverflowError` on infinities and a `ValueError` on NaNs.

`float.is_integer()`

Return True if the float instance is finite with integral value, and False otherwise:

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

Two methods support conversion to and from hexadecimal strings. Since Python’s floats are stored internally as binary numbers, converting a float to or from a *decimal* string usually involves a small rounding error. In contrast, hexadecimal strings allow exact representation and specification of floating-point numbers. This can be useful when debugging, and in numerical work.

`float.hex()`

Return a representation of a floating-point number as a hexadecimal string. For finite floating-point numbers, this representation will always include a leading 0x and a trailing p and exponent.

`classmethod float.fromhex(s)`

Class method to return the float represented by a hexadecimal string *s*. The string *s* may have leading and trailing whitespace.

Note that `float.hex()` is an instance method, while `float.fromhex()` is a class method.

A hexadecimal string takes the form:

```
[sign] ['0x'] integer ['. fraction] ['p' exponent]
```

where the optional sign may be either + or -, integer and fraction are strings of hexadecimal digits, and exponent is a decimal integer with an optional leading sign. Case is not significant, and there must be at least one hexadecimal digit in either the integer or the fraction. This syntax is similar to the syntax specified in section 6.4.4.2 of the C99 standard, and also to the syntax used in Java 1.5 onwards. In particular, the output of `float.hex()` is usable as a hexadecimal floating-point literal in C or Java code, and hexadecimal strings produced by C’s %a format character or Java’s Double.toHexString are accepted by `float.fromhex()`.

Note that the exponent is written in decimal rather than hexadecimal, and that it gives the power of 2 by which to multiply the coefficient. For example, the hexadecimal string 0x3.a7p10 represents the floating-point number (3 + 10./16 + 7./16**2) * 2.0**10, or 3740.0:

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

Applying the reverse conversion to 3740.0 gives a different hexadecimal string representing the same number:

```
>>> float.hex(3740.0)
'0x1.d3800000000000p+11'
```

4.4.4 Hashing of numeric types

For numbers x and y , possibly of different types, it's a requirement that $\text{hash}(x) == \text{hash}(y)$ whenever $x == y$ (see the `__hash__()` method documentation for more details). For ease of implementation and efficiency across a variety of numeric types (including `int`, `float`, `decimal.Decimal` and `fractions.Fraction`) Python's hash for numeric types is based on a single mathematical function that's defined for any rational number, and hence applies to all instances of `int` and `fractions.Fraction`, and all finite instances of `float` and `decimal.Decimal`. Essentially, this function is given by reduction modulo P for a fixed prime P . The value of P is made available to Python as the `modulus` attribute of `sys.hash_info`.

CPython implementation detail: Currently, the prime used is $P = 2^{31} - 1$ on machines with 32-bit C longs and $P = 2^{61} - 1$ on machines with 64-bit C longs.

Here are the rules in detail:

- If $x = m / n$ is a nonnegative rational number and n is not divisible by P , define $\text{hash}(x)$ as $m * \text{invmod}(n, P) \% P$, where `invmod(n, P)` gives the inverse of n modulo P .
- If $x = m / n$ is a nonnegative rational number and n is divisible by P (but m is not) then n has no inverse modulo P and the rule above doesn't apply; in this case define $\text{hash}(x)$ to be the constant value `sys.hash_info.inf`.
- If $x = m / n$ is a negative rational number define $\text{hash}(x)$ as $-\text{hash}(-x)$. If the resulting hash is -1 , replace it with -2 .
- The particular values `sys.hash_info.inf`, `-sys.hash_info.inf` and `sys.hash_info.nan` are used as hash values for positive infinity, negative infinity, or nans (respectively). (All hashable nans have the same hash value.)
- For a `complex` number z , the hash values of the real and imaginary parts are combined by computing $\text{hash}(z.real) + \text{sys.hash_info.imag} * \text{hash}(z.imag)$, reduced modulo $2^{sys.hash_info.width}$ so that it lies in range $(-2^{sys.hash_info.width - 1}, 2^{sys.hash_info.width - 1})$. Again, if the result is -1 , it's replaced with -2 .

To clarify the above rules, here's some example Python code, equivalent to the built-in hash, for computing the hash of a rational number, `float`, or `complex`:

```
import sys, math

def hash_fraction(m, n):
    """Compute the hash of a rational number m / n.

    Assumes m and n are integers, with n positive.
    Equivalent to hash(fractions.Fraction(m, n)).

    """
    P = sys.hash_info.modulus
    # Remove common factors of P. (Unnecessary if m and n already coprime.)
    while m % P == n % P == 0:
        m, n = m // P, n // P

    if n % P == 0:
        hash_value = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: pow(n, P-1, P) is 1, so
        hash_value = m * pow(n, P-2, P) % P
```

(continues on next page)

(continued from previous page)

```

# pow(n, P-2, P) gives the inverse of n modulo P.
hash_value = (abs(m) % P) * pow(n, P - 2, P) % P
if m < 0:
    hash_value = -hash_value
if hash_value == -1:
    hash_value = -2
return hash_value

def hash_float(x):
    """Compute the hash of a float x."""

    if math.isnan(x):
        return sys.hash_info.nan
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""

    hash_value = hash_float(z.real) + sys.hash_info.imag * hash_float(z.imag)
    # do a signed reduction modulo 2**sys.hash_info.width
    M = 2**sys.hash_info.width - 1
    hash_value = (hash_value & (M - 1)) - (hash_value & M)
    if hash_value == -1:
        hash_value = -2
    return hash_value

```

4.5 Iterator Types

Python supports a concept of iteration over containers. This is implemented using two distinct methods; these are used to allow user-defined classes to support iteration. Sequences, described below in more detail, always support the iteration methods.

One method needs to be defined for container objects to provide iteration support:

`container.__iter__()`

Return an iterator object. The object is required to support the iterator protocol described below. If a container supports different types of iteration, additional methods can be provided to specifically request iterators for those iteration types. (An example of an object supporting multiple forms of iteration would be a tree structure which supports both breadth-first and depth-first traversal.) This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

The iterator objects themselves are required to support the following two methods, which together form the *iterator protocol*:

`iterator.__iter__()`

Return the iterator object itself. This is required to allow both containers and iterators to be used with the `for` and `in` statements. This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

`iterator.__next__()`

Return the next item from the container. If there are no further items, raise the `StopIteration` exception. This method corresponds to the `tp_iternext` slot of the type structure for Python objects in the Python/C API.

Python defines several iterator objects to support iteration over general and specific sequence types, dictionaries, and other more specialized forms. The specific types are not important beyond their implementation of the iterator protocol.

Once an iterator's `__next__()` method raises `StopIteration`, it must continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken.

4.5.1 Generator Types

Python's *generators* provide a convenient way to implement the iterator protocol. If a container object's `__iter__()` method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the `__iter__()` and `__next__()` methods. More information about generators can be found in the documentation for the `yield` expression.

4.6 Sequence Types — `list`, `tuple`, `range`

There are three basic sequence types: lists, tuples, and range objects. Additional sequence types tailored for processing of *binary data* and *text strings* are described in dedicated sections.

4.6.1 Common Sequence Operations

The operations in the following table are supported by most sequence types, both mutable and immutable. The `collections.abc.Sequence` ABC is provided to make it easier to correctly implement these operations on custom sequence types.

This table lists the sequence operations sorted in ascending priority. In the table, *s* and *t* are sequences of the same type, *n*, *i*, *j* and *k* are integers and *x* is an arbitrary object that meets any type and value restrictions imposed by *s*.

The `in` and `not in` operations have the same priorities as the comparison operations. The `+` (concatenation) and `*` (repetition) operations have the same priority as the corresponding numeric operations.³

Operation	Result	Notes
<code>x in s</code>	True if an item of <i>s</i> is equal to <i>x</i> , else False	(1)
<code>x not in s</code>	False if an item of <i>s</i> is equal to <i>x</i> , else True	(1)
<code>s + t</code>	the concatenation of <i>s</i> and <i>t</i>	(6)(7)
<code>s * n</code> or <code>n * s</code>	equivalent to adding <i>s</i> to itself <i>n</i> times	(2)(7)
<code>s[i]</code>	<i>i</i> th item of <i>s</i> , origin 0	(3)
<code>s[i:j]</code>	slice of <i>s</i> from <i>i</i> to <i>j</i>	(3)(4)
<code>s[i:j:k]</code>	slice of <i>s</i> from <i>i</i> to <i>j</i> with step <i>k</i>	(3)(5)
<code>len(s)</code>	length of <i>s</i>	
<code>min(s)</code>	smallest item of <i>s</i>	
<code>max(s)</code>	largest item of <i>s</i>	
<code>s.index(x[, i[, j]])</code>	index of the first occurrence of <i>x</i> in <i>s</i> (at or after index <i>i</i> and before index <i>j</i>)	(8)
<code>s.count(x)</code>	total number of occurrences of <i>x</i> in <i>s</i>	

Sequences of the same type also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see comparisons in the language reference.)

Notes:

³ They must have since the parser can't tell the type of the operands.

- (1) While the `in` and `not in` operations are used only for simple containment testing in the general case, some specialised sequences (such as `str`, `bytes` and `bytearray`) also use them for subsequence testing:

```
>>> "gg" in "eggs"
True
```

- (2) Values of n less than 0 are treated as 0 (which yields an empty sequence of the same type as s). Note that items in the sequence s are not copied; they are referenced multiple times. This often haunts new Python programmers; consider:

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

What has happened is that `[]` is a one-element list containing an empty list, so all three elements of `[] * 3` are references to this single empty list. Modifying any of the elements of `lists` modifies this single list. You can create a list of different lists this way:

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

Further explanation is available in the FAQ entry [faq-multidimensional-list](#).

- (3) If i or j is negative, the index is relative to the end of sequence s : `len(s) + i` or `len(s) + j` is substituted. But note that -0 is still 0.
- (4) The slice of s from i to j is defined as the sequence of items with index k such that $i \leq k < j$. If i or j is greater than `len(s)`, use `len(s)`. If i is omitted or `None`, use 0. If j is omitted or `None`, use `len(s)`. If i is greater than or equal to j , the slice is empty.
- (5) The slice of s from i to j with step k is defined as the sequence of items with index $x = i + n*k$ such that $0 \leq n < (j-i)/k$. In other words, the indices are $i, i+k, i+2*k, i+3*k$ and so on, stopping when j is reached (but never including j). When k is positive, i and j are reduced to `len(s)` if they are greater. When k is negative, i and j are reduced to `len(s) - 1` if they are greater. If i or j are omitted or `None`, they become “end” values (which end depends on the sign of k). Note, k cannot be zero. If k is `None`, it is treated like 1.
- (6) Concatenating immutable sequences always results in a new object. This means that building up a sequence by repeated concatenation will have a quadratic runtime cost in the total sequence length. To get a linear runtime cost, you must switch to one of the alternatives below:
 - if concatenating `str` objects, you can build a list and use `str.join()` at the end or else write to an `io.StringIO` instance and retrieve its value when complete
 - if concatenating `bytes` objects, you can similarly use `bytes.join()` or `io.BytesIO`, or you can do in-place concatenation with a `bytearray` object. `bytearray` objects are mutable and have an efficient overallocation mechanism
 - if concatenating `tuple` objects, extend a `list` instead
 - for other types, investigate the relevant class documentation
- (7) Some sequence types (such as `range`) only support item sequences that follow specific patterns, and hence don’t support sequence concatenation or repetition.

- (8) `index` raises `ValueError` when `x` is not found in `s`. Not all implementations support passing the additional arguments `i` and `j`. These arguments allow efficient searching of subsections of the sequence. Passing the extra arguments is roughly equivalent to using `s[i:j].index(x)`, only without copying any data and with the returned index being relative to the start of the sequence rather than the start of the slice.

4.6.2 Immutable Sequence Types

The only operation that immutable sequence types generally implement that is not also implemented by mutable sequence types is support for the `hash()` built-in.

This support allows immutable sequences, such as `tuple` instances, to be used as `dict` keys and stored in `set` and `frozenset` instances.

Attempting to hash an immutable sequence that contains unhashable values will result in `TypeError`.

4.6.3 Mutable Sequence Types

The operations in the following table are defined on mutable sequence types. The `collections.abc.MutableSequence` ABC is provided to make it easier to correctly implement these operations on custom sequence types.

In the table `s` is an instance of a mutable sequence type, `t` is any iterable object and `x` is an arbitrary object that meets any type and value restrictions imposed by `s` (for example, `bytearray` only accepts integers that meet the value restriction `0 <= x <= 255`).

Operation	Result	Notes
<code>s[i] = x</code>	item <code>i</code> of <code>s</code> is replaced by <code>x</code>	
<code>s[i:j] = t</code>	slice of <code>s</code> from <code>i</code> to <code>j</code> is replaced by the contents of the iterable <code>t</code>	
<code>del s[i:j]</code>	same as <code>s[i:j] = []</code>	
<code>s[i:j:k] = t</code>	the elements of <code>s[i:j:k]</code> are replaced by those of <code>t</code>	(1)
<code>del s[i:j:k]</code>	removes the elements of <code>s[i:j:k]</code> from the list	
<code>s.append(x)</code>	appends <code>x</code> to the end of the sequence (same as <code>s[len(s):len(s)] = [x]</code>)	
<code>s.clear()</code>	removes all items from <code>s</code> (same as <code>del s[:]</code>)	(5)
<code>s.copy()</code>	creates a shallow copy of <code>s</code> (same as <code>s[:]</code>)	(5)
<code>s.extend(t)</code> or <code>s += t</code>	extends <code>s</code> with the contents of <code>t</code> (for the most part the same as <code>s[len(s):len(s)] = t</code>)	
<code>s *= n</code>	updates <code>s</code> with its contents repeated <code>n</code> times	(6)
<code>s.insert(i, x)</code>	inserts <code>x</code> into <code>s</code> at the index given by <code>i</code> (same as <code>s[i:i] = [x]</code>)	
<code>s.pop([i])</code>	retrieves the item at <code>i</code> and also removes it from <code>s</code>	(2)
<code>s.remove(x)</code>	remove the first item from <code>s</code> where <code>s[i]</code> is equal to <code>x</code>	(3)
<code>s.reverse()</code>	reverses the items of <code>s</code> in place	(4)

Notes:

- (1) `t` must have the same length as the slice it is replacing.
- (2) The optional argument `i` defaults to `-1`, so that by default the last item is removed and returned.
- (3) `remove()` raises `ValueError` when `x` is not found in `s`.
- (4) The `reverse()` method modifies the sequence in place for economy of space when reversing a large sequence. To remind users that it operates by side effect, it does not return the reversed sequence.

(5) `clear()` and `copy()` are included for consistency with the interfaces of mutable containers that don't support slicing operations (such as `dict` and `set`). `copy()` is not part of the `collections.abc.MutableSequence` ABC, but most concrete mutable sequence classes provide it.

New in version 3.3: `clear()` and `copy()` methods.

(6) The value `n` is an integer, or an object implementing `__index__()`. Zero and negative values of `n` clear the sequence. Items in the sequence are not copied; they are referenced multiple times, as explained for `s * n` under *Common Sequence Operations*.

4.6.4 Lists

Lists are mutable sequences, typically used to store collections of homogeneous items (where the precise degree of similarity will vary by application).

`class list([iterable])`

Lists may be constructed in several ways:

- Using a pair of square brackets to denote the empty list: `[]`
- Using square brackets, separating items with commas: `[a], [a, b, c]`
- Using a list comprehension: `[x for x in iterable]`
- Using the type constructor: `list()` or `list(iterable)`

The constructor builds a list whose items are the same and in the same order as `iterable`'s items. `iterable` may be either a sequence, a container that supports iteration, or an iterator object. If `iterable` is already a list, a copy is made and returned, similar to `iterable[:]`. For example, `list('abc')` returns `['a', 'b', 'c']` and `list((1, 2, 3))` returns `[1, 2, 3]`. If no argument is given, the constructor creates a new empty list, `[]`.

Many other operations also produce lists, including the `sorted()` built-in.

Lists implement all of the `common` and `mutable` sequence operations. Lists also provide the following additional method:

`sort(*, key=None, reverse=False)`

This method sorts the list in place, using only `<` comparisons between items. Exceptions are not suppressed – if any comparison operations fail, the entire sort operation will fail (and the list will likely be left in a partially modified state).

`sort()` accepts two arguments that can only be passed by keyword (*keyword-only arguments*):

`key` specifies a function of one argument that is used to extract a comparison key from each list element (for example, `key=str.lower`). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of `None` means that list items are sorted directly without calculating a separate key value.

The `functools.cmp_to_key()` utility is available to convert a 2.x style `cmp` function to a `key` function.

`reverse` is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

This method modifies the sequence in place for economy of space when sorting a large sequence. To remind users that it operates by side effect, it does not return the sorted sequence (use `sorted()` to explicitly request a new sorted list instance).

The `sort()` method is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

For sorting examples and a brief sorting tutorial, see [sortinghowto](#).

CPython implementation detail: While a list is being sorted, the effect of attempting to mutate, or even inspect, the list is undefined. The C implementation of Python makes the list appear empty for the duration, and raises [ValueError](#) if it can detect that the list has been mutated during a sort.

4.6.5 Tuples

Tuples are immutable sequences, typically used to store collections of heterogeneous data (such as the 2-tuples produced by the [enumerate\(\)](#) built-in). Tuples are also used for cases where an immutable sequence of homogeneous data is needed (such as allowing storage in a [set](#) or [dict](#) instance).

```
class tuple([iterable])
```

Tuples may be constructed in a number of ways:

- Using a pair of parentheses to denote the empty tuple: ()
- Using a trailing comma for a singleton tuple: a, or (a,)
- Separating items with commas: a, b, c or (a, b, c)
- Using the [tuple\(\)](#) built-in: tuple() or tuple(iterable)

The constructor builds a tuple whose items are the same and in the same order as *iterable*'s items. *iterable* may be either a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a tuple, it is returned unchanged. For example, tuple('abc') returns ('a', 'b', 'c') and tuple([1, 2, 3]) returns (1, 2, 3). If no argument is given, the constructor creates a new empty tuple, ().

Note that it is actually the comma which makes a tuple, not the parentheses. The parentheses are optional, except in the empty tuple case, or when they are needed to avoid syntactic ambiguity. For example, f(a, b, c) is a function call with three arguments, while f((a, b, c)) is a function call with a 3-tuple as the sole argument.

Tuples implement all of the [common](#) sequence operations.

For heterogeneous collections of data where access by name is clearer than access by index, [collections.namedtuple\(\)](#) may be a more appropriate choice than a simple tuple object.

4.6.6 Ranges

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))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

Ranges implement all of the [common](#) sequence operations except concatenation and repetition (due to the fact that range objects can only represent sequences that follow a strict pattern and repetition and concatenation will usually violate that pattern).

start

The value of the *start* parameter (or 0 if the parameter was not supplied)

stop

The value of the *stop* parameter

step

The value of the *step* parameter (or 1 if the parameter was not supplied)

The advantage of the `range` type over a regular `list` or `tuple` is that a `range` object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the `start`, `stop` and `step` values, calculating individual items and subranges as needed).

Range objects implement the `collections.abc.Sequence` ABC, and provide features such as containment tests, element index lookup, slicing and support for negative indices (see [Sequence Types — `list`, `tuple`, `range`](#)):

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

Testing range objects for equality with `==` and `!=` compares them as sequences. That is, two range objects are considered equal if they represent the same sequence of values. (Note that two range objects that compare equal might have different `start`, `stop` and `step` attributes, for example `range(0) == range(2, 1, 3)` or `range(0, 3, 2) == range(0, 4, 2)`.)

Changed in version 3.2: Implement the Sequence ABC. Support slicing and negative indices. Test `int` objects for membership in constant time instead of iterating through all items.

Changed in version 3.3: Define ‘==’ and ‘!=’ to compare range objects based on the sequence of values they define (instead of comparing based on object identity).

New in version 3.3: The `start`, `stop` and `step` attributes.

See also:

- The [linspace recipe](#) shows how to implement a lazy version of range suitable for floating point applications.

4.7 Text Sequence Type — `str`

Textual data in Python is handled with `str` objects, or *strings*. Strings are immutable *sequences* of Unicode code points. String literals are written in a variety of ways:

- Single quotes: ' allows embedded "double" quotes'
- Double quotes: " allows embedded 'single' quotes".
- Triple quoted: '''Three single quotes''', """Three double quotes"""

Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.

String literals that are part of a single expression and have only whitespace between them will be implicitly converted to a single string literal. That is, ("spam " "eggs") == "spam eggs".

See [strings](#) for more about the various forms of string literal, including supported escape sequences, and the `r` (“raw”) prefix that disables most escape sequence processing.

Strings may also be created from other objects using the `str` constructor.

Since there is no separate “character” type, indexing a string produces strings of length 1. That is, for a non-empty string `s`, `s[0] == s[0:1]`.

There is also no mutable string type, but `str.join()` or `io.StringIO` can be used to efficiently construct strings from multiple fragments.

Changed in version 3.3: For backwards compatibility with the Python 2 series, the `u` prefix is once again permitted on string literals. It has no effect on the meaning of string literals and cannot be combined with the `r` prefix.

```
class str(object=’’)
class str(object=b”, encoding=’utf-8’, errors=’strict’)
```

Return a `string` version of `object`. If `object` is not provided, returns the empty string. Otherwise, the behavior of `str()` depends on whether `encoding` or `errors` is given, as follows.

If neither `encoding` nor `errors` is given, `str(object)` returns `object.__str__()`, which is the “informal” or nicely printable string representation of `object`. For string objects, this is the string itself. If `object` does not have a `__str__()` method, then `str()` falls back to returning `repr(object)`.

If at least one of `encoding` or `errors` is given, `object` should be a *bytes-like object* (e.g. `bytes` or `bytearray`). In this case, if `object` is a `bytes` (or `bytearray`) object, then `str(bytes, encoding, errors)` is equivalent to `bytes.decode(encoding, errors)`. Otherwise, the bytes object underlying the buffer object is obtained before calling `bytes.decode()`. See [Binary Sequence Types — bytes, bytearray, memoryview](#) and [bufferobjects](#) for information on buffer objects.

Passing a `bytes` object to `str()` without the `encoding` or `errors` arguments falls under the first case of returning the informal string representation (see also the `-b` command-line option to Python). For example:

```
>>> str(b’Zoot!’)
"b'Zoot!'"
```

For more information on the `str` class and its methods, see [Text Sequence Type — `str`](#) and the [String Methods](#) section below. To output formatted strings, see the f-strings and [Format String Syntax](#) sections. In addition, see the [Text Processing Services](#) section.

4.7.1 String Methods

Strings implement all of the [common](#) sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see [`str.format\(\)`](#), [Format String Syntax](#) and [Custom String Formatting](#)) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle ([printf-style String Formatting](#)).

The [Text Processing Services](#) section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the `re` module).

`str.capitalize()`

Return a copy of the string with its first character capitalized and the rest lowercased.

Changed in version 3.8: The first character is now put into titlecase rather than uppercase. This means that characters like digraphs will only have their first letter capitalized, instead of the full character.

`str.casefold()`

Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

CASEFOLDING is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter 'ß' is equivalent to "ss". Since it is already lowercase, `lower()` would do nothing to 'ß'; `casefold()` converts it to "ss".

The casefolding algorithm is described in section 3.13 of the Unicode Standard.

New in version 3.3.

`str.center(width[, fillchar])`

Return centered in a string of length `width`. Padding is done using the specified `fillchar` (default is an ASCII space). The original string is returned if `width` is less than or equal to `len(s)`.

`str.count(sub[, start[, end]])`

Return the number of non-overlapping occurrences of substring `sub` in the range `[start, end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

`str.encode(encoding="utf-8", errors="strict")`

Return an encoded version of the string as a bytes object. Default encoding is 'utf-8'. `errors` may be given to set a different error handling scheme. The default for `errors` is 'strict', meaning that encoding errors raise a [UnicodeError](#). Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other name registered via `codecs.register_error()`, see section [Error Handlers](#). For a list of possible encodings, see section [Standard Encodings](#).

Changed in version 3.1: Support for keyword arguments added.

`str.endswith(suffix[, start[, end]])`

Return True if the string ends with the specified `suffix`, otherwise return False. `suffix` can also be a tuple of suffixes to look for. With optional `start`, test beginning at that position. With optional `end`, stop comparing at that position.

`str.expandtabs(tabsize=8)`

Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every `tabsize` characters (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the string, the current column is set to zero and the string is examined character by character. If the character is a tab (\t), one or more space characters are inserted in the result until the

current column is equal to the next tab position. (The tab character itself is not copied.) If the character is a newline (\n) or return (\r), it is copied and the current column is reset to zero. Any other character is copied unchanged and the current column is incremented by one regardless of how the character is represented when printed.

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012      0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012  0123   01234'
```

`str.find(sub[, start[, end]])`

Return the lowest index in the string where substring *sub* is found within the slice *s[start:end]*. Optional arguments *start* and *end* are interpreted as in slice notation. Return -1 if *sub* is not found.

Note: The `find()` method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the `in` operator:

```
>>> 'Py' in 'Python'
True
```

`str.format(*args, **kwargs)`

Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces {} . Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

See [Format String Syntax](#) for a description of the various formatting options that can be specified in format strings.

Note: When formatting a number (`int`, `float`, `complex`, `decimal.Decimal` and subclasses) with the `n` type (ex: '`{:n}`'.format(1234)), the function temporarily sets the `LC_CTYPE` locale to the `LC_NUMERIC` locale to decode `decimal_point` and `thousands_sep` fields of `localeconv()` if they are non-ASCII or longer than 1 byte, and the `LC_NUMERIC` locale is different than the `LC_CTYPE` locale. This temporary change affects other threads.

Changed in version 3.7: When formatting a number with the `n` type, the function sets temporarily the `LC_CTYPE` locale to the `LC_NUMERIC` locale in some cases.

`str.format_map(mapping)`

Similar to `str.format(**mapping)`, except that *mapping* is used directly and not copied to a `dict`. This is useful if for example *mapping* is a `dict` subclass:

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

New in version 3.2.

`str.index(sub[, start[, end]])`

Like `find()`, but raise `ValueError` when the substring is not found.

`str.isalnum()`

Return True if all characters in the string are alphanumeric and there is at least one character, False otherwise. A character `c` is alphanumeric if one of the following returns True: `c.isalpha()`, `c.isdecimal()`, `c.isdigit()`, or `c.isnumeric()`.

`str.isalpha()`

Return True if all characters in the string are alphabetic and there is at least one character, False otherwise. Alphabetic characters are those characters defined in the Unicode character database as “Letter”, i.e., those with general category property being one of “Lm”, “Lt”, “Lu”, “Ll”, or “Lo”. Note that this is different from the “Alphabetic” property defined in the Unicode Standard.

`str.isascii()`

Return True if the string is empty or all characters in the string are ASCII, False otherwise. ASCII characters have code points in the range U+0000-U+007F.

New in version 3.7.

`str.isdecimal()`

Return True if all characters in the string are decimal characters and there is at least one character, False otherwise. Decimal characters are those that can be used to form numbers in base 10, e.g. U+0660, ARABIC-INDIC DIGIT ZERO. Formally a decimal character is a character in the Unicode General Category “Nd”.

`str.isdigit()`

Return True if all characters in the string are digits and there is at least one character, False otherwise. Digits include decimal characters and digits that need special handling, such as the compatibility superscript digits. This covers digits which cannot be used to form numbers in base 10, like the Kharosthi numbers. Formally, a digit is a character that has the property value Numeric_Type=Digit or Numeric_Type=Decimal.

`str.isidentifier()`

Return True if the string is a valid identifier according to the language definition, section identifiers.

Call `keyword.iskeyword()` to test whether string `s` is a reserved identifier, such as `def` and `class`.

Example:

```
>>> from keyword import iskeyword
>>> 'hello'.isidentifier(), iskeyword('hello')
True, False
>>> 'def'.isidentifier(), iskeyword('def')
True, True
```

`str.islower()`

Return True if all cased characters⁴ in the string are lowercase and there is at least one cased character, False otherwise.

`str.isnumeric()`

Return True if all characters in the string are numeric characters, and there is at least one character, False otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH. Formally, numeric characters are those with the property value Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric.

`str.isprintable()`

Return True if all characters in the string are printable or the string is empty, False otherwise. Nonprintable characters are those characters defined in the Unicode character database as “Other” or “Separator”, excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when `repr()` is invoked on a string. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.)

⁴ Cased characters are those with general category property being one of “Lu” (Letter, uppercase), “Ll” (Letter, lowercase), or “Lt” (Letter, titlecase).

str.isspace()

Return True if there are only whitespace characters in the string and there is at least one character, False otherwise.

A character is *whitespace* if in the Unicode character database (see [unicodedata](#)), either its general category is Zs (“Separator, space”), or its bidirectional class is one of WS, B, or S.

str.istitle()

Return True if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

str.isupper()

Return True if all cased characters⁴ in the string are uppercase and there is at least one cased character, False otherwise.

str.join(iterable)

Return a string which is the concatenation of the strings in *iterable*. A [TypeError](#) will be raised if there are any non-string values in *iterable*, including [bytes](#) objects. The separator between elements is the string providing this method.

str.ljust(width[, fillchar])

Return the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

str.lower()

Return a copy of the string with all the cased characters⁴ converted to lowercase.

The lowercasing algorithm used is described in section 3.13 of the Unicode Standard.

str.lstrip([chars])

Return a copy of the string with leading characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or None, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped:

```
>>> '    spacious    '.lstrip()
'spacious'
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

static str.maketrans(x[, y[, z]])

This static method returns a translation table usable for `str.translate()`.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters (strings of length 1) to Unicode ordinals, strings (of arbitrary lengths) or None. Character keys will then be converted to ordinals.

If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in *x* will be mapped to the character at the same position in *y*. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

str.partition(sep)

Split the string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

str.replace(old, new[, count])

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

`str.rfind(sub[, start[, end]])`

Return the highest index in the string where substring *sub* is found, such that *sub* is contained within *s[start:end]*. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

`str.rindex(sub[, start[, end]])`

Like `rfind()` but raises `ValueError` when the substring *sub* is not found.

`str.rjust(width[, fillchar])`

Return the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.rpartition(sep)`

Split the string at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

`str.rsplit(sep=None, maxsplit=-1)`

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any whitespace string is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below.

`str.rstrip([chars])`

Return a copy of the string with trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> '    spacious    '.rstrip()
'    spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

`str.split(sep=None, maxsplit=-1)`

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most *maxsplit*+1 elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `[]`.

For example:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
```

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

For example:

```
>>> '1 2 3'.split()
['1', '2', '3']
```

(continues on next page)

(continued from previous page)

```
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> ' 1 2 3 '.split()
['1', '2', '3']
```

str.splitlines([keepends])

Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless *keepends* is given and true.

This method splits on the following line boundaries. In particular, the boundaries are a superset of *universal new-lines*.

Representation	Description
\n	Line Feed
\r	Carriage Return
\r\n	Carriage Return + Line Feed
\v or \x0b	Line Tabulation
\f or \x0c	Form Feed
\x1c	File Separator
\x1d	Group Separator
\x1e	Record Separator
\x85	Next Line (C1 Control Code)
\u2028	Line Separator
\u2029	Paragraph Separator

Changed in version 3.2: \v and \f added to list of line boundaries.

For example:

```
>>> 'ab c\nnde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\nnde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

Unlike *split()* when a delimiter string *sep* is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

For comparison, *split('\n')* gives:

```
>>> ''.split('\n')
[']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

str.startswith(prefix[, start[, end]])

Return True if string starts with the *prefix*, otherwise return False. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test string beginning at that position. With optional *end*, stop comparing string at that position.

`str.strip([chars])`

Return a copy of the string with the leading and trailing characters removed. The `chars` argument is a string specifying the set of characters to be removed. If omitted or `None`, the `chars` argument defaults to removing whitespace. The `chars` argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```
>>> '    spacious    '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

The outermost leading and trailing `chars` argument values are stripped from the string. Characters are removed from the leading end until reaching a string character that is not contained in the set of characters in `chars`. A similar action takes place on the trailing end. For example:

```
>>> comment_string = '#..... Section 3.2.1 Issue #32 .....'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

`str.swapcase()`

Return a copy of the string with uppercase characters converted to lowercase and vice versa. Note that it is not necessarily true that `s.swapcase().swapcase() == s`.

`str.title()`

Return a titlecased version of the string where words start with an uppercase character and the remaining characters are lowercase.

For example:

```
>>> 'Hello world'.title()
'Hello World'
```

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

```
>>> "they're bill's friends from the UK".title()
"They'Re Bill'S Friends From The Uk"
```

A workaround for apostrophes can be constructed using regular expressions:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                  lambda mo: mo.group(0).capitalize(),
...                  s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

`str.translate(table)`

Return a copy of the string in which each character has been mapped through the given translation table. The table must be an object that implements indexing via `__getitem__()`, typically a `mapping` or `sequence`. When indexed by a Unicode ordinal (an integer), the table object can do any of the following: return a Unicode ordinal or a string, to map the character to one or more other characters; return `None`, to delete the character from the return string; or raise a `LookupError` exception, to map the character to itself.

You can use `str.maketrans()` to create a translation map from character-to-character mappings in different formats.

See also the [codecs](#) module for a more flexible approach to custom character mappings.

`str.upper()`

Return a copy of the string with all the cased characters⁴ converted to uppercase. Note that `s.upper().isupper()` might be `False` if `s` contains uncased characters or if the Unicode category of the resulting character(s) is not “Lu” (Letter, uppercase), but e.g. “Lt” (Letter, titlecase).

The uppercasing algorithm used is described in section 3.13 of the Unicode Standard.

`str.zfill(width)`

Return a copy of the string left filled with ASCII '0' digits to make a string of length `width`. A leading sign prefix ('+'/'-') is handled by inserting the padding *after* the sign character rather than before. The original string is returned if `width` is less than or equal to `len(s)`.

For example:

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

4.7.2 printf-style String Formatting

Note: The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer formatted string literals, the `str.format()` interface, or [template strings](#) may help avoid these errors. Each of these alternatives provides their own trade-offs and benefits of simplicity, flexibility, and/or extensibility.

String objects have one unique built-in operation: the `%` operator (modulo). This is also known as the string *formatting* or *interpolation* operator. Given `format % values` (where `format` is a string), `%` conversion specifications in `format` are replaced with zero or more elements of `values`. The effect is similar to using the `sprintf()` in the C language.

If `format` requires a single argument, `values` may be a single non-tuple object.⁵ Otherwise, `values` must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The '%' character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, `(somename)`).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an '*' (asterisk), the actual width is read from the next element of the tuple in `values`, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a '.' (dot) followed by the precision. If specified as '*' (an asterisk), the actual precision is read from the next element of the tuple in `values`, and the value to convert comes after the precision.
6. Length modifier (optional).
7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the string *must* include a parenthesised mapping key into that dictionary inserted immediately after the '%' character. The mapping key selects the value to be formatted from the mapping. For example:

⁵ To format only a tuple you should therefore provide a singleton tuple whose only element is the tuple to be formatted.

```
>>> print('%(language)s has %(number)03d quote types.' %
...      {'language': "Python", "number": 2})
Python has 002 quote types.
```

In this case no * specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

Flag	Meaning
'#'	The value conversion will use the “alternate form” (where defined below).
'0'	The conversion will be zero padded for numeric values.
'-'	The converted value is left adjusted (overrides the '0' conversion if both are given).
' '	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
'+'	A sign character ('+' or '-') will precede the conversion (overrides a “space” flag).

A length modifier (h, l, or L) may be present, but is ignored as it is not necessary for Python – so e.g. %ld is identical to %d.

The conversion types are:

Conversion	Meaning	Notes
'd'	Signed integer decimal.	
'i'	Signed integer decimal.	
'o'	Signed octal value.	(1)
'u'	Obsolete type – it is identical to 'd'.	(6)
'x'	Signed hexadecimal (lowercase).	(2)
'X'	Signed hexadecimal (uppercase).	(2)
'e'	Floating point exponential format (lowercase).	(3)
'E'	Floating point exponential format (uppercase).	(3)
'f'	Floating point decimal format.	(3)
'F'	Floating point decimal format.	(3)
'g'	Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'G'	Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'c'	Single character (accepts integer or single character string).	
'r'	String (converts any Python object using <code>repr()</code>).	(5)
's'	String (converts any Python object using <code>str()</code>).	(5)
'a'	String (converts any Python object using <code>ascii()</code>).	(5)
'%'	No argument is converted, results in a '%' character in the result.	

Notes:

- (1) The alternate form causes a leading octal specifier ('0o') to be inserted before the first digit.
- (2) The alternate form causes a leading '0x' or '0X' (depending on whether the 'x' or 'X' format was used) to be inserted before the first digit.
- (3) The alternate form causes the result to always contain a decimal point, even if no digits follow it.
The precision determines the number of digits after the decimal point and defaults to 6.
- (4) The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

The precision determines the number of significant digits before and after the decimal point and defaults to 6.

- (5) If precision is N, the output is truncated to N characters.
- (6) See [PEP 237](#).

Since Python strings have an explicit length, %s conversions do not assume that '\0' is the end of the string.

Changed in version 3.1: %f conversions for numbers whose absolute value is over 1e50 are no longer replaced by %g conversions.

4.8 Binary Sequence Types — bytes, bytearray, memoryview

The core built-in types for manipulating binary data are [bytes](#) and [bytearray](#). They are supported by [memoryview](#) which uses the buffer protocol to access the memory of other binary objects without needing to make a copy.

The [array](#) module supports efficient storage of basic data types like 32-bit integers and IEEE754 double-precision floating values.

4.8.1 Bytes Objects

Bytes objects are immutable sequences of single bytes. Since many major binary protocols are based on the ASCII text encoding, bytes objects offer several methods that are only valid when working with ASCII compatible data and are closely related to string objects in a variety of other ways.

`class bytes([source[, encoding[, errors]]])`

Firstly, the syntax for bytes literals is largely the same as that for string literals, except that a `b` prefix is added:

- Single quotes: `b'still allows embedded "double" quotes'`
- Double quotes: `b"still allows embedded 'single' quotes".`
- Triple quoted: `b'''3 single quotes'''`, `b"""\3 double quotes"""`

Only ASCII characters are permitted in bytes literals (regardless of the declared source code encoding). Any binary values over 127 must be entered into bytes literals using the appropriate escape sequence.

As with string literals, bytes literals may also use a `r` prefix to disable processing of escape sequences. See [strings](#) for more about the various forms of bytes literal, including supported escape sequences.

While bytes literals and representations are based on ASCII text, bytes objects actually behave like immutable sequences of integers, with each value in the sequence restricted such that $0 \leq x < 256$ (attempts to violate this restriction will trigger [ValueError](#)). This is done deliberately to emphasise that while many binary formats include ASCII based elements and can be usefully manipulated with some text-oriented algorithms, this is not generally the case for arbitrary binary data (blindly applying text processing algorithms to binary data formats that are not ASCII compatible will usually lead to data corruption).

In addition to the literal forms, bytes objects can be created in a number of other ways:

- A zero-filled bytes object of a specified length: `bytes(10)`
- From an iterable of integers: `bytes(range(20))`
- Copying existing binary data via the buffer protocol: `bytes(obj)`

Also see the [bytes](#) built-in.

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing binary data. Accordingly, the bytes type has an additional class method to read data in that format:

classmethod fromhex(string)

This `bytes` class method returns a bytes object, decoding the given string object. The string must contain two hexadecimal digits per byte, with ASCII whitespace being ignored.

```
>>> bytes.fromhex('2Ef0 F1f2 ')
b'\xf0\xf1\xf2'
```

Changed in version 3.7: `bytes.fromhex()` now skips all ASCII whitespace in the string, not just spaces.

A reverse conversion function exists to transform a bytes object into its hexadecimal representation.

hex([sep[, bytes_per_sep]])

Return a string object containing two hexadecimal digits for each byte in the instance.

```
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

If you want to make the hex string easier to read, you can specify a single character separator `sep` parameter to include in the output. By default between each byte. A second optional `bytes_per_sep` parameter controls the spacing. Positive values calculate the separator position from the right, negative values from the left.

```
>>> value = b'\xf0\xf1\xf2'
>>> value.hex('-')
'f0-f1-f2'
>>> value.hex('_', 2)
'f0_f1f2'
>>> b'UUDDLRLRAB'.hex(' ', -4)
'55554444 4c524c52 4142'
```

New in version 3.5.

Changed in version 3.8: `bytes.hex()` now supports optional `sep` and `bytes_per_sep` parameters to insert separators between bytes in the hex output.

Since bytes objects are sequences of integers (akin to a tuple), for a bytes object `b`, `b[0]` will be an integer, while `b[0:1]` will be a bytes object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytes objects uses the literal format (`b'...'`) since it is often more useful than e.g. `bytes([46, 46, 46])`. You can always convert a bytes object into a list of integers using `list(b)`.

Note: For Python 2.x users: In the Python 2.x series, a variety of implicit conversions between 8-bit strings (the closest thing 2.x offers to a built-in binary data type) and Unicode strings were permitted. This was a backwards compatibility workaround to account for the fact that Python originally only supported 8-bit text, and Unicode text was a later addition. In Python 3.x, those implicit conversions are gone - conversions between 8-bit binary data and Unicode text must be explicit, and bytes and string objects will always compare unequal.

4.8.2 Bytearray Objects

`bytearray` objects are a mutable counterpart to `bytes` objects.

class `bytearray`(`[source[, encoding[, errors]]]`)

There is no dedicated literal syntax for bytearray objects, instead they are always created by calling the constructor:

- Creating an empty instance: `bytearray()`
- Creating a zero-filled instance with a given length: `bytearray(10)`
- From an iterable of integers: `bytearray(range(20))`
- Copying existing binary data via the buffer protocol: `bytearray(b'Hi !')`

As bytearray objects are mutable, they support the *mutable* sequence operations in addition to the common bytes and bytearray operations described in *Bytes and Bytearray Operations*.

Also see the `bytearray` built-in.

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing binary data. Accordingly, the bytearray type has an additional class method to read data in that format:

classmethod `fromhex`(`string`)

This `bytearray` class method returns bytearray object, decoding the given string object. The string must contain two hexadecimal digits per byte, with ASCII whitespace being ignored.

```
>>> bytearray.fromhex('2Ef0 F1f2 ')
bytearray(b'.\xf0\xf1\xf2')
```

Changed in version 3.7: `bytearray.fromhex()` now skips all ASCII whitespace in the string, not just spaces.

A reverse conversion function exists to transform a bytearray object into its hexadecimal representation.

`hex([sep[, bytes_per_sep]])`

Return a string object containing two hexadecimal digits for each byte in the instance.

```
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

New in version 3.5.

Changed in version 3.8: Similar to `bytes.hex()`, `bytearray.hex()` now supports optional `sep` and `bytes_per_sep` parameters to insert separators between bytes in the hex output.

Since bytearray objects are sequences of integers (akin to a list), for a bytearray object `b`, `b[0]` will be an integer, while `b[0:1]` will be a bytearray object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytearray objects uses the bytes literal format (`bytearray(b'...')`) since it is often more useful than e.g. `bytearray([46, 46, 46])`. You can always convert a bytearray object into a list of integers using `list(b)`.

4.8.3 Bytes and Bytearray Operations

Both bytes and bytearray objects support the [common](#) sequence operations. They interoperate not just with operands of the same type, but with any [bytes-like object](#). Due to this flexibility, they can be freely mixed in operations without causing errors. However, the return type of the result may depend on the order of operands.

Note: The methods on bytes and bytearray objects don't accept strings as their arguments, just as the methods on strings don't accept bytes as their arguments. For example, you have to write:

```
a = "abc"  
b = a.replace("a", "f")
```

and:

```
a = b"abc"  
b = a.replace(b"a", b"f")
```

Some bytes and bytearray operations assume the use of ASCII compatible binary formats, and hence should be avoided when working with arbitrary binary data. These restrictions are covered below.

Note: Using these ASCII based operations to manipulate binary data that is not stored in an ASCII based format may lead to data corruption.

The following methods on bytes and bytearray objects can be used with arbitrary binary data.

`bytes.count(sub[, start[, end]])`
`bytearray.count(sub[, start[, end]])`

Return the number of non-overlapping occurrences of subsequence *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

The subsequence to search for may be any [bytes-like object](#) or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

`bytes.decode(encoding="utf-8", errors="strict")`
`bytearray.decode(encoding="utf-8", errors="strict")`

Return a string decoded from the given bytes. Default encoding is 'utf-8'. *errors* may be given to set a different error handling scheme. The default for *errors* is 'strict', meaning that encoding errors raise a [UnicodeError](#). Other possible values are 'ignore', 'replace' and any other name registered via `codecs.register_error()`, see section [Error Handlers](#). For a list of possible encodings, see section [Standard Encodings](#).

Note: Passing the *encoding* argument to `str` allows decoding any [bytes-like object](#) directly, without needing to make a temporary bytes or bytearray object.

Changed in version 3.1: Added support for keyword arguments.

`bytes.endswith(suffix[, start[, end]])`
`bytearray.endswith(suffix[, start[, end]])`

Return True if the binary data ends with the specified *suffix*, otherwise return False. *suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

The suffix(es) to search for may be any [bytes-like object](#).

```
bytes.find(sub[, start[, end]])  
bytarray.find(sub[, start[, end]])
```

Return the lowest index in the data where the subsequence *sub* is found, such that *sub* is contained in the slice *s[start:end]*. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Note: The `find()` method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the `in` operator:

```
>>> b'Py' in b'Python'  
True
```

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

```
bytes.index(sub[, start[, end]])  
bytarray.index(sub[, start[, end]])
```

Like `find()`, but raise `ValueError` when the subsequence is not found.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

```
bytes.join(iterable)  
bytarray.join(iterable)
```

Return a bytes or bytarray object which is the concatenation of the binary data sequences in *iterable*. A `TypeError` will be raised if there are any values in *iterable* that are not *bytes-like objects*, including `str` objects. The separator between elements is the contents of the bytes or bytarray object providing this method.

```
static bytes.maketrans(from, to)  
static bytarray.maketrans(from, to)
```

This static method returns a translation table usable for `bytes.translate()` that will map each character in *from* into the character at the same position in *to*; *from* and *to* must both be *bytes-like objects* and have the same length.

New in version 3.1.

```
bytes.partition(sep)  
bytarray.partition(sep)
```

Split the sequence at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself or its bytarray copy, and the part after the separator. If the separator is not found, return a 3-tuple containing a copy of the original sequence, followed by two empty bytes or bytarray objects.

The separator to search for may be any *bytes-like object*.

```
bytes.replace(old, new[, count])  
bytarray.replace(old, new[, count])
```

Return a copy of the sequence with all occurrences of subsequence *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

The subsequence to search for and its replacement may be any *bytes-like object*.

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
bytes.rfind(sub[, start[, end]])
```

`bytearray.rfind(sub[, start[, end]])`

Return the highest index in the sequence where the subsequence *sub* is found, such that *sub* is contained within *s* [start:end]. Optional arguments *start* and *end* are interpreted as in slice notation. Return -1 on failure.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

`bytes.rindex(sub[, start[, end]])`

`bytearray.rindex(sub[, start[, end]])`

Like `rfind()` but raises `ValueError` when the subsequence *sub* is not found.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

`bytes.rpartition(sep)`

`bytearray.rpartition(sep)`

Split the sequence at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself or its bytearray copy, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty bytes or bytearray objects, followed by a copy of the original sequence.

The separator to search for may be any *bytes-like object*.

`bytes.startswith(prefix[, start[, end]])`

`bytearray.startswith(prefix[, start[, end]])`

Return True if the binary data starts with the specified *prefix*, otherwise return False. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

The prefix(es) to search for may be any *bytes-like object*.

`bytes.translate(table, /, delete=b")`

`bytearray.translate(table, /, delete=b")`

Return a copy of the bytes or bytearray object where all bytes occurring in the optional argument *delete* are removed, and the remaining bytes have been mapped through the given translation table, which must be a bytes object of length 256.

You can use the `bytes.maketrans()` method to create a translation table.

Set the *table* argument to None for translations that only delete characters:

```
>>> b'read this short text'.translate(None, b'aeiou')
b'r d ths shrt txt'
```

Changed in version 3.6: *delete* is now supported as a keyword argument.

The following methods on bytes and bytearray objects have default behaviours that assume the use of ASCII compatible binary formats, but can still be used with arbitrary binary data by passing appropriate arguments. Note that all of the bytearray methods in this section do *not* operate in place, and instead produce new objects.

`bytes.center(width[, fillbyte])`

`bytearray.center(width[, fillbyte])`

Return a copy of the object centered in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
bytes.ljust(width[, fillbyte])
bytarray.ljust(width[, fillbyte])
```

Return a copy of the object left justified in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
bytes.lstrip([chars])
bytarray.lstrip([chars])
```

Return a copy of the sequence with specified leading bytes removed. The *chars* argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or `None`, the *chars* argument defaults to removing ASCII whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped:

```
>>> b'    spacious    '.lstrip()
b'spacious'
>>> b'www.example.com'.lstrip(b'cmowz.')
b'example.com'
```

The binary sequence of byte values to remove may be any *bytes-like object*.

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
bytes.rjust(width[, fillbyte])
bytarray.rjust(width[, fillbyte])
```

Return a copy of the object right justified in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
bytes.rsplit(sep=None, maxsplit=-1)
bytarray.rsplit(sep=None, maxsplit=-1)
```

Split the binary sequence into subsequences of the same type, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any subsequence consisting solely of ASCII whitespace is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below.

```
bytes.rstrip([chars])
bytarray.rstrip([chars])
```

Return a copy of the sequence with specified trailing bytes removed. The *chars* argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or `None`, the *chars* argument defaults to removing ASCII whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> b'    spacious    '.rstrip()
b'    spacious'
```

(continues on next page)

(continued from previous page)

```
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

The binary sequence of byte values to remove may be any *bytes-like object*.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.split(sep=None, maxsplit=-1)`
`bytearray.split(sep=None, maxsplit=-1)`

Split the binary sequence into subsequences of the same type, using *sep* as the delimiter string. If *maxsplit* is given and non-negative, at most *maxsplit* splits are done (thus, the list will have at most *maxsplit*+1 elements). If *maxsplit* is not specified or is *-1*, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty subsequences (for example, `b'1,,2'.split(b',')` returns `[b'1', b'', b'2']`). The *sep* argument may consist of a multibyte sequence (for example, `b'1<>2<>3'.split(b'<>')` returns `[b'1', b'2', b'3']`). Splitting an empty sequence with a specified separator returns `[b'']` or `[bytearray(b'')]` depending on the type of object being split. The *sep* argument may be any *bytes-like object*.

For example:

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,,2,,3,'.split(b',')
[b'1', b'2', b'', b'3', b'']
```

If *sep* is not specified or is *None*, a different splitting algorithm is applied: runs of consecutive ASCII whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the sequence has leading or trailing whitespace. Consequently, splitting an empty sequence or a sequence consisting solely of ASCII whitespace without a specified separator returns `[]`.

For example:

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b'    1    2    3    '.split()
[b'1', b'2', b'3']
```

`bytes.strip([chars])`
`bytearray.strip([chars])`

Return a copy of the sequence with specified leading and trailing bytes removed. The *chars* argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or *None*, the *chars* argument defaults to removing ASCII whitespace. The *chars* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```
>>> b'    spacious    '.strip()
b'spacious'
>>> b'www.example.com'.strip(b'cmowz.')
b'example'
```

The binary sequence of byte values to remove may be any *bytes-like object*.

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

The following methods on bytes and bytarray objects assume the use of ASCII compatible binary formats and should not be applied to arbitrary binary data. Note that all of the bytarray methods in this section do *not* operate in place, and instead produce new objects.

```
bytes.capitalize()
bytarray.capitalize()
```

Return a copy of the sequence with each byte interpreted as an ASCII character, and the first byte capitalized and the rest lowercased. Non-ASCII byte values are passed through unchanged.

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
bytes.expandtabs (tabsize=8)
bytarray.expandtabs (tabsize=8)
```

Return a copy of the sequence where all ASCII tab characters are replaced by one or more ASCII spaces, depending on the current column and the given tab size. Tab positions occur every *tabsize* bytes (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the sequence, the current column is set to zero and the sequence is examined byte by byte. If the byte is an ASCII tab character (b'\t'), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the current byte is an ASCII newline (b'\n') or carriage return (b'\r'), it is copied and the current column is reset to zero. Any other byte value is copied unchanged and the current column is incremented by one regardless of how the byte value is represented when printed:

```
>>> b'01\t012\t0123\t01234'.expandtabs()
b'01      012      0123    01234'
>>> b'01\t012\t0123\t01234'.expandtabs(4)
b'01  012  0123  01234'
```

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
bytes.isalnum()
bytarray.isalnum()
```

Return True if all bytes in the sequence are alphabetical ASCII characters or ASCII decimal digits and the sequence is not empty, False otherwise. Alphabetic ASCII characters are those byte values in the sequence b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'. ASCII decimal digits are those byte values in the sequence b'0123456789'.

For example:

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

```
bytes.isalpha()
bytarray.isalpha()
```

Return True if all bytes in the sequence are alphabetic ASCII characters and the sequence is

not empty, `False` otherwise. Alphabetic ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

For example:

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

`bytes.isascii()`
`bytearray.isascii()`

Return `True` if the sequence is empty or all bytes in the sequence are ASCII, `False` otherwise. ASCII bytes are in the range 0-0x7F.

New in version 3.7.

`bytes.isdigit()`
`bytearray.isdigit()`

Return `True` if all bytes in the sequence are ASCII decimal digits and the sequence is not empty, `False` otherwise. ASCII decimal digits are those byte values in the sequence `b'0123456789'`.

For example:

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

`bytes.islower()`
`bytearray.islower()`

Return `True` if there is at least one lowercase ASCII character in the sequence and no uppercase ASCII characters, `False` otherwise.

For example:

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

`bytes.isspace()`
`bytearray.isspace()`

Return `True` if all bytes in the sequence are ASCII whitespace and the sequence is not empty, `False` otherwise. ASCII whitespace characters are those byte values in the sequence `b' \t\n\r\x0b\f'` (space, tab, newline, carriage return, vertical tab, form feed).

`bytes.istitle()`
`bytearray.istitle()`

Return `True` if the sequence is ASCII titlecase and the sequence is not empty, `False` otherwise. See `bytes.title()` for more details on the definition of “titlecase”.

For example:

```
>>> b'Hello World'.istitle()
True
```

(continues on next page)

(continued from previous page)

```
>>> b'Hello world'.istitle()
False
```

`bytes.isupper()`
`bytearray.isupper()`

Return True if there is at least one uppercase alphabetic ASCII character in the sequence and no lowercase ASCII characters, False otherwise.

For example:

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

`bytes.lower()`
`bytearray.lower()`

Return a copy of the sequence with all the uppercase ASCII characters converted to their corresponding lowercase counterpart.

For example:

```
>>> b'Hello World'.lower()
b'hello world'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.splitlines(keepends=False)`
`bytearray.splitlines(keepends=False)`

Return a list of the lines in the binary sequence, breaking at ASCII line boundaries. This method uses the [universal newlines](#) approach to splitting lines. Line breaks are not included in the resulting list unless `keepends` is given and true.

For example:

```
>>> b'ab c\nnde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\nnde fg\rkl\r\n'.splitlines(keepends=True)
[b'ab c\n', b'\n', b'de fg\r', b'kl\r\n']
```

Unlike `split()` when a delimiter string `sep` is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([b''], [b'Two lines', b''])
>>> b"".splitlines(), b"One line\n".splitlines()
([], [b'One line'])
```

`bytes.swapcase()`

`bytearray.swapcase()`

Return a copy of the sequence with all the lowercase ASCII characters converted to their corresponding uppercase counterpart and vice-versa.

For example:

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Unlike `str.swapcase()`, it is always the case that `bin.swapcase().swapcase() == bin` for the binary versions. Case conversions are symmetrical in ASCII, even though that is not generally true for arbitrary Unicode code points.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.title()`

`bytearray.title()`

Return a titlecased version of the binary sequence where words start with an uppercase ASCII character and the remaining characters are lowercase. Uncased byte values are left unmodified.

For example:

```
>>> b'Hello world'.title()
b'Hello World'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. All other byte values are uncased.

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

```
>>> b"they're bill's friends from the UK".title()
b"They'Re Bill'S Friends From The Uk"
```

A workaround for apostrophes can be constructed using regular expressions:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(rb"[A-Za-z]+('[A-Za-z]+)?",
...                  lambda mo: mo.group(0)[0:1].upper() +
...                             mo.group(0)[1:].lower(),
...                  s)
...
>>> titlecase(b"they're bill's friends.")
b"They're Bill's Friends."
```

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.upper()`

`bytearray.upper()`

Return a copy of the sequence with all the lowercase ASCII characters converted to their corresponding uppercase counterpart.

For example:

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.zfill(width)`

`bytearray.zfill(width)`

Return a copy of the sequence left filled with ASCII `b'0'` digits to make a sequence of length `width`. A leading sign prefix (`b'+' / b'-'`) is handled by inserting the padding *after* the sign character rather than before. For `bytes` objects, the original sequence is returned if `width` is less than or equal to `len(seq)`.

For example:

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

4.8.4 printf-style Bytes Formatting

Note: The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). If the value being printed may be a tuple or dictionary, wrap it in a tuple.

Bytes objects (`bytes/bytarray`) have one unique built-in operation: the `%` operator (modulo). This is also known as the bytes *formatting* or *interpolation* operator. Given `format % values` (where `format` is a bytes object), `%` conversion specifications in `format` are replaced with zero or more elements of `values`. The effect is similar to using the `sprintf()` in the C language.

If `format` requires a single argument, `values` may be a single non-tuple object.⁵ Otherwise, `values` must be a tuple with exactly the number of items specified by the format bytes object, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The `'%'` character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, `(somename)`).
3. Conversion flags (optional), which affect the result of some conversion types.

4. Minimum field width (optional). If specified as an '*' (asterisk), the actual width is read from the next element of the tuple in *values*, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a '.' (dot) followed by the precision. If specified as '*' (an asterisk), the actual precision is read from the next element of the tuple in *values*, and the value to convert comes after the precision.
6. Length modifier (optional).
7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the bytes object *must* include a parenthesised mapping key into that dictionary inserted immediately after the '%' character. The mapping key selects the value to be formatted from the mapping. For example:

```
>>> print(b'%(%(language)s has %(number)03d quote types.' %
...       {b'language': b"Python", b"number": 2})
b'Python has 002 quote types.'
```

In this case no * specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

Flag	Meaning
' #'	The value conversion will use the “alternate form” (where defined below).
' 0 '	The conversion will be zero padded for numeric values.
' - '	The converted value is left adjusted (overrides the ' 0 ' conversion if both are given).
' '	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
' + '	A sign character (' +' or ' - ') will precede the conversion (overrides a “space” flag).

A length modifier (h, l, or L) may be present, but is ignored as it is not necessary for Python – so e.g. %ld is identical to %d.

The conversion types are:

Conversion	Meaning	Notes
'd'	Signed integer decimal.	
'i'	Signed integer decimal.	
'o'	Signed octal value.	(1)
'u'	Obsolete type – it is identical to 'd'.	(8)
'x'	Signed hexadecimal (lowercase).	(2)
'X'	Signed hexadecimal (uppercase).	(2)
'e'	Floating point exponential format (lowercase).	(3)
'E'	Floating point exponential format (uppercase).	(3)
'f'	Floating point decimal format.	(3)
'F'	Floating point decimal format.	(3)
'g'	Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'G'	Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'c'	Single byte (accepts integer or single byte objects).	
'b'	Bytes (any object that follows the buffer protocol or has <code>__bytes__()</code>).	(5)
's'	's' is an alias for 'b' and should only be used for Python2/3 code bases.	(6)
'a'	Bytes (converts any Python object using <code>repr(obj).encode('ascii', 'backslashreplace')</code>).	(5)
'r'	'r' is an alias for 'a' and should only be used for Python2/3 code bases.	(7)
'%'	No argument is converted, results in a '%' character in the result.	

Notes:

- (1) The alternate form causes a leading octal specifier ('0o') to be inserted before the first digit.
- (2) The alternate form causes a leading '0x' or '0X' (depending on whether the 'x' or 'X' format was used) to be inserted before the first digit.
- (3) The alternate form causes the result to always contain a decimal point, even if no digits follow it.
The precision determines the number of digits after the decimal point and defaults to 6.
- (4) The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.
The precision determines the number of significant digits before and after the decimal point and defaults to 6.
- (5) If precision is N, the output is truncated to N characters.
- (6) b'%s' is deprecated, but will not be removed during the 3.x series.
- (7) b'%r' is deprecated, but will not be removed during the 3.x series.
- (8) See [PEP 237](#).

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

See also:

[PEP 461](#) - Adding % formatting to bytes and bytearray

New in version 3.5.

4.8.5 Memory Views

`memoryview` objects allow Python code to access the internal data of an object that supports the buffer protocol without copying.

```
class memoryview(obj)
```

Create a `memoryview` that references `obj`. `obj` must support the buffer protocol. Built-in objects that support the buffer protocol include `bytes` and `bytearray`.

A `memoryview` has the notion of an *element*, which is the atomic memory unit handled by the originating object `obj`. For many simple types such as `bytes` and `bytearray`, an element is a single byte, but other types such as `array.array` may have bigger elements.

`len(view)` is equal to the length of `tolist`. If `view.ndim = 0`, the length is 1. If `view.ndim = 1`, the length is equal to the number of elements in the view. For higher dimensions, the length is equal to the length of the nested list representation of the view. The `itemsize` attribute will give you the number of bytes in a single element.

A `memoryview` supports slicing and indexing to expose its data. One-dimensional slicing will result in a subview:

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

If `format` is one of the native format specifiers from the `struct` module, indexing with an integer or a tuple of integers is also supported and returns a single *element* with the correct type. One-dimensional memoryviews can be indexed with an integer or a one-integer tuple. Multi-dimensional memoryviews can be indexed with tuples of exactly `ndim` integers where `ndim` is the number of dimensions. Zero-dimensional memoryviews can be indexed with the empty tuple.

Here is an example with a non-byte format:

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::-2].tolist()
[-11111111, -33333333]
```

If the underlying object is writable, the memoryview supports one-dimensional slice assignment. Resizing is not allowed:

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
```

(continues on next page)

(continued from previous page)

```
>>> v[1:4] = b'123'
>>> data
bytarray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different structures
>>> v[2:6] = b'spam'
>>> data
bytarray(b'z1spam')
```

One-dimensional memoryviews of hashable (read-only) types with formats ‘B’, ‘b’ or ‘c’ are also hashable. The hash is defined as `hash(m) == hash(m.tobytes())`:

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

Changed in version 3.3: One-dimensional memoryviews can now be sliced. One-dimensional memoryviews with formats ‘B’, ‘b’ or ‘c’ are now hashable.

Changed in version 3.4: `memoryview` is now registered automatically with `collections.abc.Sequence`

Changed in version 3.5: memoryviews can now be indexed with tuple of integers.

`memoryview` has several methods:

`__eq__(exporter)`

A memoryview and a [PEP 3118](#) exporter are equal if their shapes are equivalent and if all corresponding values are equal when the operands’ respective format codes are interpreted using `struct` syntax.

For the subset of `struct` format strings currently supported by `tolist()`, `v` and `w` are equal if `v.tolist() == w.tolist()`:

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

If either format string is not supported by the `struct` module, then the objects will always compare as unequal (even if the format strings and buffer contents are identical):

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

Note that, as with floating point numbers, `v is w` does *not* imply `v == w` for `memoryview` objects.

Changed in version 3.3: Previous versions compared the raw memory disregarding the item format and the logical array structure.

`tobytes(order=None)`

Return the data in the buffer as a bytestring. This is equivalent to calling the `bytes` constructor on the `memoryview`.

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

For non-contiguous arrays the result is equal to the flattened list representation with all elements converted to bytes. `tobytes()` supports all format strings, including those that are not in `struct` module syntax.

New in version 3.8: `order` can be {'C', 'F', 'A'}. When `order` is 'C' or 'F', the data of the original array is converted to C or Fortran order. For contiguous views, 'A' returns an exact copy of the physical memory. In particular, in-memory Fortran order is preserved. For non-contiguous views, the data is converted to C first. `order=None` is the same as `order='C'`.

`hex([sep[, bytes_per_sep]])`

Return a string object containing two hexadecimal digits for each byte in the buffer.

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

New in version 3.5.

Changed in version 3.8: Similar to `bytes.hex()`, `memoryview.hex()` now supports optional `sep` and `bytes_per_sep` parameters to insert separators between bytes in the hex output.

`tolist()`

Return the data in the buffer as a list of elements.

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

Changed in version 3.3: `tolist()` now supports all single character native formats in `struct` module syntax as well as multi-dimensional representations.

`toreadonly()`

Return a readonly version of the memoryview object. The original memoryview object is unchanged.

```
>>> m = memoryview(bytarray(b'abc'))
>>> mm = m.toreadonly()
>>> mm.tolist()
[89, 98, 99]
>>> mm[0] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> m[0] = 43
>>> mm.tolist()
[43, 98, 99]
```

New in version 3.8.

`release()`

Release the underlying buffer exposed by the memoryview object. Many objects take special actions when a view is held on them (for example, a `bytarray` would temporarily forbid resizing); therefore, calling `release()` is handy to remove these restrictions (and free any dangling resources) as soon as possible.

After this method has been called, any further operation on the view raises a `ValueError` (except `release()` itself which can be called multiple times):

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

The context management protocol can be used for a similar effect, using the `with` statement:

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

New in version 3.2.

`cast(format[, shape])`

Cast a memoryview to a new format or shape. `shape` defaults to `[byte_length//new_itemsize]`, which means that the result view will be one-dimensional. The return value is a new memoryview, but the buffer itself is not copied. Supported casts are 1D -> C-contiguous and C-contiguous -> 1D.

The destination format is restricted to a single element native format in `struct` syntax. One of the formats must be a byte format ('B', 'b' or 'c'). The byte length of the result must be the same as the original length.

Cast 1D/long to 1D/unsigned bytes:

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

Cast 1D/unsigned bytes to 1D/char:

```
>>> b = bytearray(b'xyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview: invalid value for format "B"
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

Cast 1D/bytes to 3D/ints to 1D/signed char:

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
```

(continues on next page)

(continued from previous page)

```
>>> z.nbytes
48
```

Cast 1D/unsigned long to 2D/unsigned long:

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

New in version 3.3.

Changed in version 3.5: The source format is no longer restricted when casting to a byte view.

There are also several readonly attributes available:

obj

The underlying object of the memoryview:

```
>>> b = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

New in version 3.3.

nbytes

`nbytes == product(shape) * itemsize == len(m.tobytes())`. This is the amount of space in bytes that the array would use in a contiguous representation. It is not necessarily equal to `len(m)`:

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

Multi-dimensional arrays:

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
```

(continues on next page)

(continued from previous page)

```
>>> len(y)
3
>>> y.nbytes
96
```

New in version 3.3.

readonly

A bool indicating whether the memory is read only.

format

A string containing the format (in `struct` module style) for each element in the view. A memoryview can be created from exporters with arbitrary format strings, but some methods (e.g. `tolist()`) are restricted to native single element formats.

Changed in version 3.3: format 'B' is now handled according to the struct module syntax. This means that `memoryview(b'abc')[0] == b'abc'[0] == 97.`

itemsize

The size in bytes of each element of the memoryview:

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim

An integer indicating how many dimensions of a multi-dimensional array the memory represents.

shape

A tuple of integers the length of `ndim` giving the shape of the memory as an N-dimensional array.

Changed in version 3.3: An empty tuple instead of `None` when `ndim = 0`.

strides

A tuple of integers the length of `ndim` giving the size in bytes to access each element for each dimension of the array.

Changed in version 3.3: An empty tuple instead of `None` when `ndim = 0`.

suboffsets

Used internally for PIL-style arrays. The value is informational only.

c_contiguous

A bool indicating whether the memory is C-*contiguous*.

New in version 3.3.

f_contiguous

A bool indicating whether the memory is Fortran *contiguous*.

New in version 3.3.

contiguous

A bool indicating whether the memory is *contiguous*.

New in version 3.3.

4.9 Set Types — `set`, `frozenset`

A `set` object is an unordered collection of distinct *hashable* objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference. (For other containers see the built-in `dict`, `list`, and `tuple` classes, and the `collections` module.)

Like other collections, sets support `x` in `set`, `len(set)`, and `for x in set`. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

There are currently two built-in set types, `set` and `frozenset`. The `set` type is mutable — the contents can be changed using methods like `add()` and `remove()`. Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. The `frozenset` type is immutable and *hashable* — its contents cannot be altered after it is created; it can therefore be used as a dictionary key or as an element of another set.

Non-empty sets (not frozensets) can be created by placing a comma-separated list of elements within braces, for example: `{'jack', 'sjoerd'}`, in addition to the `set` constructor.

The constructors for both classes work the same:

```
class set([iterable])
class frozenset([iterable])
```

Return a new set or frozenset object whose elements are taken from *iterable*. The elements of a set must be *hashable*.

To represent sets of sets, the inner sets must be `frozenset` objects. If *iterable* is not specified, a new empty set is returned.

Instances of `set` and `frozenset` provide the following operations:

`len(s)`

Return the number of elements in set *s* (cardinality of *s*).

`x in s`

Test *x* for membership in *s*.

`x not in s`

Test *x* for non-membership in *s*.

`isdisjoint(other)`

Return `True` if the set has no elements in common with *other*. Sets are disjoint if and only if their intersection is the empty set.

`issubset(other)`

`set <= other`

Test whether every element in the set is in *other*.

`set < other`

Test whether the set is a proper subset of *other*, that is, `set <= other` and `set != other`.

`issuperset(other)`

`set >= other`

Test whether every element in *other* is in the set.

`set > other`

Test whether the set is a proper superset of *other*, that is, `set >= other` and `set != other`.

`union(*others)`

`set | other | ...`

Return a new set with elements from the set and all others.

`intersection(*others)`

set & other & ...

Return a new set with elements common to the set and all others.

difference(*others)

set - other - ...

Return a new set with elements in the set that are not in the others.

symmetric_difference(other)

set ^ other

Return a new set with elements in either the set or *other* but not both.

copy()

Return a shallow copy of the set.

Note, the non-operator versions of `union()`, `intersection()`, `difference()`, and `symmetric_difference()`, `issubset()`, and `issuperset()` methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like `set('abc') & 'cbs'` in favor of the more readable `set('abc').intersection('cbs')`.

Both `set` and `frozenset` support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

Instances of `set` are compared to instances of `frozenset` based on their members. For example, `set('abc') == frozenset('abc')` returns `True` and so does `set('abc') in set([frozenset('abc')])`.

The subset and equality comparisons do not generalize to a total ordering function. For example, any two nonempty disjoint sets are not equal and are not subsets of each other, so *all* of the following return `False`: `a < b`, `a == b`, or `a > b`.

Since sets only define partial ordering (subset relationships), the output of the `list.sort()` method is undefined for lists of sets.

Set elements, like dictionary keys, must be *hashable*.

Binary operations that mix `set` instances with `frozenset` return the type of the first operand. For example: `frozenset('ab') | set('bc')` returns an instance of `frozenset`.

The following table lists operations available for `set` that do not apply to immutable instances of `frozenset`:

update(*others)

set |= other | ...

Update the set, adding elements from all others.

intersection_update(*others)

set &= other & ...

Update the set, keeping only elements found in it and all others.

difference_update(*others)

set -= other | ...

Update the set, removing elements found in others.

symmetric_difference_update(other)

set ^= other

Update the set, keeping only elements found in either set, but not in both.

add(elem)

Add element *elem* to the set.

`remove (elem)`

Remove element *elem* from the set. Raises `KeyError` if *elem* is not contained in the set.

`discard (elem)`

Remove element *elem* from the set if it is present.

`pop ()`

Remove and return an arbitrary element from the set. Raises `KeyError` if the set is empty.

`clear ()`

Remove all elements from the set.

Note, the non-operator versions of the `update()`, `intersection_update()`, `difference_update()`, and `symmetric_difference_update()` methods will accept any iterable as an argument.

Note, the *elem* argument to the `__contains__()`, `remove()`, and `discard()` methods may be a set. To support searching for an equivalent frozenset, a temporary one is created from *elem*.

4.10 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(**kwargs)
class dict(mapping, **kwargs)
class dict(iterable, **kwargs)
```

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 from the positional argument. If a key being added is already present, the value from the keyword argument replaces the value from the positional argument.

To illustrate, the following examples all return a dictionary equal to `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
```

(continues on next page)

(continued from previous page)

```
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Providing keyword arguments as in the first example only works for keys that are valid Python identifiers. Otherwise, any valid keys can be used.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

list(d)

Return a list of all the keys used in the dictionary *d*.

len(d)

Return the number of items in the dictionary *d*.

d[key]

Return the item of *d* with key *key*. Raises a *KeyError* if *key* is not in the map.

If a subclass of dict defines a method `__missing__(self, key)` and *key* is not present, the `d[key]` operation calls that method with the key *key* as argument. The `d[key]` operation then returns or raises whatever is returned or raised by the `__missing__(key)` call. No other operations or methods invoke `__missing__(self, key)`. If `__missing__(self, key)` is not defined, *KeyError* is raised. `__missing__(self, key)` must be a method; it cannot be an instance variable:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

The example above shows part of the implementation of `collections.Counter`. A different `__missing__(self, key)` method is used by `collections.defaultdict`.

d[key] = value

Set `d[key]` to *value*.

del d[key]

Remove `d[key]` from *d*. Raises a *KeyError* if *key* is not in the map.

key in d

Return True if *d* has a key *key*, else False.

key not in d

Equivalent to `not key in d`.

iter(d)

Return an iterator over the keys of the dictionary. This is a shortcut for `iter(d.keys())`.

clear()

Remove all items from the dictionary.

copy()

Return a shallow copy of the dictionary.

classmethod fromkeys (iterable[, value])

Create a new dictionary with keys from *iterable* and values set to *value*.

fromkeys () is a class method that returns a new dictionary. *value* defaults to `None`. All of the values refer to just a single instance, so it generally doesn't make sense for *value* to be a mutable object such as an empty list. To get distinct values, use a dict comprehension instead.

get (key[, default])

Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

items ()

Return a new view of the dictionary's items ((key, value) pairs). See the [documentation of view objects](#).

keys ()

Return a new view of the dictionary's keys. See the [documentation of view objects](#).

pop (key[, default])

If *key* is in the dictionary, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a `KeyError` is raised.

popitem ()

Remove and return a (key, value) pair from the dictionary. Pairs are returned in LIFO (last-in, first-out) order.

popitem () is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling *popitem ()* raises a `KeyError`.

Changed in version 3.7: LIFO order is now guaranteed. In prior versions, *popitem ()* would return an arbitrary key/value pair.

reversed (d)

Return a reverse iterator over the keys of the dictionary. This is a shortcut for `reversed (d.keys ())`.

New in version 3.8.

setdefault (key[, default])

If *key* is in the dictionary, return its value. If not, insert *key* with a value of *default* and return *default*. *default* defaults to `None`.

update ([other])

Update the dictionary with the key/value pairs from *other*, overwriting existing keys. Return `None`.

update () accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: `d.update (red=1, blue=2)`.

values ()

Return a new view of the dictionary's values. See the [documentation of view objects](#).

An equality comparison between one `dict.values ()` view and another will always return `False`. This also applies when comparing `dict.values ()` to itself:

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

Dictionaries compare equal if and only if they have the same (key, value) pairs (regardless of ordering). Order comparisons ('<', '<=', '>=' , '>') raise `TypeError`.

Dictionaries preserve insertion order. Note that updating a key does not affect the order. Keys added after deletion are inserted at the end.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

Changed in version 3.7: Dictionary order is guaranteed to be insertion order. This behavior was an implementation detail of CPython from 3.6.

Dictionaries and dictionary views are reversible.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(reversed(d))
['four', 'three', 'two', 'one']
>>> list(reversed(d.values()))
[4, 3, 2, 1]
>>> list(reversed(d.items()))
[('four', 4), ('three', 3), ('two', 2), ('one', 1)]
```

Changed in version 3.8: Dictionaries are now reversible.

See also:

`types.MappingProxyType` can be used to create a read-only view of a `dict`.

4.10.1 Dictionary view objects

The objects returned by `dict.keys()`, `dict.values()` and `dict.items()` are *view objects*. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

Dictionary views can be iterated over to yield their respective data, and support membership tests:

`len(dictview)`

Return the number of entries in the dictionary.

`iter(dictview)`

Return an iterator over the keys, values or items (represented as tuples of `(key, value)`) in the dictionary.

Keys and values are iterated over in insertion order. This allows the creation of `(value, key)` pairs using `zip()`: `pairs = zip(d.values(), d.keys())`. Another way to create the same list is `pairs = [(v, k) for (k, v) in d.items()]`.

Iterating views while adding or deleting entries in the dictionary may raise a `RuntimeError` or fail to iterate over all entries.

Changed in version 3.7: Dictionary order is guaranteed to be insertion order.

x in dictview

Return True if *x* is in the underlying dictionary's keys, values or items (in the latter case, *x* should be a (key, value) tuple).

reversed(dictview)

Return a reverse iterator over the keys, values or items of the dictionary. The view will be iterated in reverse order of the insertion.

Changed in version 3.8: Dictionary views are now reversible.

Keys views are set-like since their entries are unique and hashable. If all values are hashable, so that (key, value) pairs are unique and hashable, then the items view is also set-like. (Values views are not treated as set-like since the entries are generally not unique.) For set-like views, all of the operations defined for the abstract base class `collections.abc.Set` are available (for example, ==, <, or ^).

An example of dictionary view usage:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order (insertion order)
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```

4.11 Context Manager Types

Python's `with` statement supports the concept of a runtime context defined by a context manager. This is implemented using a pair of methods that allow user-defined classes to define a runtime context that is entered before the statement body is executed and exited when the statement ends:

`contextmanager.__enter__()`

Enter the runtime context and return either this object or another object related to the runtime context. The value returned by this method is bound to the identifier in the `as` clause of `with` statements using this context manager.

An example of a context manager that returns itself is a [file object](#). File objects return themselves from `__enter__()` to allow `open()` to be used as the context expression in a `with` statement.

An example of a context manager that returns a related object is the one returned by `decimal.localcontext()`. These managers set the active decimal context to a copy of the original decimal context and then return the copy. This allows changes to be made to the current decimal context in the body of the `with` statement without affecting code outside the `with` statement.

`contextmanager.__exit__(exc_type, exc_val, exc_tb)`

Exit the runtime context and return a Boolean flag indicating if any exception that occurred should be suppressed. If an exception occurred while executing the body of the `with` statement, the arguments contain the exception type, value and traceback information. Otherwise, all three arguments are `None`.

Returning a true value from this method will cause the `with` statement to suppress the exception and continue execution with the statement immediately following the `with` statement. Otherwise the exception continues propagating after this method has finished executing. Exceptions that occur during execution of this method will replace any exception that occurred in the body of the `with` statement.

The exception passed in should never be reraised explicitly - instead, this method should return a false value to indicate that the method completed successfully and does not want to suppress the raised exception. This allows context management code to easily detect whether or not an `__exit__()` method has actually failed.

Python defines several context managers to support easy thread synchronisation, prompt closure of files or other objects, and simpler manipulation of the active decimal arithmetic context. The specific types are not treated specially beyond their implementation of the context management protocol. See the [contextlib](#) module for some examples.

Python's [generators](#) and the [contextlib.contextmanager](#) decorator provide a convenient way to implement these protocols. If a generator function is decorated with the [contextlib.contextmanager](#) decorator, it will return a context manager implementing the necessary `__enter__()` and `__exit__()` methods, rather than the iterator produced by an undecorated generator function.

Note that there is no specific slot for any of these methods in the type structure for Python objects in the Python/C API. Extension types wanting to define these methods must provide them as a normal Python accessible method. Compared to the overhead of setting up the runtime context, the overhead of a single class dictionary lookup is negligible.

4.12 Other Built-in Types

The interpreter supports several other kinds of objects. Most of these support only one or two operations.

4.12.1 Modules

The only special operation on a module is attribute access: `m.name`, where `m` is a module and `name` accesses a name defined in `m`'s symbol table. Module attributes can be assigned to. (Note that the `import` statement is not, strictly speaking, an operation on a module object; `import foo` does not require a module object named `foo` to exist, rather it requires an (external) `definition` for a module named `foo` somewhere.)

A special attribute of every module is `__dict__`. This is the dictionary containing the module's symbol table. Modifying this dictionary will actually change the module's symbol table, but direct assignment to the `__dict__` attribute is not possible (you can write `m.__dict__['a'] = 1`, which defines `m.a` to be 1, but you can't write `m.__dict__ = {}`). Modifying `__dict__` directly is not recommended.

Modules built into the interpreter are written like this: `<module 'sys' (built-in)>`. If loaded from a file, they are written as `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`.

4.12.2 Classes and Class Instances

See objects and class for these.

4.12.3 Functions

Function objects are created by function definitions. The only operation on a function object is to call it: `func(argument-list)`.

There are really two flavors of function objects: built-in functions and user-defined functions. Both support the same operation (to call the function), but the implementation is different, hence the different object types.

See function for more information.

4.12.4 Methods

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as `append()` on lists) and class instance methods. Built-in methods are described with the types that support them.

If you access a method (a function defined in a class namespace) through an instance, you get a special object: a *bound method* (also called *instance method*) object. When called, it will add the `self` argument to the argument list. Bound methods have two special read-only attributes: `m.__self__` is the object on which the method operates, and `m.__func__` is the function implementing the method. Calling `m(arg-1, arg-2, ..., arg-n)` is completely equivalent to calling `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`.

Like function objects, bound method objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (`meth.__func__`), setting method attributes on bound methods is disallowed. Attempting to set an attribute on a method results in an `AttributeError` being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method' # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

See types for more information.

4.12.5 Code Objects

Code objects are used by the implementation to represent “pseudo-compiled” executable Python code such as a function body. They differ from function objects because they don’t contain a reference to their global execution environment. Code objects are returned by the built-in `compile()` function and can be extracted from function objects through their `__code__` attribute. See also the `code` module.

A code object can be executed or evaluated by passing it (instead of a source string) to the `exec()` or `eval()` built-in functions.

See types for more information.

4.12.6 Type Objects

Type objects represent the various object types. An object’s type is accessed by the built-in function `type()`. There are no special operations on types. The standard module `types` defines names for all standard built-in types.

Types are written like this: `<class 'int'>`.

4.12.7 The Null Object

This object is returned by functions that don’t explicitly return a value. It supports no special operations. There is exactly one null object, named `None` (a built-in name). `type(None)()` produces the same singleton.

It is written as `None`.

4.12.8 The Ellipsis Object

This object is commonly used by slicing (see slicings). It supports no special operations. There is exactly one ellipsis object, named `Ellipsis` (a built-in name). `type(Ellipsis)()` produces the `Ellipsis` singleton.

It is written as `Ellipsis` or `...`.

4.12.9 The NotImplemented Object

This object is returned from comparisons and binary operations when they are asked to operate on types they don’t support. See comparisons for more information. There is exactly one `NotImplemented` object. `type(NotImplemented)()` produces the singleton instance.

It is written as `NotImplemented`.

4.12.10 Boolean Values

Boolean values are the two constant objects `False` and `True`. They are used to represent truth values (although other values can also be considered false or true). In numeric contexts (for example when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively. The built-in function `bool()` can be used to convert any value to a Boolean, if the value can be interpreted as a truth value (see section *Truth Value Testing* above).

They are written as `False` and `True`, respectively.

4.12.11 Internal Objects

See types for this information. It describes stack frame objects, traceback objects, and slice objects.

4.13 Special Attributes

The implementation adds a few special read-only attributes to several object types, where they are relevant. Some of these are not reported by the `dir()` built-in function.

`object.__dict__`

A dictionary or other mapping object used to store an object's (writable) attributes.

`instance.__class__`

The class to which a class instance belongs.

`class.__bases__`

The tuple of base classes of a class object.

`definition.__name__`

The name of the class, function, method, descriptor, or generator instance.

`definition.__qualname__`

The *qualified name* of the class, function, method, descriptor, or generator instance.

New in version 3.3.

`class.__mro__`

This attribute is a tuple of classes that are considered when looking for base classes during method resolution.

`class.mro()`

This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class instantiation, and its result is stored in `__mro__`.

`class.__subclasses__()`

Each class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. Example:

```
>>> int.__subclasses__()  
[<class 'bool'>]
```


BUILT-IN EXCEPTIONS

In Python, all exceptions must be instances of a class that derives from `BaseException`. In a `try` statement with an `except` clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which *it* is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an “associated value” indicating the detailed cause of the error. This may be a string or a tuple of several items of information (e.g., an error code and a string explaining the code). The associated value is usually passed as arguments to the exception class’s constructor.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition “just like” the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be subclassed to define new exceptions; programmers are encouraged to derive new exceptions from the `Exception` class or one of its subclasses, and not from `BaseException`. More information on defining exceptions is available in the Python Tutorial under `tut-userexceptions`.

When raising (or re-raising) an exception in an `except` or `finally` clause `__context__` is automatically set to the last exception caught; if the new exception is not handled the traceback that is eventually displayed will include the originating exception(s) and the final exception.

When raising a new exception (rather than using a bare `raise` to re-raise the exception currently being handled), the implicit exception context can be supplemented with an explicit cause by using `from` with `raise`:

```
raise new_exc from original_exc
```

The expression following `from` must be an exception or `None`. It will be set as `__cause__` on the raised exception. Setting `__cause__` also implicitly sets the `__suppress_context__` attribute to `True`, so that using `raise new_exc from None` effectively replaces the old exception with the new one for display purposes (e.g. converting `KeyError` to `AttributeError`), while leaving the old exception available in `__context__` for introspection when debugging.

The default traceback display code shows these chained exceptions in addition to the traceback for the exception itself. An explicitly chained exception in `__cause__` is always shown when present. An implicitly chained exception in `__context__` is shown only if `__cause__` is `None` and `__suppress_context__` is `false`.

In either case, the exception itself is always shown after any chained exceptions so that the final line of the traceback always shows the last exception that was raised.

5.1 Base classes

The following exceptions are used mostly as base classes for other exceptions.

`exception BaseException`

The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that, use `Exception`). If `str()` is called on an instance of this class, the representation of the argument(s) to the instance are returned, or the empty string when there were no arguments.

`args`

The tuple of arguments given to the exception constructor. Some built-in exceptions (like `OSError`) expect a certain number of arguments and assign a special meaning to the elements of this tuple, while others are usually called only with a single string giving an error message.

`with_traceback(tb)`

This method sets `tb` as the new traceback for the exception and returns the exception object. It is usually used in exception handling code like this:

```
try:  
    ...  
except SomeException:  
    tb = sys.exc_info()[2]  
    raise OtherException(...).with_traceback(tb)
```

`exception Exception`

All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.

`exception ArithmeticError`

The base class for those built-in exceptions that are raised for various arithmetic errors: `OverflowError`, `ZeroDivisionError`, `FloatingPointError`.

`exception BufferError`

Raised when a buffer related operation cannot be performed.

`exception LookupError`

The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: `IndexError`, `KeyError`. This can be raised directly by `codecs.lookup()`.

5.2 Concrete exceptions

The following exceptions are the exceptions that are usually raised.

`exception AssertionError`

Raised when an `assert` statement fails.

`exception AttributeError`

Raised when an attribute reference (see attribute-references) or assignment fails. (When an object does not support attribute references or attribute assignments at all, `TypeError` is raised.)

`exception EOFError`

Raised when the `input()` function hits an end-of-file condition (EOF) without reading any data. (N.B.: the `io.IOBase.read()` and `io.IOBase.readline()` methods return an empty string when they hit EOF.)

`exception FloatingPointError`

Not currently used.

exception GeneratorExit

Raised when a [generator](#) or [coroutine](#) is closed; see `generator.close()` and `coroutine.close()`. It directly inherits from [BaseException](#) instead of [Exception](#) since it is technically not an error.

exception ImportError

Raised when the `import` statement has troubles trying to load a module. Also raised when the “from list” in `from ... import` has a name that cannot be found.

The `name` and `path` attributes can be set using keyword-only arguments to the constructor. When set they represent the name of the module that was attempted to be imported and the path to any file which triggered the exception, respectively.

Changed in version 3.3: Added the `name` and `path` attributes.

exception ModuleNotFoundError

A subclass of [ImportError](#) which is raised by `import` when a module could not be located. It is also raised when `None` is found in `sys.modules`.

New in version 3.6.

exception IndexError

Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not an integer, [TypeError](#) is raised.)

exception KeyError

Raised when a mapping (dictionary) key is not found in the set of existing keys.

exception KeyboardInterrupt

Raised when the user hits the interrupt key (normally Control-C or Delete). During execution, a check for interrupts is made regularly. The exception inherits from [BaseException](#) so as to not be accidentally caught by code that catches [Exception](#) and thus prevent the interpreter from exiting.

exception MemoryError

Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C’s `malloc()` function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

exception NameError

Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

exception NotImplementedError

This exception is derived from [RuntimeError](#). In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method, or while the class is being developed to indicate that the real implementation still needs to be added.

Note: It should not be used to indicate that an operator or method is not meant to be supported at all – in that case either leave the operator / method undefined or, if a subclass, set it to `None`.

Note: `NotImplementedError` and `NotImplemented` are not interchangeable, even though they have similar names and purposes. See [NotImplemented](#) for details on when to use it.

exception OSError([arg])

exception OSError(*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]])

This exception is raised when a system function returns a system-related error, including I/O failures such as “file not found” or “disk full” (not for illegal argument types or other incidental errors).

The second form of the constructor sets the corresponding attributes, described below. The attributes default to *None* if not specified. For backwards compatibility, if three arguments are passed, the *args* attribute contains only a 2-tuple of the first two constructor arguments.

The constructor often actually returns a subclass of *OSError*, as described in *OS exceptions* below. The particular subclass depends on the final *errno* value. This behaviour only occurs when constructing *OSError* directly or via an alias, and is not inherited when subclassing.

errno

A numeric error code from the C variable *errno*.

winerror

Under Windows, this gives you the native Windows error code. The *errno* attribute is then an approximate translation, in POSIX terms, of that native error code.

Under Windows, if the *winerror* constructor argument is an integer, the *errno* attribute is determined from the Windows error code, and the *errno* argument is ignored. On other platforms, the *winerror* argument is ignored, and the *winerror* attribute does not exist.

strerror

The corresponding error message, as provided by the operating system. It is formatted by the C functions *perror()* under POSIX, and *FormatMessage()* under Windows.

filename**filename2**

For exceptions that involve a file system path (such as *open()* or *os.unlink()*), *filename* is the file name passed to the function. For functions that involve two file system paths (such as *os.rename()*), *filename2* corresponds to the second file name passed to the function.

Changed in version 3.3: *EnvironmentError*, *IOError*, *WindowsError*, *socket.error*, *select.error* and *mmap.error* have been merged into *OSError*, and the constructor may return a subclass.

Changed in version 3.4: The *filename* attribute is now the original file name passed to the function, instead of the name encoded to or decoded from the filesystem encoding. Also, the *filename2* constructor argument and attribute was added.

exception OverflowError

Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for integers (which would rather raise *MemoryError* than give up). However, for historical reasons, *OverflowError* is sometimes raised for integers that are outside a required range. Because of the lack of standardization of floating point exception handling in C, most floating point operations are not checked.

exception RecursionError

This exception is derived from *RuntimeError*. It is raised when the interpreter detects that the maximum recursion depth (see *sys.getrecursionlimit()*) is exceeded.

New in version 3.5: Previously, a plain *RuntimeError* was raised.

exception ReferenceError

This exception is raised when a weak reference proxy, created by the *weakref.proxy()* function, is used to access an attribute of the referent after it has been garbage collected. For more information on weak references, see the *weakref* module.

exception RuntimeError

Raised when an error is detected that doesn’t fall in any of the other categories. The associated value is a string indicating what precisely went wrong.

exception StopIteration

Raised by built-in function `next()` and an *iterator*'s `__next__()` method to signal that there are no further items produced by the iterator.

The exception object has a single attribute `value`, which is given as an argument when constructing the exception, and defaults to `None`.

When a *generator* or *coroutine* function returns, a new `StopIteration` instance is raised, and the value returned by the function is used as the `value` parameter to the constructor of the exception.

If a generator code directly or indirectly raises `StopIteration`, it is converted into a `RuntimeError` (retaining the `StopIteration` as the new exception's cause).

Changed in version 3.3: Added `value` attribute and the ability for generator functions to use it to return a value.

Changed in version 3.5: Introduced the `RuntimeError` transformation via `from __future__ import generator_stop`, see [PEP 479](#).

Changed in version 3.7: Enable [PEP 479](#) for all code by default: a `StopIteration` error raised in a generator is transformed into a `RuntimeError`.

exception StopAsyncIteration

Must be raised by `__anext__()` method of an *asynchronous iterator* object to stop the iteration.

New in version 3.5.

exception SyntaxError

Raised when the parser encounters a syntax error. This may occur in an `import` statement, in a call to the built-in functions `exec()` or `eval()`, or when reading the initial script or standard input (also interactively).

Instances of this class have attributes `filename`, `lineno`, `offset` and `text` for easier access to the details. `str()` of the exception instance returns only the message.

exception IndentationError

Base class for syntax errors related to incorrect indentation. This is a subclass of `SyntaxError`.

exception TabError

Raised when indentation contains an inconsistent use of tabs and spaces. This is a subclass of `IndentationError`.

exception SystemError

Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms).

You should report this to the author or maintainer of your Python interpreter. Be sure to report the version of the Python interpreter (`sys.version`; it is also printed at the start of an interactive Python session), the exact error message (the exception's associated value) and if possible the source of the program that triggered the error.

exception SystemExit

This exception is raised by the `sys.exit()` function. It inherits from `BaseException` instead of `Exception` so that it is not accidentally caught by code that catches `Exception`. This allows the exception to properly propagate up and cause the interpreter to exit. When it is not handled, the Python interpreter exits; no stack traceback is printed. The constructor accepts the same optional argument passed to `sys.exit()`. If the value is an integer, it specifies the system exit status (passed to C's `exit()` function); if it is `None`, the exit status is zero; if it has another type (such as a string), the object's value is printed and the exit status is one.

A call to `sys.exit()` is translated into an exception so that clean-up handlers (`finally` clauses of `try` statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The `os._exit()` function can be used if it is absolutely positively necessary to exit immediately (for example, in the child process after a call to `os.fork()`).

code

The exit status or error message that is passed to the constructor. (Defaults to None.)

exception TypeError

Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

This exception may be raised by user code to indicate that an attempted operation on an object is not supported, and is not meant to be. If an object is meant to support a given operation but has not yet provided an implementation, *NotImplementedError* is the proper exception to raise.

Passing arguments of the wrong type (e.g. passing a *list* when an *int* is expected) should result in a *TypeError*, but passing arguments with the wrong value (e.g. a number outside expected boundaries) should result in a *ValueError*.

exception UnboundLocalError

Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of *NameError*.

exception UnicodeError

Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of *ValueError*.

UnicodeError has attributes that describe the encoding or decoding error. For example, `err.object[err.start:err.end]` gives the particular invalid input that the codec failed on.

encoding

The name of the encoding that raised the error.

reason

A string describing the specific codec error.

object

The object the codec was attempting to encode or decode.

start

The first index of invalid data in *object*.

end

The index after the last invalid data in *object*.

exception UnicodeEncodeError

Raised when a Unicode-related error occurs during encoding. It is a subclass of *UnicodeError*.

exception UnicodeDecodeError

Raised when a Unicode-related error occurs during decoding. It is a subclass of *UnicodeError*.

exception UnicodeTranslateError

Raised when a Unicode-related error occurs during translating. It is a subclass of *UnicodeError*.

exception ValueError

Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as *IndexError*.

exception ZeroDivisionError

Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

The following exceptions are kept for compatibility with previous versions; starting from Python 3.3, they are aliases of *OSError*.

exception EnvironmentError

exception IOError

exception WindowsError

Only available on Windows.

5.2.1 OS exceptions

The following exceptions are subclasses of [OSError](#), they get raised depending on the system error code.

exception BlockingIOError

Raised when an operation would block on an object (e.g. socket) set for non-blocking operation. Corresponds to `errno EAGAIN`, `EALREADY`, `EWOULD_BLOCK` and `EINPROGRESS`.

In addition to those of [OSError](#), [BlockingIOError](#) can have one more attribute:

characters_written

An integer containing the number of characters written to the stream before it blocked. This attribute is available when using the buffered I/O classes from the [io](#) module.

exception ChildProcessError

Raised when an operation on a child process failed. Corresponds to `errno ECHILD`.

exception ConnectionError

A base class for connection-related issues.

Subclasses are [BrokenPipeError](#), [ConnectionAbortedError](#), [ConnectionRefusedError](#) and [ConnectionResetError](#).

exception BrokenPipeError

A subclass of [ConnectionError](#), raised when trying to write on a pipe while the other end has been closed, or trying to write on a socket which has been shutdown for writing. Corresponds to `errno EPIPE` and `ESHUTDOWN`.

exception ConnectionAbortedError

A subclass of [ConnectionError](#), raised when a connection attempt is aborted by the peer. Corresponds to `errno ECONNABORTED`.

exception ConnectionRefusedError

A subclass of [ConnectionError](#), raised when a connection attempt is refused by the peer. Corresponds to `errno ECONNREFUSED`.

exception ConnectionResetError

A subclass of [ConnectionError](#), raised when a connection is reset by the peer. Corresponds to `errno ECONNRESET`.

exception FileExistsError

Raised when trying to create a file or directory which already exists. Corresponds to `errno EEXIST`.

exception FileNotFoundError

Raised when a file or directory is requested but doesn't exist. Corresponds to `errno ENOENT`.

exception InterruptedError

Raised when a system call is interrupted by an incoming signal. Corresponds to `errno EINTR`.

Changed in version 3.5: Python now retries system calls when a syscall is interrupted by a signal, except if the signal handler raises an exception (see [PEP 475](#) for the rationale), instead of raising [InterruptedError](#).

exception IsADirectoryError

Raised when a file operation (such as `os.remove()`) is requested on a directory. Corresponds to `errno EISDIR`.

exception NotADirectoryError

Raised when a directory operation (such as `os.listdir()`) is requested on something which is not a directory. Corresponds to `errno ENOTDIR`.

exception `PermissionError`

Raised when trying to run an operation without the adequate access rights - for example filesystem permissions.
Corresponds to `errno EACCES` and `EPERM`.

exception `ProcessLookupError`

Raised when a given process doesn't exist. Corresponds to `errno ESRCH`.

exception `TimeoutError`

Raised when a system function timed out at the system level. Corresponds to `errno ETIMEDOUT`.

New in version 3.3: All the above `OSError` subclasses were added.

See also:

[PEP 3151](#) - Reworking the OS and IO exception hierarchy

5.3 Warnings

The following exceptions are used as warning categories; see the [Warning Categories](#) documentation for more details.

exception `Warning`

Base class for warning categories.

exception `UserWarning`

Base class for warnings generated by user code.

exception `DeprecationWarning`

Base class for warnings about deprecated features when those warnings are intended for other Python developers.

exception `PendingDeprecationWarning`

Base class for warnings about features which are obsolete and expected to be deprecated in the future, but are not deprecated at the moment.

This class is rarely used as emitting a warning about a possible upcoming deprecation is unusual, and `DeprecationWarning` is preferred for already active deprecations.

exception `SyntaxWarning`

Base class for warnings about dubious syntax.

exception `RuntimeWarning`

Base class for warnings about dubious runtime behavior.

exception `FutureWarning`

Base class for warnings about deprecated features when those warnings are intended for end users of applications that are written in Python.

exception `ImportWarning`

Base class for warnings about probable mistakes in module imports.

exception `UnicodeWarning`

Base class for warnings related to Unicode.

exception `BytesWarning`

Base class for warnings related to `bytes` and `bytearray`.

exception `ResourceWarning`

Base class for warnings related to resource usage. Ignored by the default warning filters.

New in version 3.2.

5.4 Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        |    +-- FloatingPointError
        |    +-- OverflowError
        |    +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
        |    +-- ModuleNotFoundError
    +-- LookupError
        |    +-- IndexError
        |    +-- KeyError
    +-- MemoryError
    +-- NameError
        |    +-- UnboundLocalError
    +-- OSError
        |    +-- BlockingIOError
        |    +-- ChildProcessError
        |    +-- ConnectionError
            |    +-- BrokenPipeError
            |    +-- ConnectionAbortedError
            |    +-- ConnectionRefusedError
            |    +-- ConnectionResetError
        |    +-- FileExistsError
        |    +-- FileNotFoundError
        |    +-- InterruptedError
        |    +-- IsADirectoryError
        |    +-- NotADirectoryError
        |    +-- PermissionError
        |    +-- ProcessLookupError
        |    +-- TimeoutError
    +-- ReferenceError
    +-- RuntimeError
        |    +-- NotImplementedError
        |    +-- RecursionError
    +-- SyntaxError
        |    +-- IndentationError
            |    +-- TabError
    +-- SystemError
    +-- TypeError
    +-- ValueError
        |    +-- UnicodeError
            |    +-- UnicodeDecodeError
            |    +-- UnicodeEncodeError
            |    +-- UnicodeTranslateError
```

(continues on next page)

(continued from previous page)

```
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning
```



Astronomy ESFRI & Research Infrastructure Cluster
ASTERICS - 653477



2nd ASTERICS-OBELICS International School

4-8 June 2018, Annecy, France.



H2020-Astronomy ESFRI and Research Infrastructure Cluster (Grant Agreement number: 653477).

PYTHON LIBRARIES

Tamás Gál
tamas.gal@fau.de



<https://github.com/tamasgal>



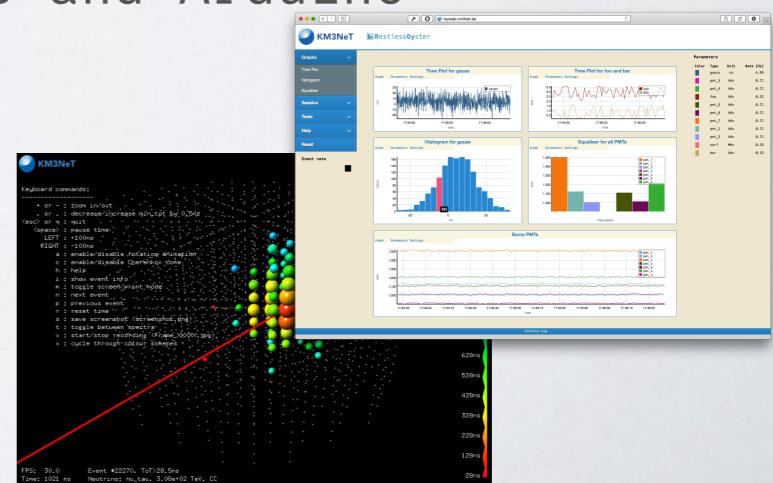
@tamasgal

OVERVIEW

- Who is this clown?
 - Python Introduction
 - Basic Python Internals
 - Libraries and Tools for Scientific Computing
 - NumPy
 - Numba
 - NumExpr
 - SciPy
 - AstroPy
 - Pandas
 - SymPy
 - Matplotlib
 - Jupyter
 - IPython
-
- { Make it faster!
- { Tools for scientists!

WHO IS THIS CLOWN?

- Tamás Gál, born 1985 in Debrecen (Hungary)
- PhD candidate in astro particle physics at Erlangen Centre for Astroparticle Physics (ECAP) working on the KM3NeT project
- Programming background:
 - Coding enthusiast since ~1993
 - First real application written in Amiga Basic (toilet manager, tons of GOTOs ;)
 - Python, Julia, JavaScript and C/C++/Obj-C for work
 - Haskell for fun
 - Earlier also Java, Perl, PHP, Delphi, MATLAB, whatsoever...
 - I also like playing around with integrated circuits and Arduino
- Some related projects:
 - KM3Pipe (analysis framework in the KM3NeT experiment),
 - RainbowAlga (interactive 3D neutrino event display),
 - ROyWeb (interactive realtime visualisation/graphing)

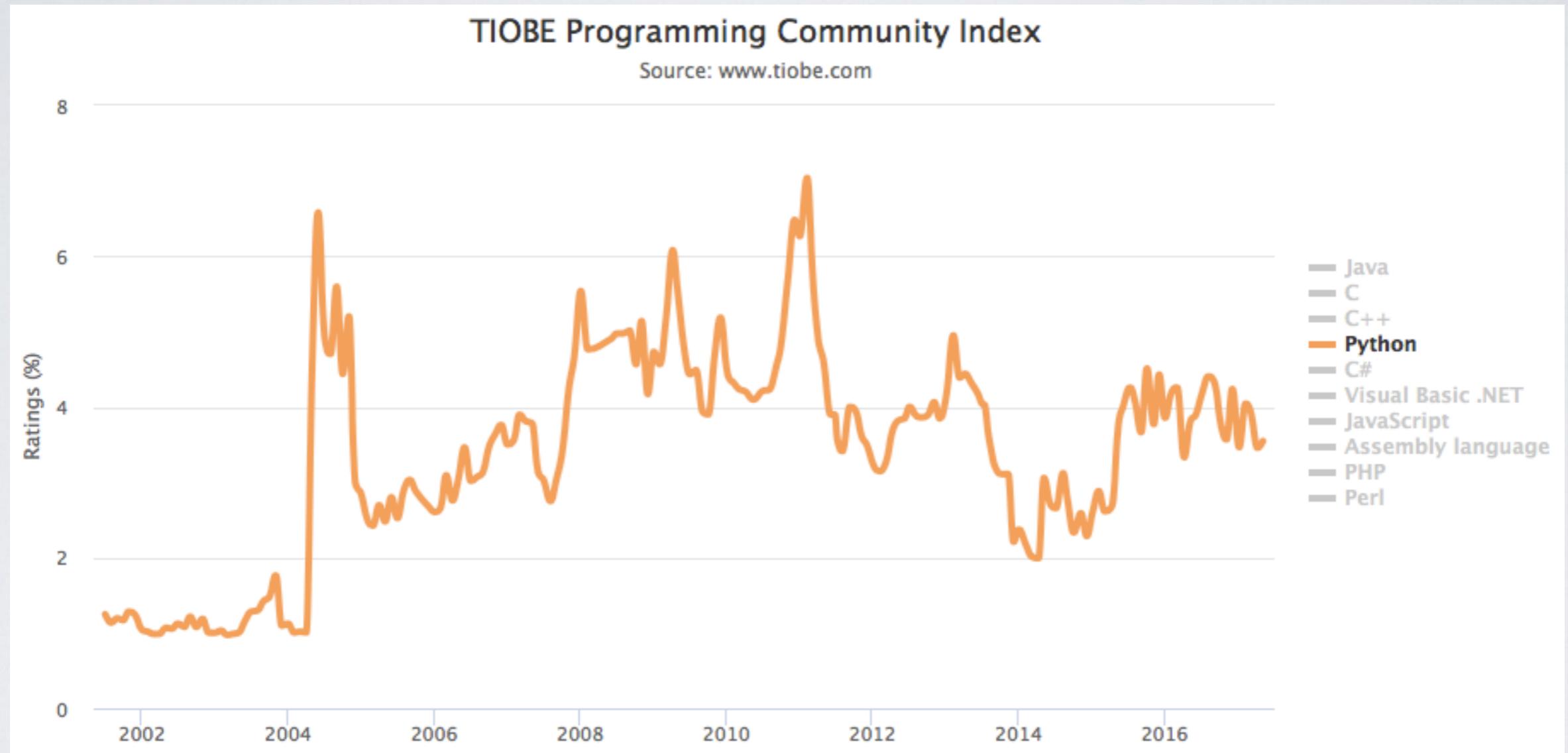


PYTHON

BRIEF HISTORY OF PYTHON

- Rough idea in the late 1980s
- Meant to descend the ABC language
- First line of code in December 1989 by Guido van Rossum
- Python 2.0 in October 2000
- Python 3.0 in December 2008

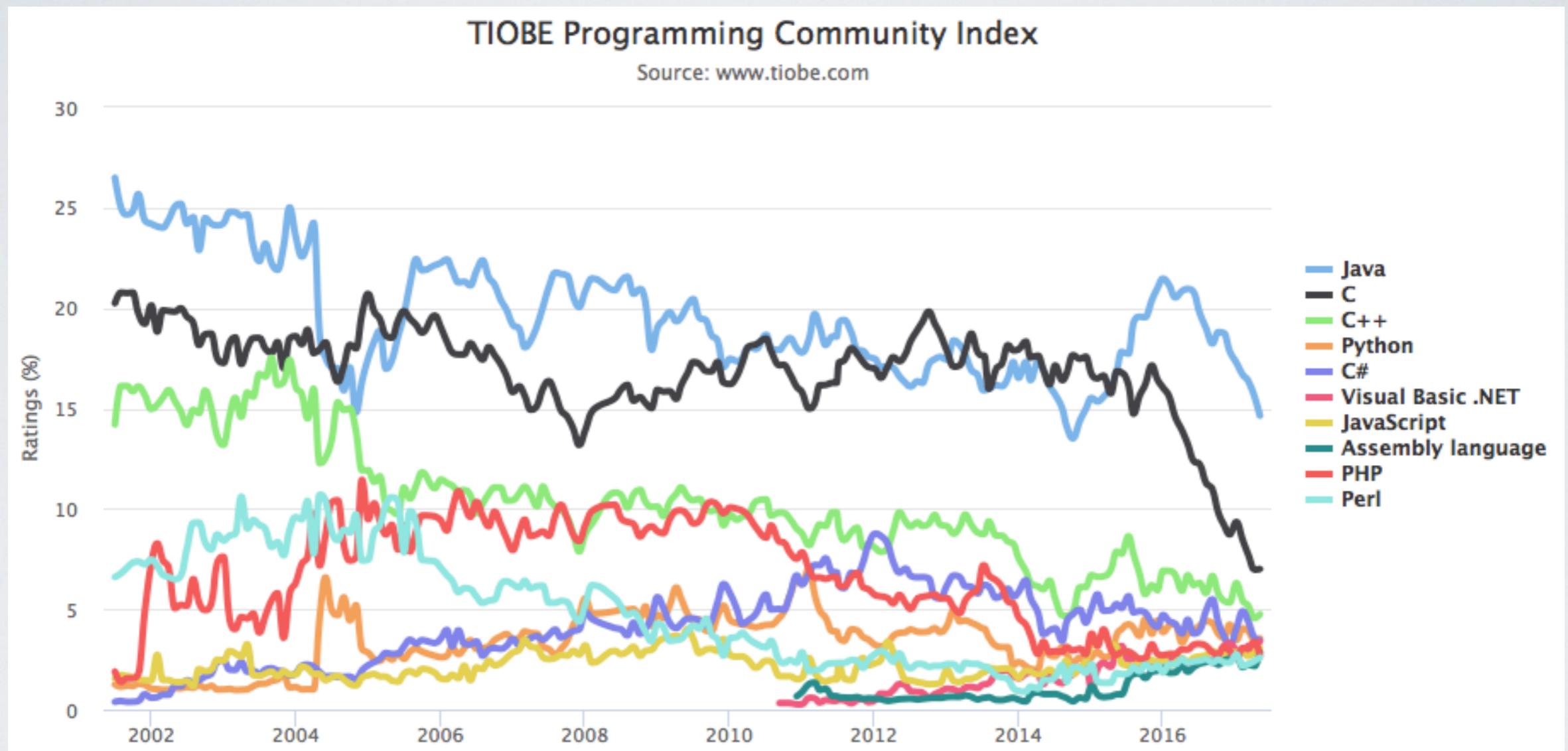
PYTHON'S POPULARITY



“Programming language of the year” in 2007 and 2010.

POPULAR LANGUAGES

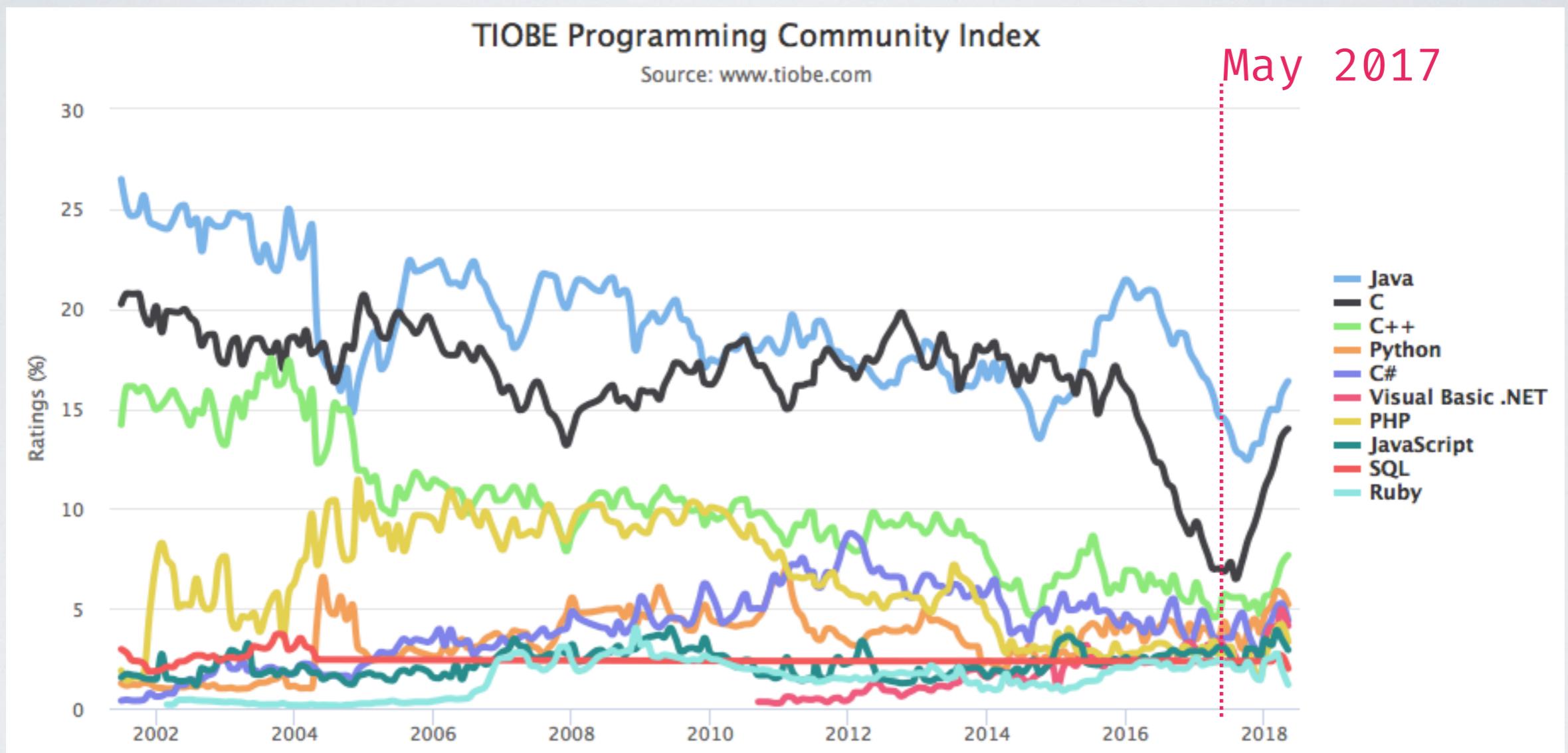
(MAY 2017)



Python is the fourth most popular language and rocks the top 10 since 2003.

POPULAR LANGUAGES

(MAY 2018)



Python is still the fourth most popular language and rocks the top 10 since 2003.

YOUR JOURNEY THROUGH PYTHON?

(JUST A VERY ROUGH GUESS, NOT A MEAN GAME)

Raise your hand and keep it up until you answer a question with “no”.

- Have you ever launched the Python interpreter?
- Wrote for/while-loops or if/else statements?
- ...your own functions?
- ...classes?
- ...list/dict/set comprehensions?
- Do you know what a generator is?
- Have you ever implemented a decorator?
- ...a metaclass?
- ...a C-extension?
- Do you know and can you explain the output of the following line?

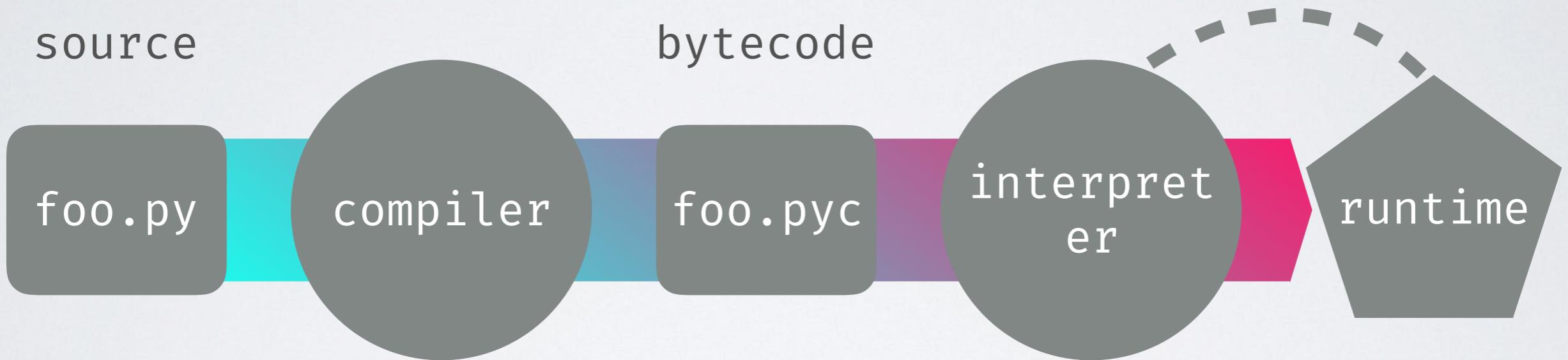
```
print(5 is 7 - 2, 300 is 302 - 2)
```



BASIC PYTHON INTERNALS

to understand the performance issues

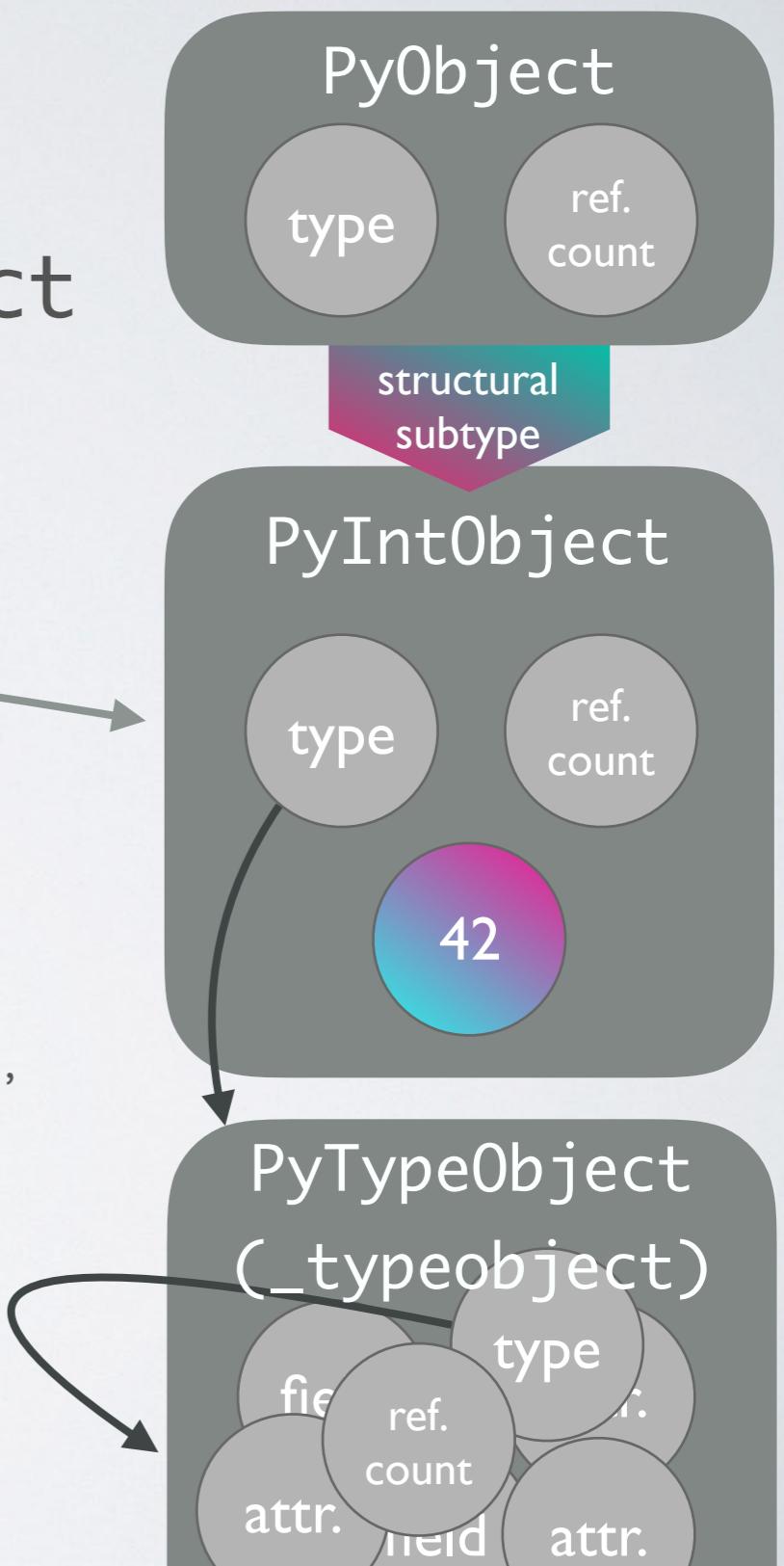
FROM SOURCE TO RUNTIME



DATA IN PYTHON

- Every piece of data is a PyObject

```
>>> dir(42)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__',
'__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__',
'__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length',
'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real',
'to_bytes']
```



THE TYPE OF A PyObject

“An object has a ‘type’ that determines what it represents and what kind of data it contains.

An object’s type is fixed when it is created. Types themselves are represented as objects. The type itself has a type pointer pointing to the object representing the type ‘type’, which contains a pointer to itself!”

– object.h

YOUR BEST FRIEND AND WORST ENEMY: GIL - Global Interpreter Lock

- The GIL prevents parallel execution of (Python) bytecode
- Even though Python has real threads, they never execute code at the same time
- Context switching between threads creates overhead (the user cannot control thread-priority)
- Threads perform pretty bad on CPU bound tasks
- They do a great job speeding up I/O heavy tasks

THREADS AND CPU BOUND TASKS

single thread:

```
N = 100000000

def count(n):
    while n != 0: n -=1

%time count(N)

CPU times: user 5.59 s, sys: 32.5 ms, total: 5.62 s
Wall time: 7.71 s
```

two threads:

```
from threading import Thread

def count_threaded(n):
    t1 = Thread(target=count, args=(N/2,))
    t2 = Thread(target=count, args=(N/2,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()

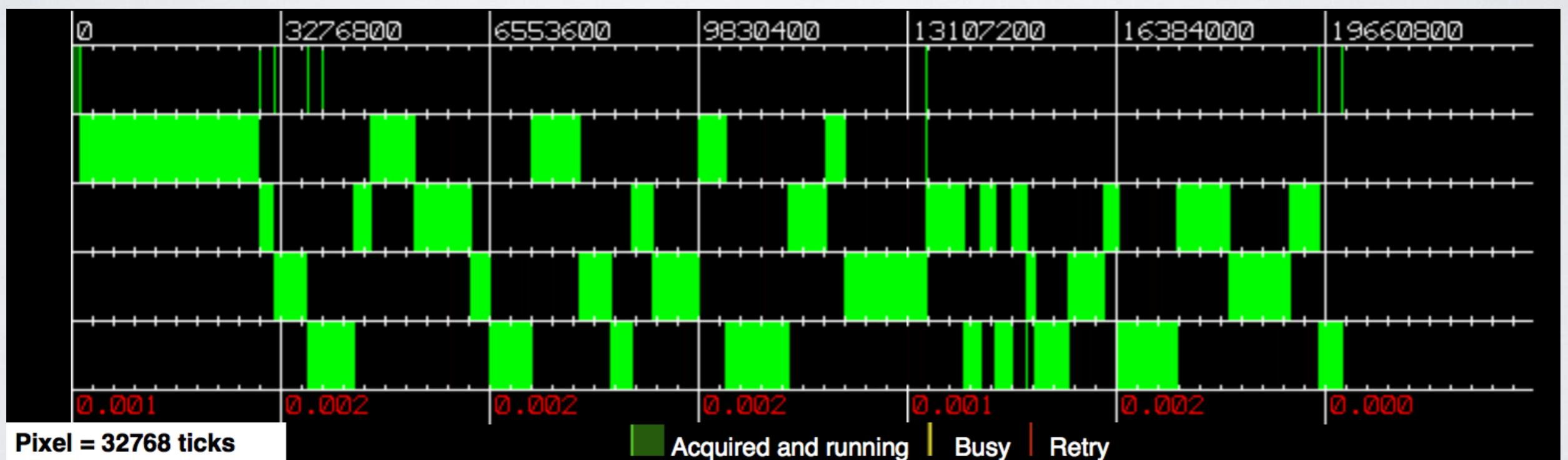
%time count_threaded(N)

CPU times: user 7.18 s, sys: 31 ms, total: 7.21 s
Wall time: 9.01 s
```

This is probably not really what you expected...

THREADS FIGHTING FOR THE GIL

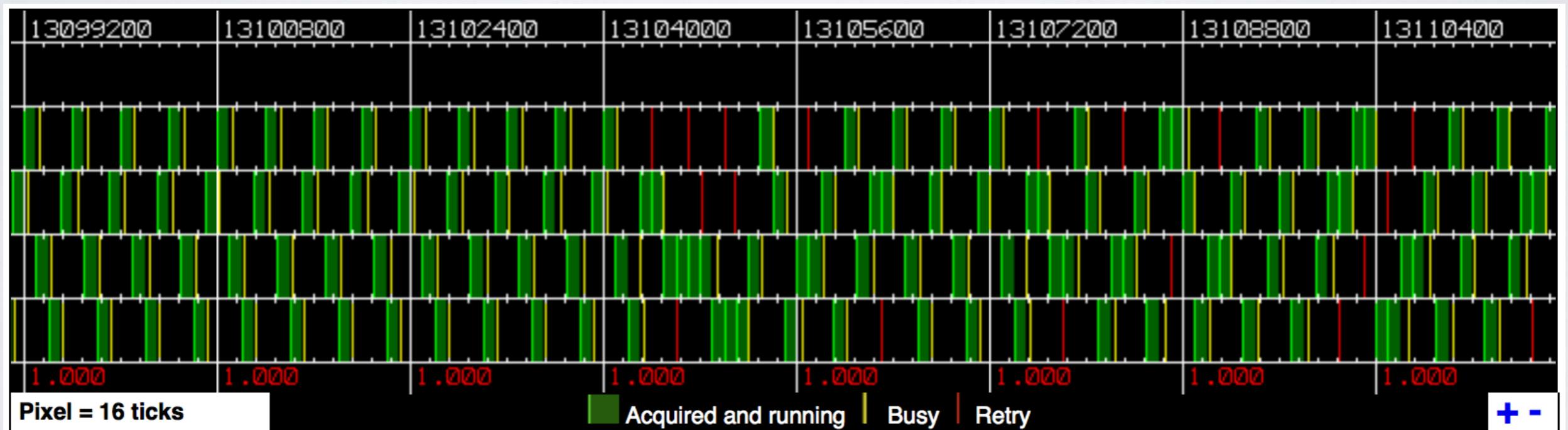
OS X: 4 threads on 1 CPU (Python 2.6)



By David M Beazley: <http://dabeaz.com/GIL/gilvis>

THREADS FIGHTING FOR THE GIL

OS X: 4 threads on 4 CPUs (Python 2.6)



By David M Beazley: <http://dabeaz.com/GIL/gilvis>

OK, huge overhead for every single object,
no real parallel execution of code ...

**How should Python ever compete with all
those super fast C/Fortran libraries?**

C-extensions and interfacing C/Fortran!

Those can release the GIL and do the heavy stuff in the background.

A DUMB SPEED COMPARISON

CALCULATING THE MEAN OF 1000000 RANDOM NUMBERS

pure Python:

```
def mean(numbers):
    return sum(numbers)/len(numbers)

numbers = list(range(1000000))
%timeit mean(numbers)

8.59 ms ± 234 µs per loop
```

NumPy (~13x faster):

```
numbers = np.random.random(1000000)
%timeit np.mean(numbers)

638 µs ± 38.3 µs per loop
```

Numba (~8x faster):

```
@nb.jit
def numba_mean(numbers):
    s = 0
    N = len(numbers)
    for i in range(N):
        s += numbers[i]
    return s/N

numbers = np.random.random(1000000)
%timeit numba_mean(numbers)

1.1 ms ± 6.64 µs per loop
```

Julia (~16x faster):

```
numbers = rand(1000000)
@benchmark mean(numbers)

BenchmarkTools.Trial:
  memory estimate: 16 bytes
  allocs estimate: 1
  -----
  minimum time:      464.824 µs (0.00% GC)
  median time:       524.386 µs (0.00% GC)
  mean time:         544.573 µs (0.00% GC)
  maximum time:     2.095 ms (0.00% GC)
  -----
  samples:           8603
  evals/sample:      1
```

CRAZY LLVM COMPILER OPTIMISATIONS

SUMMING UP NUMBERS FROM 0 TO N=100,000,000

pure Python:

```
def simple_sum(N):
    s = 0
    for i in range(N):
        s += i
    return s

%time simple_sum(N)

CPU times: user 7.13 s, sys: 103 ms, total: 7.23 s
Wall time: 7.43 s

4999999950000000
```

NumPy (~80x faster):

```
np_numbers = np.array(range(N))

%time np.sum(np_numbers)

CPU times: user 84 ms, sys: 2.65 ms, total: 86.6 ms
Wall time: 91.1 ms

4999999950000000
```

Numba (~300000x faster):

```
@nb.jit
def simple_sum(N):
    s = 0
    for i in range(N):
        s += i
    return s

%time numba_sum(N)

CPU times: user 11 µs, sys: 3 µs, total: 14 µs
Wall time: 21.9 µs

4999999950000000
```

Julia (~700000x faster):

```
function simple_sum(N)
    s = 0
    for i ∈ 1:N
        s += i
    end
    return s
end

simple_sum (generic function)

@time simple_sum(N)

0.000002 seconds (5

4999999950000000

Source line: 3
testq %rdi, %rdi
jle L32
leaq -1(%rdi), %rax
leaq -2(%rdi), %rcx
mulq %rcx
shldq $63, %rax, %rdx
leaq -1(%rdx,%rdi,2), %rax

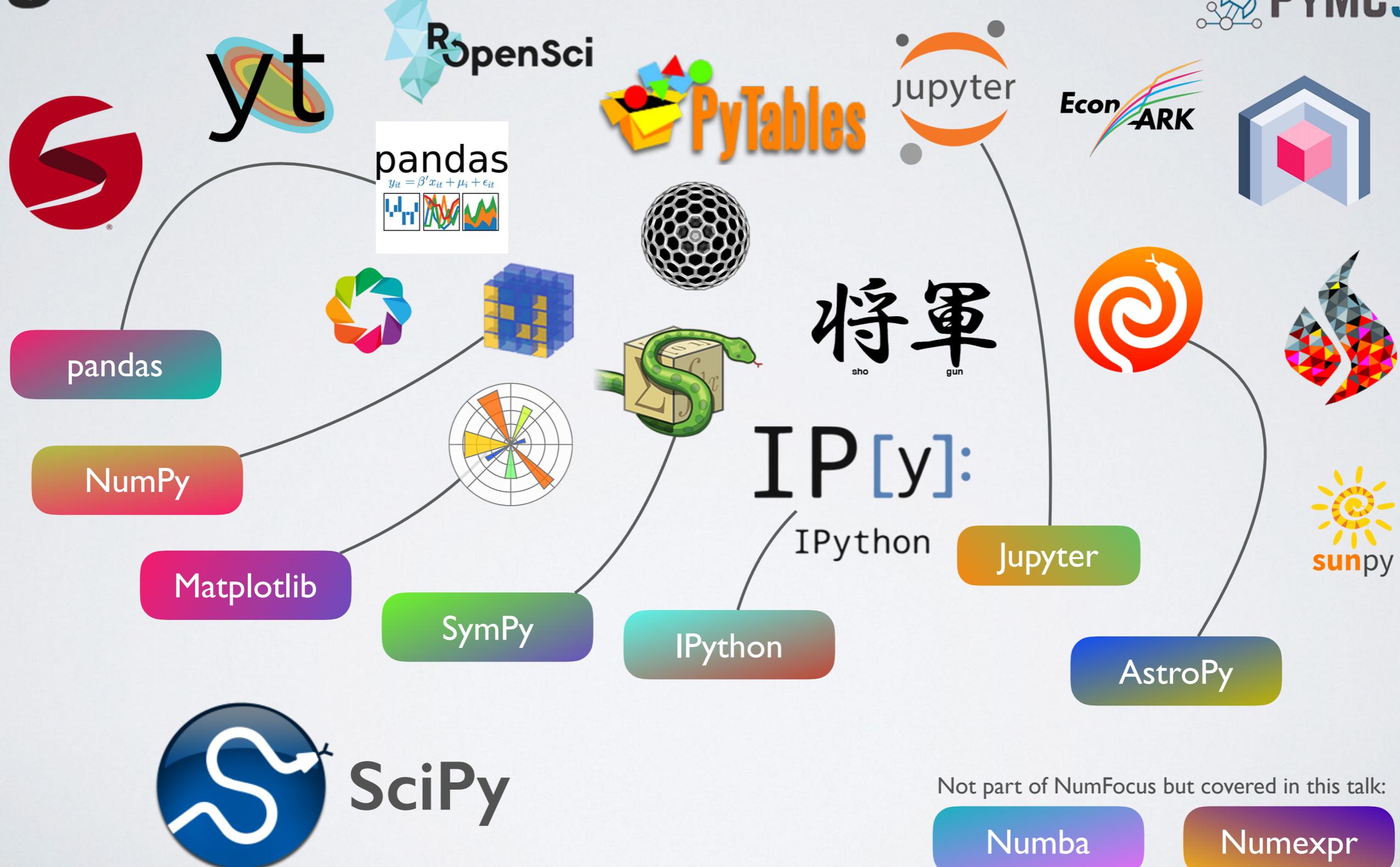
Source line: 6
L32:
popq %rbp
retq
nopw %cs:(%rax,%rax)
```

PYTHON LIBRARIES

for scientific computing



NUMFOCUS
OPEN CODE = BETTER SCIENCE





SCIPY

Scientific Computing Tools for Python

THE SCIPY STACK

- Core packages
 - SciPy Library: numerical algorithms, signal processing, optimisation, statistics etc.
 - NumPy
 - Matplotlib: 2D/3D plotting library
 - pandas: high performance, easy to use data structures
 - SymPy: symbolic mathematics and computer algebra
 - IPython: a rich interactive interface to process data and test ideas
 - Jupyter: notebooks to document and code at the same time
 - nose: testing framework for Python code
- Other packages:
 - Chaco, Mayavi, Cython, Scikits (scikit-learn, scikit-image), h5py, PyTables and much more

<https://www.scipy.org>

SCIPY CORE LIBRARY

- Clustering package (`scipy.cluster`)
- Constants (`scipy.constants`)
- Discrete Fourier transforms (`scipy.fftpack`)
- Integration and ODEs (`scipy.integrate`)
- Interpolation (`scipy.interpolate`)
- Input and output (`scipy.io`)
- Linear algebra (`scipy.linalg`)
- Miscellaneous routines (`scipy.misc`)
- Multi-dimensional image processing (`scipy.ndimage`)
- Orthogonal distance regression (`scipy.odr`)
- Optimization and root finding (`scipy.optimize`)
- Signal processing (`scipy.signal`)
- Sparse matrices (`scipy.sparse`)
- Sparse linear algebra (`scipy.sparse.linalg`)
- Compressed Sparse Graph Routines
(`scipy.sparse.csgraph`)
- Spatial algorithms and data structures (`scipy.spatial`)
- Special functions (`scipy.special`)
- Statistical functions (`scipy.stats`)
- Statistical functions for masked arrays (`scipy.stats.mstats`)

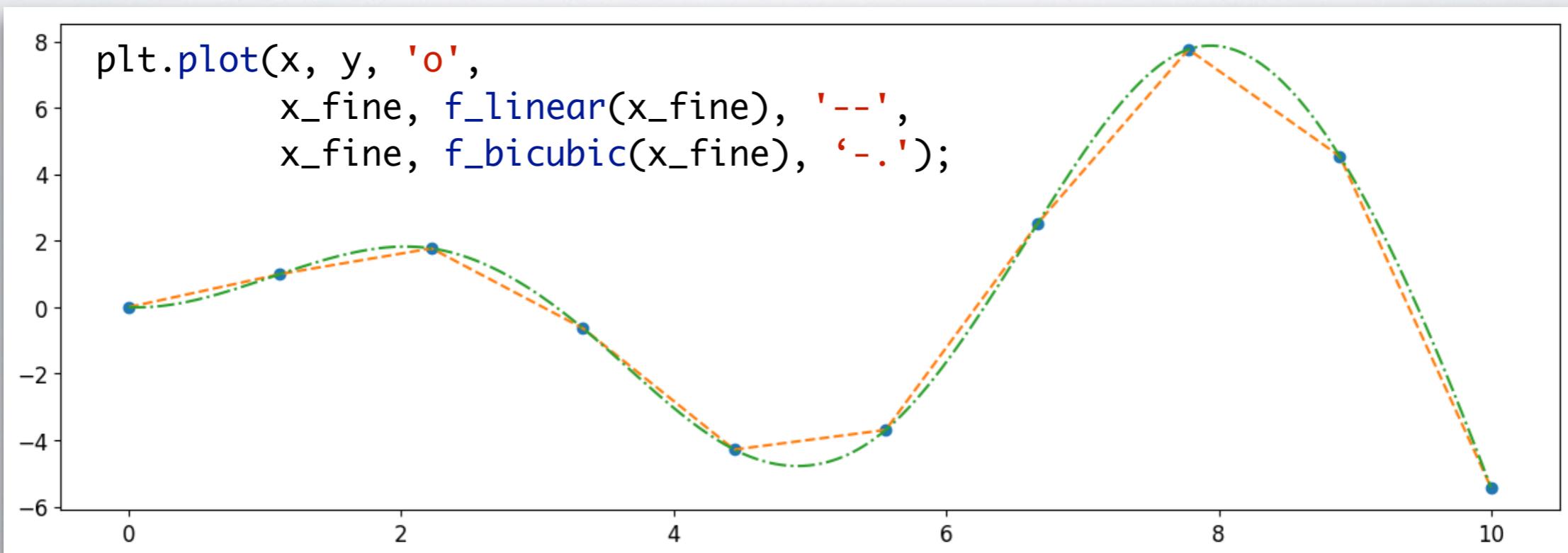
SCIPY INTERPOLATE

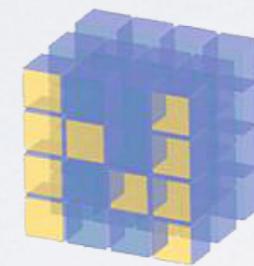
```
from scipy import interpolate

x = np.linspace(0, 10, 10)
y = np.sin(x)

x_fine = np.linspace(0, 10, 500)

f_linear = interpolate.interp1d(x, y, kind='linear')
f_bicubic = interpolate.interp1d(x, y, kind='cubic')
```





NUMPY
Numerical Python

NUMPY

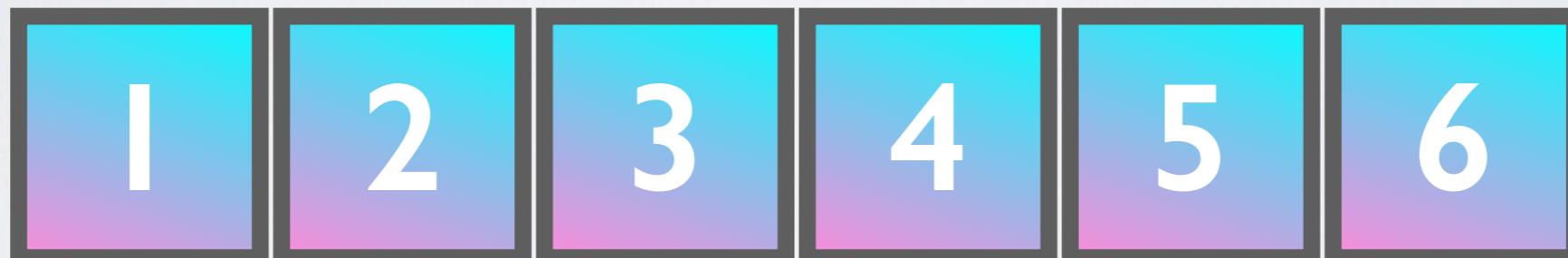
NumPy is the fundamental package for scientific computing with Python.

- gives us a powerful N-dimensional array object: `ndarray`
- broadcasting functions
- tools for integrating C/C++ and Fortran
- linear algebra, Fourier transform and random number capabilities
- most of the scientific libraries build upon NumPy

NUMPY: ndarray

```
a = np.arange(6)  
a  
  
array([0, 1, 2, 3, 4, 5])
```

ndim: 1
shape: (6,)



Continuous array in memory with a fixed type,
no pointer madness!

C/Fortran compatible memory layout,
so they can be passed to those
without any further efforts.

NUMPY: ARRAY OPERATIONS AND ufuncs

```
a * 23
```

```
array([ 0, 23, 46, 69, 92, 115])
```

```
a**a
```

```
array([ 1, 1, 4, 27, 256, 3125])
```

easy and intuitive
element-wise
operations

a ufunc, which can operate both on scalars and arrays (element-wise)

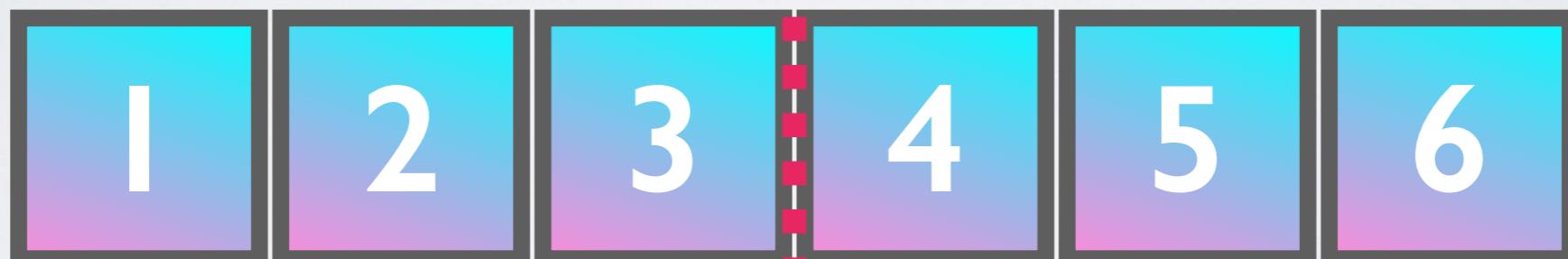
```
np.exp(a)
```

```
array([ 1. , 2.71828183, 7.3890561 , 20.08553692,
       54.59815003, 148.4131591 ])
```

RESHAPING ARRAYS

```
a = np.arange(6)  
a  
  
array([0, 1, 2, 3, 4, 5])
```

ndim: 1
shape: (6,)



a[0]

a[1]

```
a.reshape(2, 3)  
  
array([[0, 1, 2],  
       [3, 4, 5]])
```

No rearrangement of the elements
but setting the iterator limits internally!

RESHAPING ARRAYS IS CHEAP

```
a = np.arange(10000000)  
%timeit b = a.reshape(100, 5000, 20)  
563 ns ± 8.18 ns per loop (mean ± std.
```

Don't worry, we will discover NumPy in the hands-on workshop!



The logo for matplotlib consists of the word "matplotlib" in a large, bold, blue sans-serif font. The letter "o" is replaced by a circular pie chart with several colored segments (orange, yellow, green, blue) and radial grid lines.

MATPLOTLIB

A Python plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments.

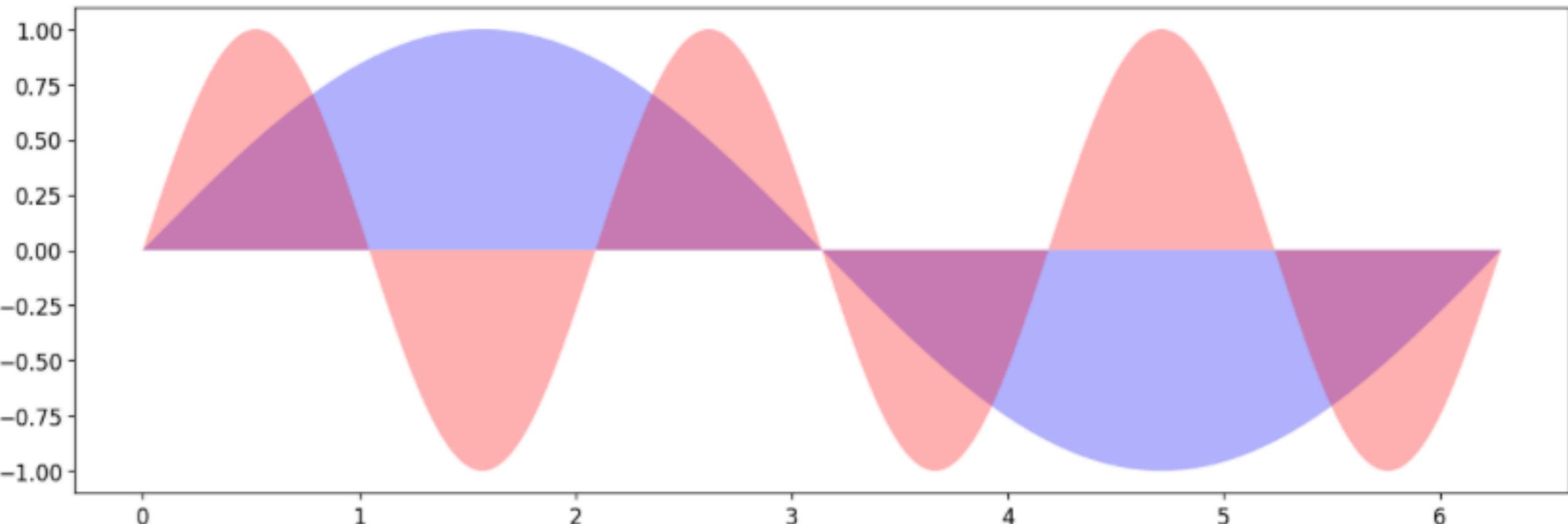
- Integrates well with IPython and Jupyter
- Plots, histograms, power spectra, bar charts, error chars, scatterplots, etc. with an easy to use API
- Full control of line styles, font properties, axes properties etc.
- The easiest way to get started is browsing its wonderful gallery full of thumbnails and copy&paste examples:
<http://matplotlib.org/gallery.html>

MATPLOTLIB EXAMPLE

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi, 500)
y1 = np.sin(x)
y2 = np.sin(3 * x)

fig, ax = plt.subplots()
ax.fill(x, y1, 'b', x, y2, 'r', alpha=0.3)
plt.show()
```

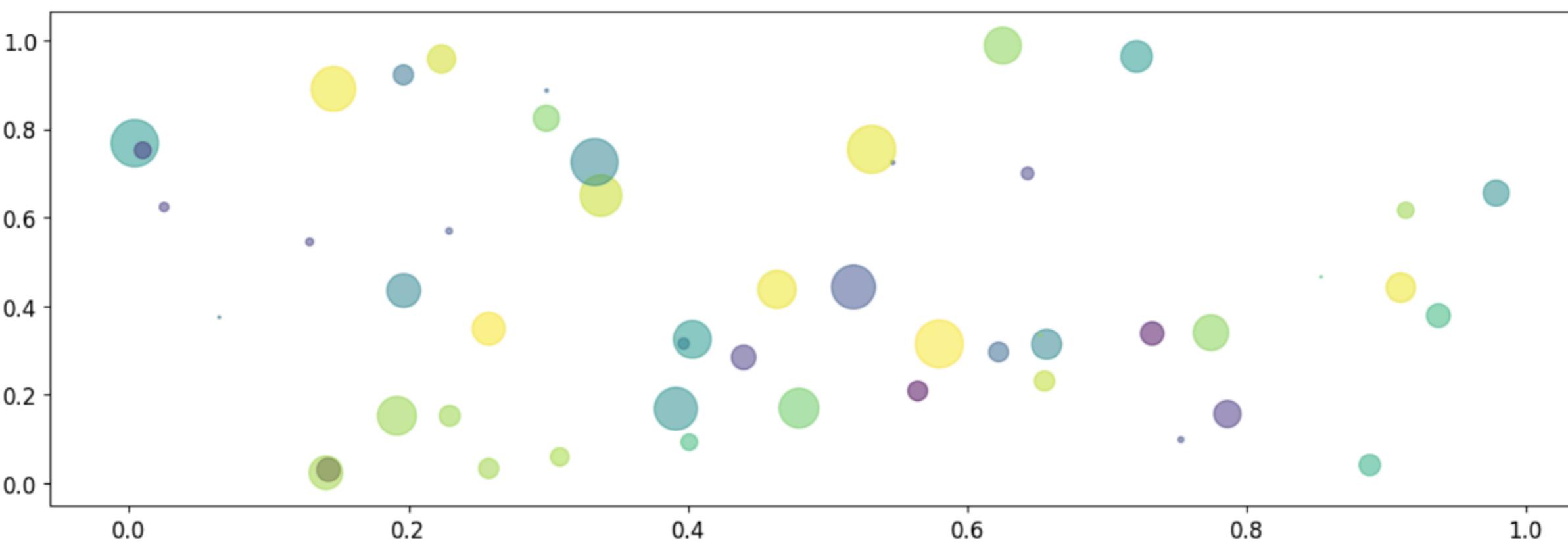


MATPLOTLIB EXAMPLE

```
import numpy as np
import matplotlib.pyplot as plt

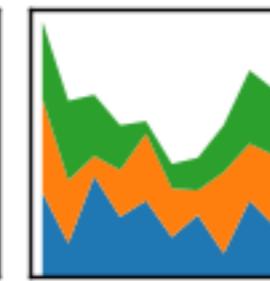
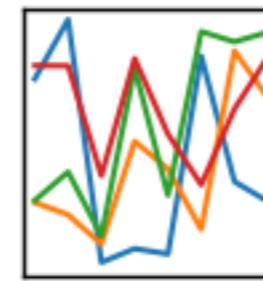
N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2

plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



PANDAS

A Python Data Analysis Library inspired by data frames in R:

- gives us a powerful data structure: `DataFrame`
- database-like handling of data
- integrates well with NumPy
- wraps the Matplotlib API (which can also cause troubles ;)
- has a huge number of I/O related functions to parse data:
`CSV`, `HDF5`, `SQL`, `Feather`, `JSON`, `HTML`, `Excel`, and more...

THE DataFrame

A table-like structure, where you can access elements by row and column.

```
hits = pd.read_hdf("event_file.h5", "events/23")
hits.head(3)
```

	channel_id	dom_id	event_id	id	pmt_id	time	tot	triggered
0	25	808430036		0	0	30652287	21	0
1	18	808430036		0	0	30656200	16	0
2	15	808430449		0	0	30648451	26	0

THE DataFrame

Lots of functions to allow filtering, manipulating and aggregating the data to fit your needs.

```
▼ active_doms = hits.pivot_table(index='event_id',
                                   values='dom_id',
                                   aggfunc=lambda x: set(x))
```

Don't worry, we will discover Pandas in the hands-on workshop!

sponsored by
CONTINUUM®
ANALYTICS

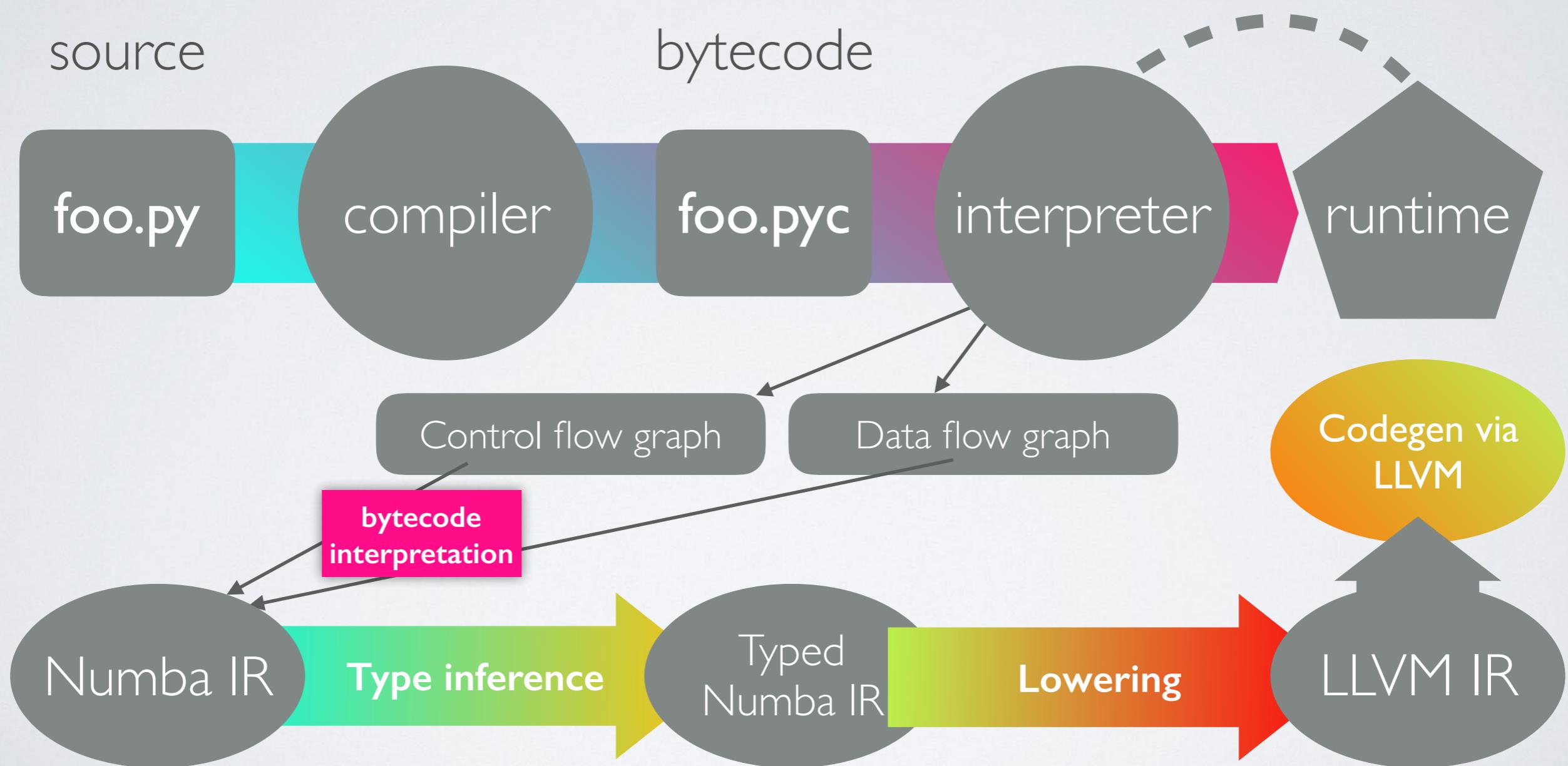
 NUMBA
JIT (LLVM) compiler for Python

NUMBA

Numba is a compiler for Python array and numerical functions that gives you the power to speed up code written directly in Python.

- uses LLVM to boil down pure Python code to JIT optimised machine code
- only accelerates selected functions decorated by yourself
- native code generation for CPU (default) and GPU
- integration with the Python scientific software stack (thanks to NumPy)
- runs side by side with regular Python code or third-party C extensions and libraries
- great CUDA support
- N-core scalability by releasing the GIL (beware: no protection from race conditions!)
- create NumPy ufuncs with the `@[gu]vectorize` decorator(s)

FROM SOURCE TO RUNTIME



NUMBA JIT-EXAMPLE

```
numbers = np.arange(1000000).reshape(2500, 400)
```

```
def sum2d(arr):  
    M, N = arr.shape  
    result = 0.0  
    for i in range(M):  
        for j in range(N):  
            result += arr[i,j]  
    return result
```

```
@nb.jit  
def sum2d_jit(arr):  
    M, N = arr.shape  
    result = 0.0  
    for i in range(M):  
        for j in range(N):  
            result += arr[i,j]  
    return result
```

289 ms ± 3.02 ms per loop

2.13 ms ± 42.6 µs per loop

~135x faster, with a single line of code

NUMBA VECTORIZE-EXAMPLE

```
a = np.arange(1000000, dtype='f8')
b = np.arange(1000000, dtype='f8') + 23
```

NumPy:

```
np.abs(a - b) / (np.abs(a) + np.abs(b))           23 ms ± 845 µs per loop
```

Numba @vectorize:

```
@nb.vectorize
def nb_rel_diff(a, b):
    return abs(a - b) / (abs(a) + abs(b))
```

```
rel_diff(a, b)           3.56 ms ± 43.2 µs per loop
```

~6x faster

NUMEXPR

initially written by David Cooke

Routines for the fast evaluation of array expressions element-wise
by using a vector-based virtual machine.

NUMEXPR USAGE EXAMPLE

```
import numpy as np
import numexpr as ne

a = np.arange(5)
b = np.linspace(0, 2, 5)

ne.evaluate("a**2 + 3*b")

array([ 0. ,  2.5,  7. , 13.5, 22. ])
```

NUMEXPR SPEED-UP

```
a = np.random.random(1000000)
```

NumPy:

```
2 * a**3 - 4 * a**5 + 6 * np.log(a)
```

82.4 ms ± 1.88 ms per loop

Numexpr with 4 threads:

```
ne.set_num_threads(4)
```

```
ne.evaluate("2 * a**3 - 4 * a**5 + 6 * log(a)")
```

7.85 ms ± 103 µs per loop

~10x faster

NUMEXPR – SUPPORTED OPERATORS

- Logical operators: `&`, `|`, `~`
- Comparison operators:
`<`, `<=`, `==`, `!=`, `>=`, `>`
- Unary arithmetic operators: `-`
- Binary arithmetic operators:
`+`, `-`, `*`, `/`, `**`, `%`, `<<`, `>>`

NUMEXPR – SUPPORTED FUNCTIONS

- `where(bool, number1, number2)`: number -- number1 if the bool condition is true, number2 otherwise.
- `{sin,cos,tan}(float|complex)`: float|complex -- trigonometric sine, cosine or tangent.
- `{arcsin,arccos,arctan}(float|complex)`: float|complex -- trigonometric inverse sine, cosine or tangent.
- `arctan2(float1, float2)`: float -- trigonometric inverse tangent of float1/float2.
- `{sinh,cosh,tanh}(float|complex)`: float|complex -- hyperbolic sine, cosine or tangent.
- `{arsinh,arccosh,arctanh}(float|complex)`: float|complex -- hyperbolic inverse sine, cosine or tangent.
- `{log,log10,log1p}(float|complex)`: float|complex -- natural, base-10 and log(1+x) logarithms.
- `{exp,expm1}(float|complex)`: float|complex -- exponential and exponential minus one.
- `sqrt(float|complex)`: float|complex -- square root.
- `abs(float|complex)`: float|complex -- absolute value.
- `conj(complex)`: complex -- conjugate value.
- `{real,imag}(complex)`: float -- real or imaginary part of complex.
- `complex(float, float)`: complex -- complex from real and imaginary parts.
- `contains(str, str)`: bool -- returns True for every string in `op1` that contains `op2`.
- `sum(number, axis=None)`: Sum of array elements over a given axis. Negative axis are not supported.
- `prod(number, axis=None)`: Product of array elements over a given axis. Negative axis are not supported.



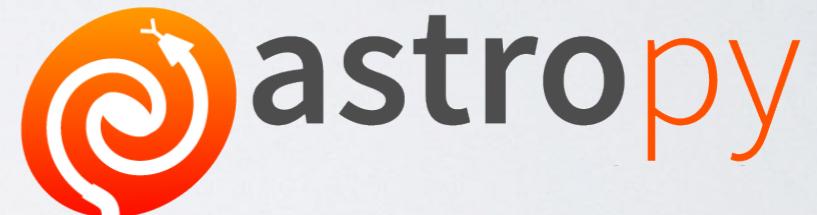
A Community Python Library for Astronomy

THE HISTORY OF ASTROPY

(standard situation back in 2011)

- Example Problem: convert from EQ J2000 RA/Dec to Galactic coordinates
- Solution in Python
 - pyast
 - Astrolib
 - Astrophysics
 - PyEphem
 - PyAstro
 - Kapteyn
 - ???

huge discussion
started in June 2011
series of votes



First public version (v0.2) presented and described in the following paper:
<http://adsabs.harvard.edu/abs/2013A&A...558A..33A>

ASTROPY CORE PACKAGE

A community-driven package intended to contain much of the core functionality and some common tools needed for performing astronomy and astrophysics with Python.

- **Data structures and transformations**

- constants, units and quantities, N-dimensional datasets, data tables, times and dates, astronomical coordinate system, models and fitting, analytic functions

- **Files and I/O**

- unified read/write interface
- FITS, ASCII tables, VOTable (XML), Virtual Observatory access, HDF5, YAML, ...

- **Astronomy computations and utilities**

- cosmological calculations, convolution and filtering, data visualisations, astrostatistics tools

ASTROPY AFFILIATED PACKAGES

- Tons of astronomy related packages
- which are not part of the core package,
- but has requested to be included as part of the Astropy project's community

ASTROPY EXAMPLE

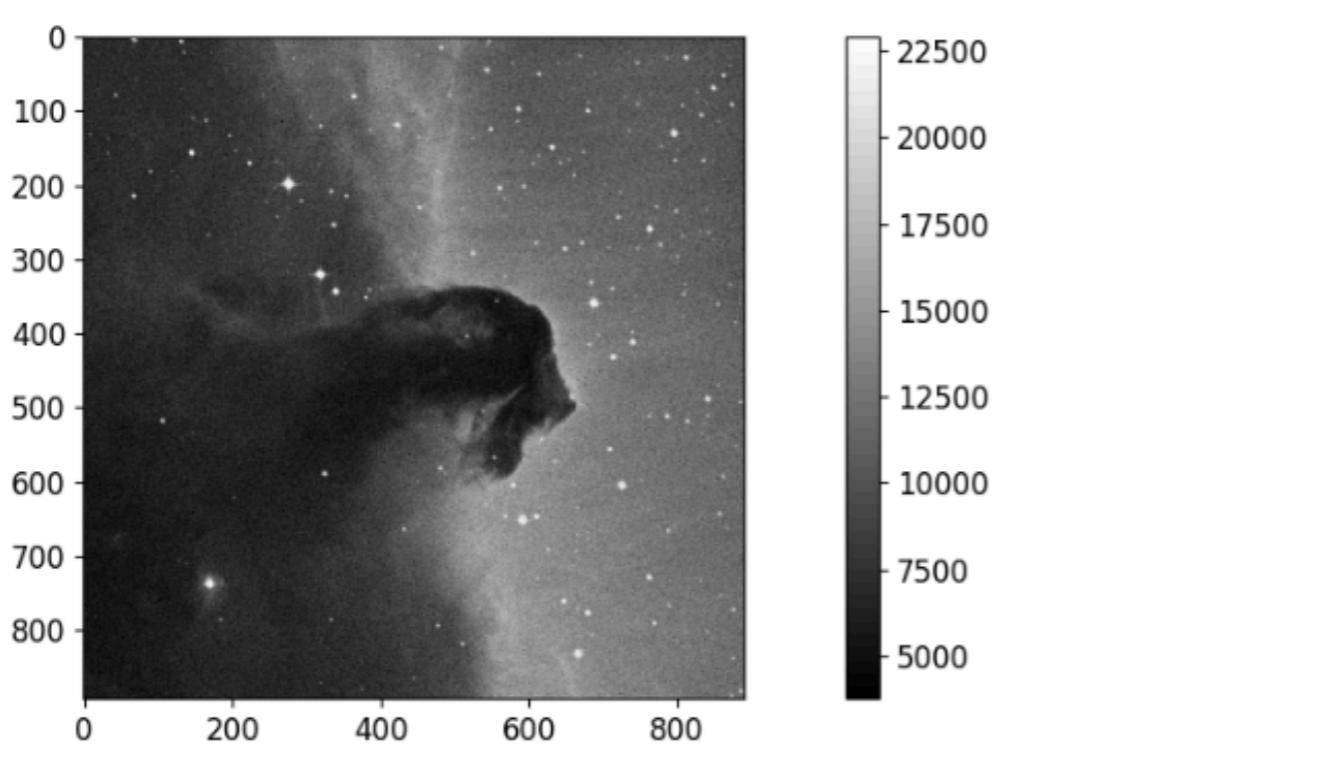
```
from astropy.utils.data import download_file
from astropy.io import fits

image_file = download_file('http://data.astropy.org/tutorials/FITS-images/HorseHead.fits')
Downloading http://data.astropy.org/tutorials/FITS-images/HorseHead.fits [Done]

fits.info(image_file)
Filename: /Users/tamasgal/.astropy/cache/download/py3/2c9202ae878ecfcb60878ceb63837f5f
No.    Name        Type      Cards   Dimensions   Format
 0  PRIMARY    PrimaryHDU     161   (891, 893)   int16
 1  er.mask    TableHDU      25   1600R x 4C   [F6.2, F6.2, F6.2, F6.2]

image_data = fits.getdata(image_file, ext=0)

plt.figure()
plt.imshow(image_data, cmap='gray')
plt.colorbar();
```



← downloading via HTTP

← checking some FITS meta

← extracting image data

← plotting via Matplotlib

ASTROPY EXAMPLE

```
from astropy.coordinates import SkyCoord
import astropy.units as u

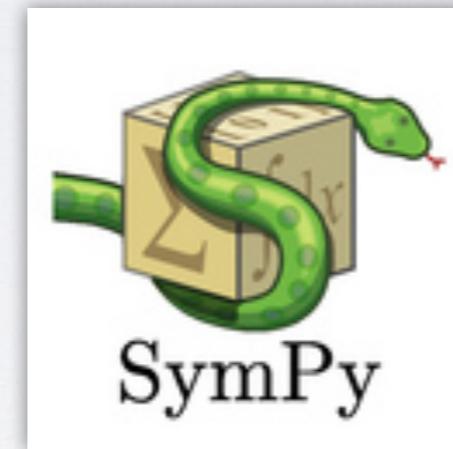
m13 = SkyCoord.from_name('m13')
m13

<SkyCoord (ICRS): (ra, dec) in deg
  ( 250.4234583, 36.4613056)>

m13.ra, m13.ra.to(u.hourangle)

(<Longitude 250.4234583 deg>, <Longitude 16.69489722 hourangle>)
```

Don't worry, we will discover AstroPy in the hands-on workshop!



A Python library for symbolic mathematics.

SIMPY

- It aims to become a full-featured computer algebra system (CAS)
- while keeping the code as simple as possible
- in order to be comprehensible and easily extensible.
- SymPy is written entirely in Python.
- It only depends on mpmath, a pure Python library for arbitrary floating point arithmetic

SIMPY

- solving equations
- solving differential equations
- simplifications: trigonometry, polynomials
- substitutions
- factorisation, partial fraction decomposition
- limits, differentiation, integration, Taylor series
- combinatorics, statistics, ...
- much much more

SIMPY EXAMPLE

Base Python

```
In [1]: import math  
  
In [2]: math.sqrt(8)  
Out[2]: 2.8284271247461903  
  
In [3]: math.sqrt(8)**2  
Out[3]: 8.000000000000002
```

SymPy

```
In [4]: import sympy  
  
In [5]: sympy.sqrt(8)  
Out[5]: 2*sqrt(2)  
  
In [6]: sympy.sqrt(8)**2  
Out[6]: 8
```

SIMPY EXAMPLE

```
In [15]: x, y = sympy.symbols('x y')
```

```
In [16]: expr = x + 2*y
```

```
In [17]: expr
```

```
Out[17]: x + 2*y
```

```
In [18]: expr + 1
```

```
Out[18]: x + 2*y + 1
```

```
In [19]: expr * x
```

```
Out[19]: x*(x + 2*y)
```

```
In [20]: sympy.expand(expr * x)
```

```
Out[20]: x**2 + 2*x*y
```

SIMPY EXAMPLE

```
In [1]: import sympy
```

```
In [2]: from sympy import init_printing, integrate, diff, exp, cos, sin, oo
```

```
In [3]: init_printing(use_unicode=True)
```

```
In [4]: x = sympy.symbols('x')
```

```
In [5]: diff(sin(x)*exp(x), x)
```

```
Out[5]:
```

$$e^x \cdot \sin(x) + e^x \cdot \cos(x)$$

```
In [6]: integrate(exp(x)*sin(x) + exp(x)*cos(x), x)
```

```
Out[6]:
```

$$\frac{e^x \cdot \sin(x)}{2}$$

```
In [7]: integrate(sin(x**2), (x, -oo, oo))
```

```
Out[7]:
```

$$\frac{\sqrt{2} \cdot \sqrt{\pi}}{2}$$



I P [y]:

IPython

IPYTHON

- The interactive Python shell!
- Object introspection
- Input history, persistent across sessions
- Extensible tab completion
- “Magic” commands (basically macros)
- Easily embeddable in other Python programs and GUIs
- Integrated access to the pdb debugger and the Python profiler
- Syntax highlighting
- real multi-line editing
- Provides a kernel for Jupyter
- ...and such more!



Project Jupyter is an open source project that offers a set of tools for interactive and exploratory computing.

JUPYTER

- Born out of the IPython project in 2014
- Jupyter provides a console and a notebook server for all kinds of languages
(the name Jupyter comes from **Julia**, **Python** and **R**)
- An easy way to explore and prototype
- Notebooks support Markdown and LaTeX-like input and rendering
 - Allows sharing code and analysis results
 - Extensible (slideshow plugins, JupyterLab, VIM binding, ...)

JUPYTER CONSOLE

A terminal frontend for kernels which use the Jupyter protocol.

The screenshot displays three terminal windows illustrating the Jupyter console:

- Top Window:** Shows the command `jupyter kernelspec list` being run, displaying available kernels: haskell, julia-0.5, julia-0.6, km3net, and python3.
- Middle Left Window:** Shows the command `jupyter console` being run, resulting in the Python 3.6 kernel starting. It displays the Python version and license information.
- Middle Right Window:** Shows the command `jupyter console --kernel=julia-0.5` being run, resulting in the Julia 0.5 kernel starting. It displays the Julia version and a welcome message.

JUPYTER NOTEBOOK

- A Web-based application suitable for capturing the whole computation process:
 - developing
 - documenting
 - and executing code
 - as well as communicating the results.
- Two main components:
 - a web application: a browser-based tool for interactive authoring of documents which combine explanatory text, mathematics, computations and their rich media output.
 - notebook documents: a representation of all content visible in the web application, including inputs and outputs of the computations, explanatory text, mathematics, images, and rich media representations of objects.

JUPYTER NOTEBOOK

The screenshot shows a Jupyter Notebook interface running on a Mac OS X system. The top navigation bar includes standard OS X icons for window control and a tab for 'localhost:8888/notebooks/Research/DU-2'. The main window has two side-by-side panes.

Left Pane (Code Input): This pane contains two code cells. The first cell's input is:

```
In [51]: df[df["hit_time"] < 40]["tot"].hist(bins=255)
```

Its output is:

```
Out[51]: <matplotlib.axes._subplots.AxesSubplot at 0x11d73d6a0>
```

Below this is another cell's input:

```
In [72]: df[df['time_length'] > 100]
```

Its output is:

channel_id	dom_id	time	tot	triggered	event_id	hit_time	time_length
0	23	808953148	241952	64	0	0	0
1	25	808953148	241953	30	0	0	1
2	27	808953148	241957	34	0	0	5
3	30	808953148	241978	25	0	0	26
4	0	808953148	241955	37	0	0	3
5	0	808953148	242041	37	0	0	89
6	1	808953148	242041	46	0	0	89

Right Pane (Rendered Output): This pane shows a 3D surface plot of event data over time. The x-axis represents time from December 2015 to February 2016, with labels rotated for readability. The y-axis represents floor levels from 0 to 18. The z-axis represents signal intensity. A color bar at the bottom indicates intensity from -1.6988e-07 to 19.9997. Above the plot, there is Python code:

```
(useOffset=False, axis='y')

)
formatter(xfmt)
(bbox_to_anchor=(1.005, 1), loc=2, borderaxespad=0.);

_lines():

')
```

Annotations in red text are overlaid on the right side of the right pane:

- "cells for code/markup input" points to the code cell area.
- "rendered output for text/images/tables etc." points to the 3D plot area.

JUPYTERLAB

- The next level of interacting with notebooks
- Extensible: terminal, text editor, image viewer, etc.
- Supports editing multiple notebooks at once
- Drag and drop support to arrange panes

JUPYTERLAB

Screenshot of JupyterLab interface showing multiple notebooks and a file browser.

File Browser:

- Path: Research > Playground
- Files:
 - Julia (a month ago)
 - scipy_2015_sklearn_t... (6 months ago)
 - System Monitoring (a year ago)
 - 1.2_Tools_numpy_p... (a year ago)
 - 3D Line Fit.ipynb (a year ago)
 - An introduction to Ma... (6 months ago)
 - Aussie Rules Football.... (a year ago)
 - Bad Colour Maps.ipynb (a year ago)
 - Coin Flip - Waiting for ... (a year ago)
 - Configparser.ipynb (a year ago)
 - Cython.ipynb (a year ago)
 - Distances of points in ... (a year ago)
 - Distributions.ipynb (a year ago)
 - Draw Picture Pixel by ... (a year ago)
 - DU Plot.ipynb (10 months ago)
 - Fun with the Pipeline.i... (a year ago)
 - HDF5 Basics.ipynb (a month ago)
 - HDF5 Formats.ipynb (a year ago)
 - HDF5 Performance.ip... (8 months ago)
 - Hit vs CHit Performan... (a year ago)
 - HitSeries.ipynb (a year ago)
 - Interact.ipynb (a year ago)
 - Känguruh.ipynb (a year ago)
 - Leap Seconds.ipynb (a year ago)
 - Linear Equations Syst... (a year ago)
 - Machine Learning.ipynb (6 months ago)
 - Matplotlib Subplots.ip... (a year ago)
 - Mensch Ärgere Dich ... (4 months ago)
 - Neural Networks.ipynb (7 months ago)
 - Numba.ipynb (4 months ago)
 - Numexpr.ipynb (8 months ago)
 - Numpy - Named Tipl... (a year ago)
 - Pandas Appendix.ipynb (a year ago)

Notebooks:

- DU2-DOM9 Lo.ipynb:**
 - In [21]:

```
fig, ax = plt.subplots()
du2dom9 = db.doms.via_omkey((2, 9), "D_ARCA003")
du2dom3 = db.doms.via_omkey((2, 3), "D_ARCA003")
temp[temp.SOURCE_NAME == du2dom9.clb_upi].plot('DATETIME',
'VALUE', ax=ax, label=du2dom9)
temp[temp.SOURCE_NAME == du2dom3.clb_upi].plot('DATETIME',
'VALUE', ax=ax, label=du2dom3)
plt.xlabel("Time on 2016-11-04 [UTC]")
plt.ylabel("Temperature [°C]")
```

 - Out[21]:
- K40.ipynb:**
 - In [11]:

```
times, channel_ids = [np.array(i) for i in
zip(*foo)]
print(len(times))
#print(channel_ids)

diffs = np.diff(times)
#print(diffs)
idx = np.where(np.diff(times) < 20)[0]
#print(idx)
break
narf(times)
#print(channel_ids[idx])
```

 - Out[11]:

channel_id	dom_id	id	pmt_id	time	tot	triggered	event_id	
0	28	808430449	0	0	20292053	28	False	0
1	12	808430571	1	0	20290049	26	False	0
2	8	808447091	2	0	20288472	27	False	0

 - In [104]:

```
tmax = 20
def mongincidence(times, tdcs):
    coincidences = []
    cur_t = 0
    las_t = 0
    for t_idx, t in enumerate(times):
        cur_t = t
        diff = cur_t - las_t
        if diff < tmax and t_idx > 0:
            coincidences.append(((tdcs[t_idx - 1],
tdcs[t_idx]), diff))
            las_t = cur_t
    return coincidences
```

 - In [105]:

```
mongincidence((1, 20, 21), (10, 11, 12))
```

 - Out[105]:

(t1, t2)	value
(10, 11)	19
(11, 12)	1
- IPython: Users.ipynb:**
 - In [5]:

```
...: del shorterr
...:
```

 - In [6]:

```
import numpy as np
np.add
```

JUPYTERHUB

- JupyterHub creates a multi-user Hub which spawns, manages, and proxies multiple instances of the single-user Jupyter notebook server
- A nice environment for teaching
- Great tool for collaborations
(ask your IT admin ;)

SOME OTHER USEFUL LIBRARIES

SEABORN

statistical data visualisation
uses matplotlib as backend

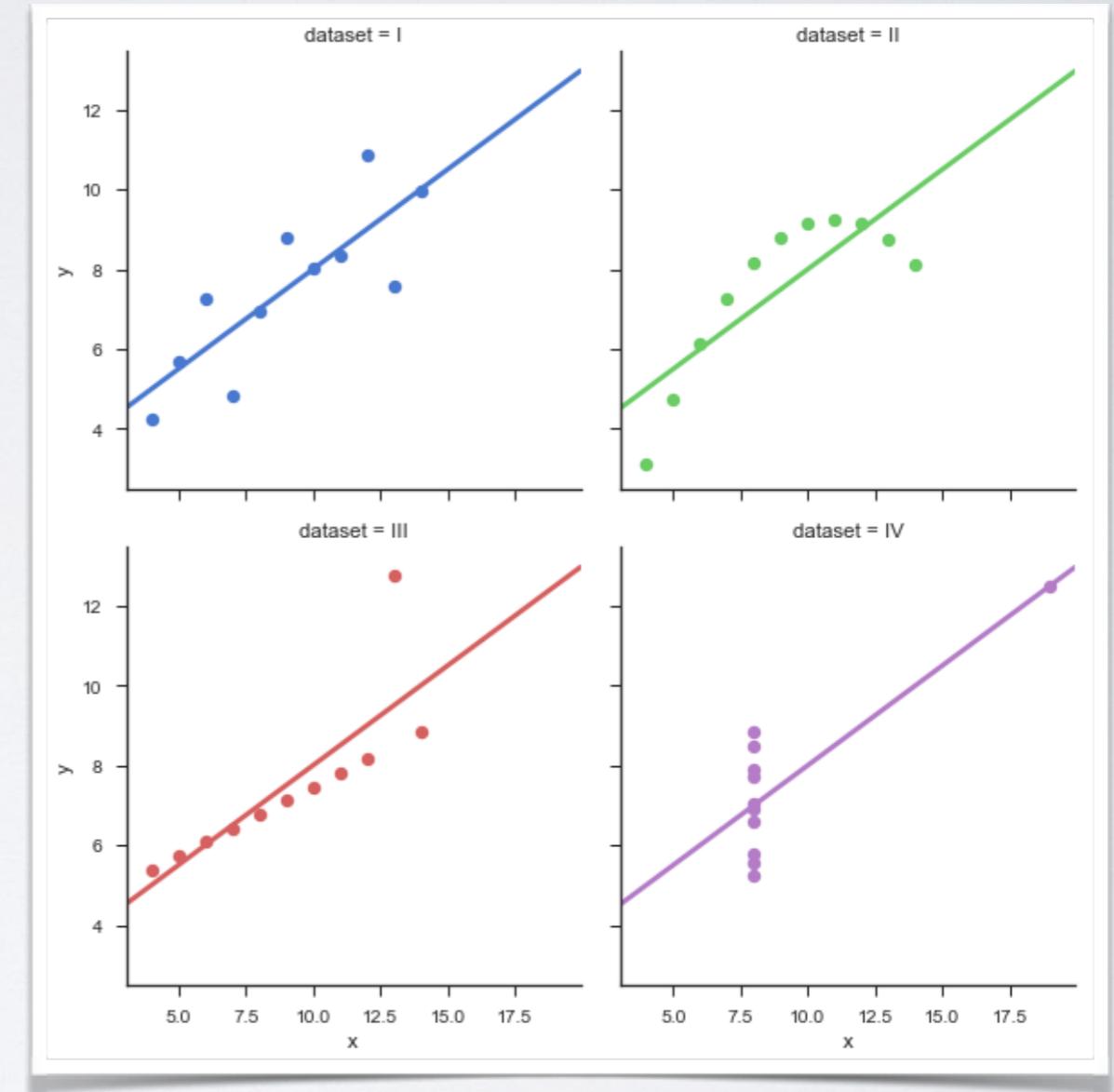
<https://seaborn.pydata.org>

CONVENIENT WRAPPER FUNCTIONS FOR MATPLOTLIB

```
import seaborn as sns
sns.set(style="ticks")

df = sns.load_dataset("anscombe")

# Show the results of a linear regression
# within each dataset
sns.lmplot(x="x", y="y", col="dataset",
            hue="dataset", data=df,
            col_wrap=2, ci=None,
            palette="muted", size=4,
            scatter_kws={"s": 50, "alpha": 1})
```

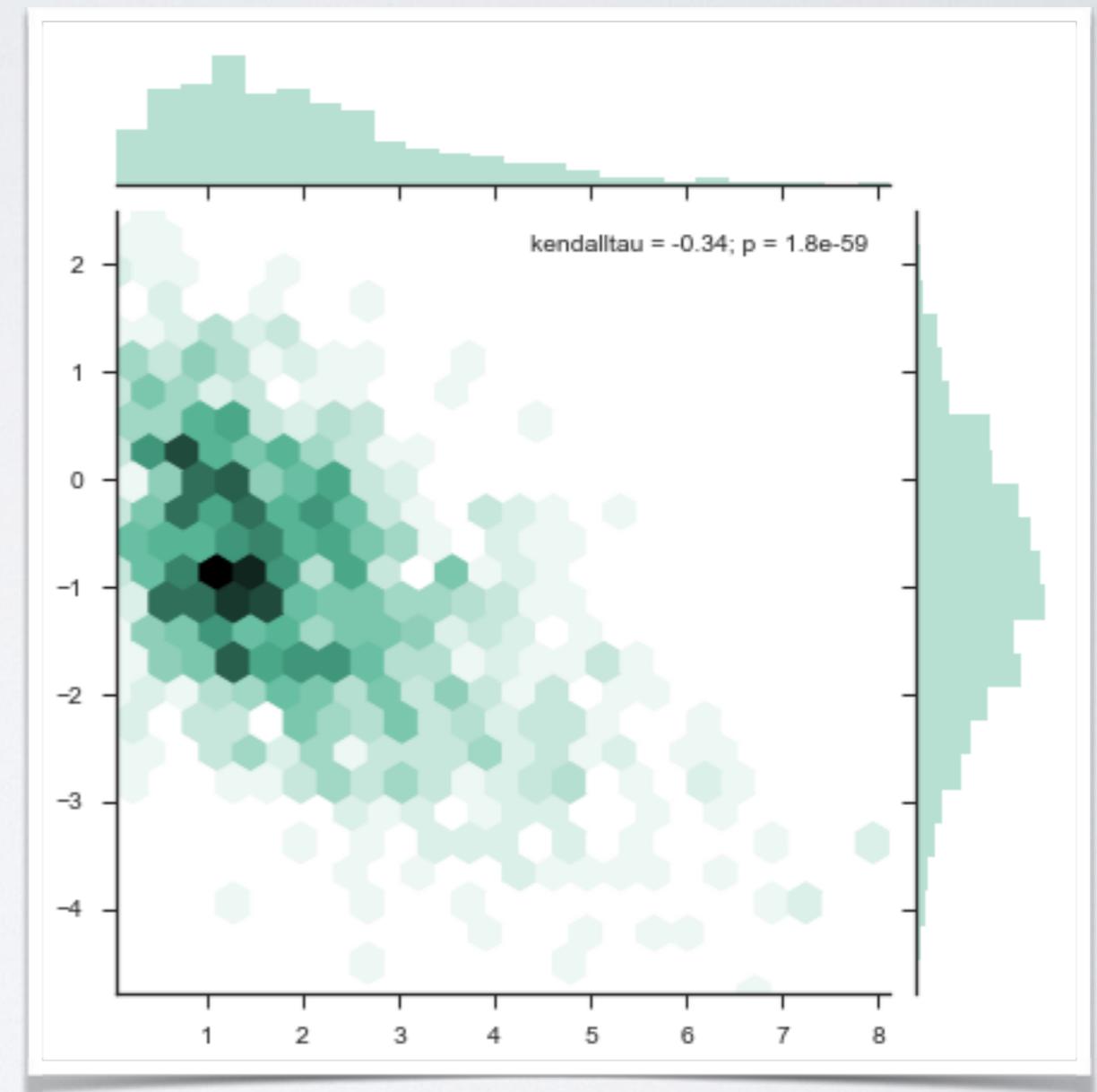


CONVENIENT WRAPPER FUNCTIONS FOR MATPLOTLIB

```
import numpy as np
from scipy.stats import kendalltau
import seaborn as sns
sns.set(style="ticks")

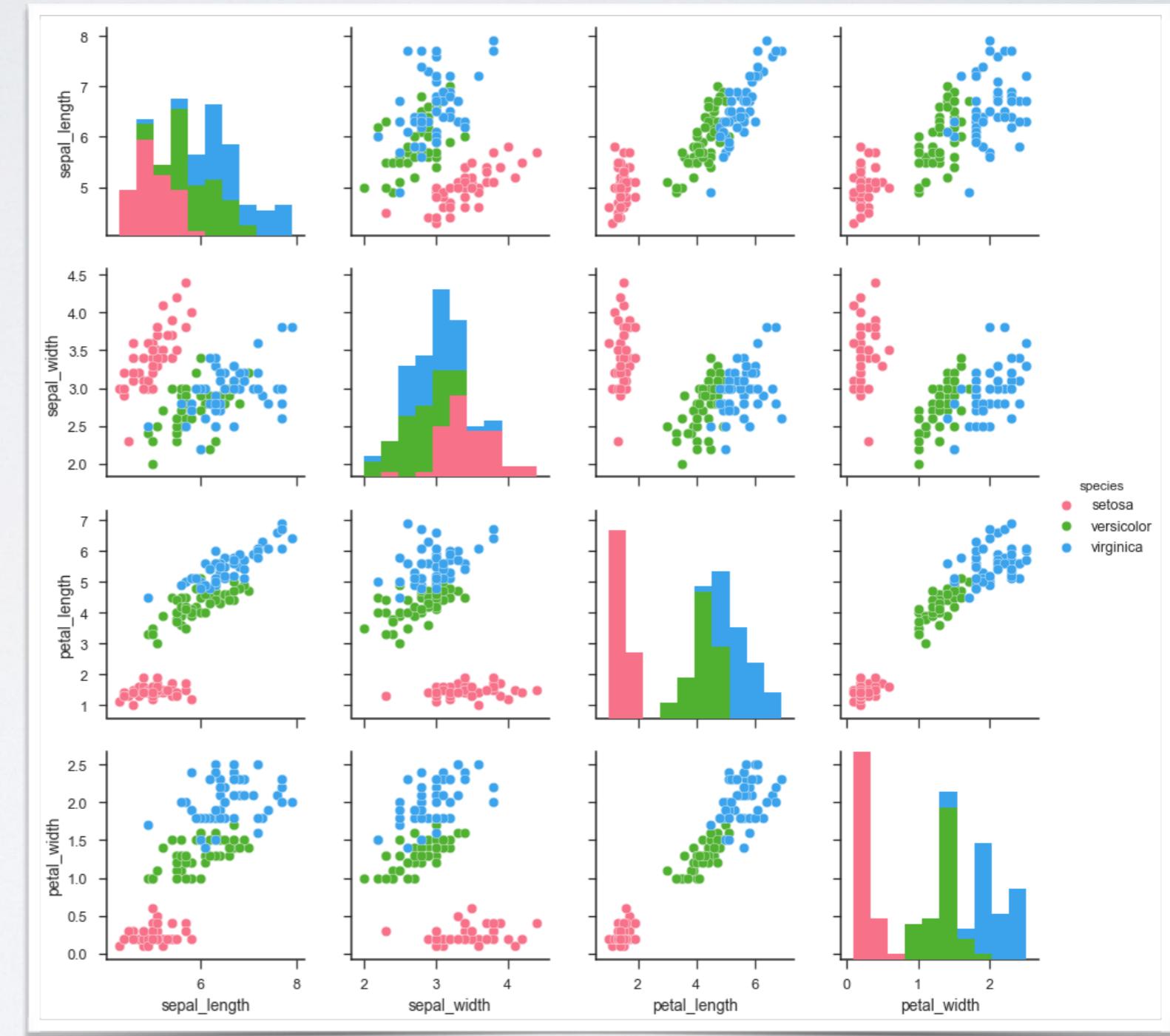
rs = np.random.RandomState(11)
x = rs.gamma(2, size=1000)
y = -.5 * x + rs.normal(size=1000)

sns.jointplot(x, y, kind="hex",
                stat_func=kendalltau,
                color="#4CB391")
```



CONVENIENT WRAPPER FUNCTIONS FOR MATPLOTLIB

```
import seaborn as sns  
sns.set(style="ticks",  
       color_codes=True)  
  
iris = sns.load_dataset("iris")  
sns.pairplot(iris,  
                      hue="species",  
                      palette="husl")
```



You will learn more about
seaborn from **David Kirkby!**

DOCOPT

creates beautiful command-line interfaces

by Vladimir Keleshev

<https://github.com/docopt/docopt>

ARGPARSE/OPTPARSE

Many classes and functions,
default values,
extensive documentation,
very hard to memorise
a basic setup.

The screenshot shows a Python Software Framework window with multiple tabs and code examples related to argparse. The tabs include 'Librerie' and '16.4. argparse...'. The main pane displays code snippets and their outputs:

- Top Tab:** Shows code: `>>> parser.parse_args(['b', '--help'])`, output: usage: PROG b [-h] [--baz {X,Y,Z}] optional arguments: and a note about help output.
- Second Tab:** Shows code: `>>> parser.parse_args(['-h', '--foo FOO'] [bar])`, output: PROG: error: extra arguments found: badger.
- Third Tab:** A detailed section titled '16.4.4.3. Arguments containing -'. It discusses the `parse_args()` method's behavior with negative numbers and positional arguments. It includes code examples for various cases involving `-x`, `-1`, and `-5`.
- Fourth Tab:** A section titled '16.4.4.4. Argument abbreviations (prefix matching)'. It explains how long options can be abbreviated if they are unique. It includes code examples for abbreviating `--bacon` and `--badger`.
- Bottom Tab:** A note about parser-level defaults and the `get_default(dest)` method.

DOCOPT

```
#!/usr/bin/env python
```

```
"""
```

```
Naval Fate.
```

```
Usage:
```

```
    naval_fate ship new <name> ...
    naval_fate ship <name> move <x> <y> [ --speed=<kn>]
    naval_fate ship shoot <x> <y>
    naval_fate mine (set|remove) <x> <y> [ --moored| --drifting]
    naval_fate -h | --help
    naval_fate --version
```

```
Options:
```

-h --help	Show this screen.
--version	Show version.
--speed=<kn>	Speed in knots [default: 10].
--moored	Moored (anchored) mine.
--drifting	Drifting mine.

```
"""
```

```
from docopt import docopt
```

```
arguments = docopt(__doc__, version='Naval Fate 2.0')
```

DOCOPT

```
naval_fate ship Guardian move 10 50 --speed=20
```



```
arguments =  
{  
    "--drifting": false,  
    "--help": false,  
    "--moored": false,  
    "--speed": "20",  
    "--version": false,  
    "<name>": [  
        "Guardian"  
    ],  
    "<x>": "10",  
    "<y>": "50",  
    "mine": false,  
    "move": true,  
    "new": false,  
    "remove": false,  
    "set": false,  
    "ship": true,  
    "shoot": false  
}
```

CLICK

a mature command line utility interface package

<http://click.pocoo.org>

CLICK

- Much more advanced compared to docopt
- The no.1 choice if you want to go crazy with command line utilities

```
import click

@click.command()
@click.option('--count', default=1, help='Number of greetings.')
@click.option('--name', prompt='Your name',
              help='The person to greet.')
def hello(count, name):
    """Simple program that greets NAME for a total of COUNT times."""
    for x in range(count):
        click.echo('Hello %s!' % name)

if __name__ == '__main__':
    hello()
```

SO, WHAT NOW?

FINAL PERSONAL THOUGHTS

I spent a lot of time optimising Python code in the past years, here is a short summary of my personal experience.

- There were several attempts to make Python itself faster w.r.t. low level programming, none of them are satisfying (PyPy may have a future, but still doesn't fully support Python 3), many of them were abandoned
- Think twice (or more) before you bake Cython or any other static compilation into your project. The two language problem is real and it's hard to get it right. The performance gain is often disillusioning compared to the work, workarounds and "mess" one needs to deal with.
- Me and my lovely dev-team made the best experiences with numba
 - no clutter or double bookkeeping, no (static) compilation
 - minimal dependencies (basically only LLVMlite)
 - often orders of magnitudes faster than comparable low level algorithms utilising custom Cython class instances or ctypes
 - downside: super slow without numba ...
- When it comes to high performance code using Python, you have to think in numpy arrays and cannot model your own datatypes like e.g. in C or C++ (structs, classes ...)

MY RECEIPT FOR PERFORMANT PYTHON CODE

- **Avoid massive amounts of Python class instances**

(e.g. don't create a class for a Point and then a list of 10 million points!)

- **Use numpy arrays for large homogenous data**

(w.r.t. the "points" example above, create a 3xN numpy recarray instead, so you can access points.x, points.y and point.z. Subclass the array if you need some special functionality)

- **Vectorisation is a good idea (most of the time).**

For basic operations, you most likely find a dedicated function in numpy or scipy.

- **Try to reuse already allocated memory** (allocations are expensive!)

- **Always profile first, before you do heavy optimisations!**

"[...] premature optimization is the root of all evil." –D. Knuth

Keep in mind, this doesn't mean that you sit down and hack together code, whatever works, this is not what Donald meant! Take care of the basic principles of performant code from the very beginning, otherwise you will have a hard time to refactor.

- **Do not reinvent the wheel.**

You mostly find a lib which does what you need, better, faster and for no cost.

Ohne more thing ...

AN EXAMPLE WHY IT'S SO HARD TO MAKE PYTHON FAST?

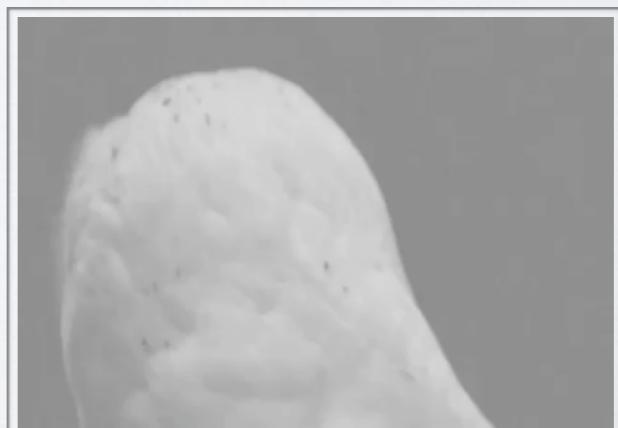
JUST A SIMPLE, BUT CRUCIAL ASPECT ...

- Python lets you do anything.
- Here is a "pure" function, written in Python:

```
def square(x):  
    return float(x)**2
```

- Every decent compiler should now be able to optimise code using this function (repeated calls, tail recursion elimination, inlining, thread safety guarantees, etc.)

```
import builtins  
builtins.float = int
```



THANK YOU!

...also many thanks to Vincent and Jayesh,
and the whole organising committee!

Acknowledgement

- H2020-Astronomy ESFRI and Research Infrastructure Cluster (Grant Agreement number: 653477).

Useful Python Libraries

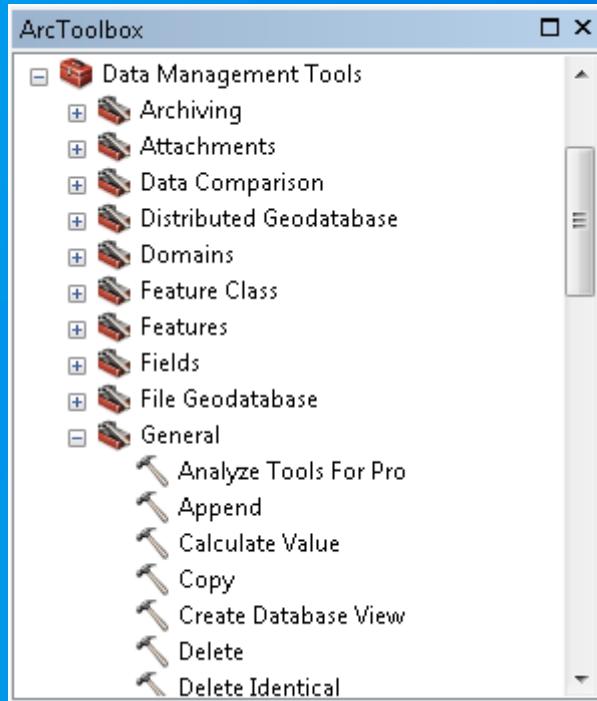
Ghislain Prince

This presentation

- Standard libraries
 - “batteries included”
- 3rd party libs

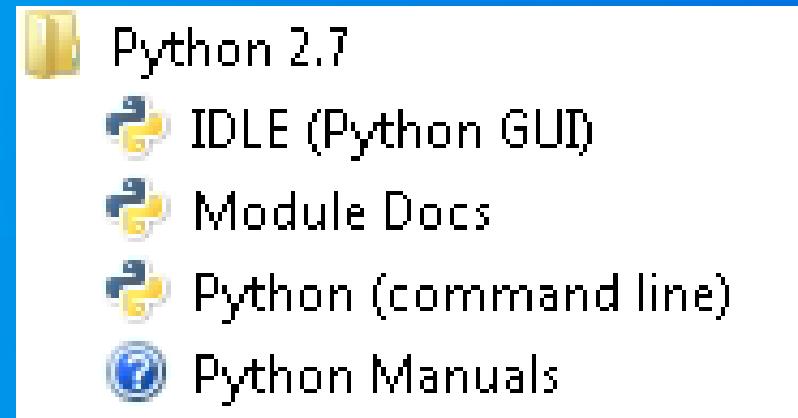
Regarding python 3

- docs.python.org/2/library/2to3.html
- “Analyze tools for Pro” tool
- Future : pick you python (Conda)



Standard Library is Large, powerful and Well-Documented

- docs.python.org
- Installed with ArcGIS



3rd Party Library

- Not part of standard library
- Pypi – the Python Package Index
 - 60,000+ packages
 - <https://pypi.python.org/pypi>
 - pip
- C:\>c:\Python27\ArcGIS10.3\Scripts\pip.exe install requests
- Downloading requests-2.7.0-py2.py3-none-any.whl (470kB)
- 100% |#####| 471kB 718kB/s
- Installing collected packages: requests
- Successfully installed requests-2.7.0

Data Formats

- XML
- JSON
- CSV
- Excel
- PDF
- Numpy/Pandas*
- NetCDF

3 XML options within the standard library

- SAX
 - Can handle huge datasets
 - Hard to use, verbose code
- DOM
 - Familiar: Javascript uses the DOM
 - Easy (ish)
- Etree
 - Feels like python
 - Extremely easy and concise for common tasks

JSON

- json module

```
>>> import json  
>>> d = json.load(open("states.json", 'r'))  
>>> print(type(d))  
<class 'dict'>
```

CSV & Excel

- csv module

```
>>> import csv  
>>> with open('eggs.csv', 'rb') as csvfile:  
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')  
...     for row in spamreader:  
...         print ', '.join(row)  
Spam, Spam, Spam, Spam, Spam, Baked Beans  
Spam, Lovely Spam, Wonderful Spam
```

- xlrd & xlwt (third-party, included in ArcGIS)

PDF

- **arcpy.mapping.PDFDocument**
- **reportlab (third-party)**
 - *allows rapid creation of rich PDF documents, and also creation of charts in a variety of bitmap and vector formats.*

Networking

- Calls to HTTP servers
 - `urllib2`
 - `requests` (third-party, pip-install)
 - `asyncio`

Computing

- Numpy
- Pandas *
- Scipy *
- Sympy *
- R bridge (coming soon: github.com/R-ArcGIS ... wait this isn't python at all)

- * Pro 1.0 & planned for Desktop/Server 10.4

Resources

arcpy.wordpress.com

twitter.com/arcpy

desktop.arcgis.com/en/desktop/



Understanding our world.

VINOD KUMAR VERMA, PGT(CS), KV OEF KANPUR &
SACHIN BHARDWAJ, PGT(CS), KV NO.1 TEZPUR

USING PYTHON LIBRARIES

COLLECTION OF MODULES

Introduction

- As our program become larger and more complex the need to organize our code becomes greater. We have already learnt in ***Function chapter that large and complex program should be divided into functions*** that perform a specific task. As we write more and more ***functions in a program, we should consider organizing of functions by storing them in modules***
- A module is simply a file that contains Python code. When we break a program into modules, each modules should contain functions that perform related tasks.
- Commonly used modules that contains source code for generic needs are called ***Libraries***.

Introduction

- When we speak about working with libraries in Python, we are, in fact, working with modules that are created inside Library or Packages. Thus a Python program comprises three main components:
 - Library or Package
 - Module
 - Function/Sub-routine

Relationship between Module, Package and Library in Python

- A **Module** is a file containing Python definitions (docstrings) , functions, variables, classes and statements
- Python package is simply a directory of Python module(s)
- Library is a collection of various packages. Conceptually there is no difference between package and Python library. In Python a library is used to loosely describe a collection of core or main modules

Commonly used Python libraries

STANDARD LIBRARY

math module	Provides mathematical functions
cmath module	Provides function for complex numbers
random module	For generating random numbers
Statistics module	Functions for statistical operation
Urllib	Provides URL handling functions so that you can access websites from within your program.
NumPy library	This library provides some advance math functionalities along with tools to create and manipulate numeric arrays
SciPy library	Another useful library that offers algorithmic and mathematical tools for scientific calculation
Tkinter library	Provides traditional user interface toolkit and helps you to create user friendly GUI interface for different types of applications.
Matplotlib library	Provides functions and tools to produce quality output in variety of formats such as plot, charts, graph etc,

What is module?

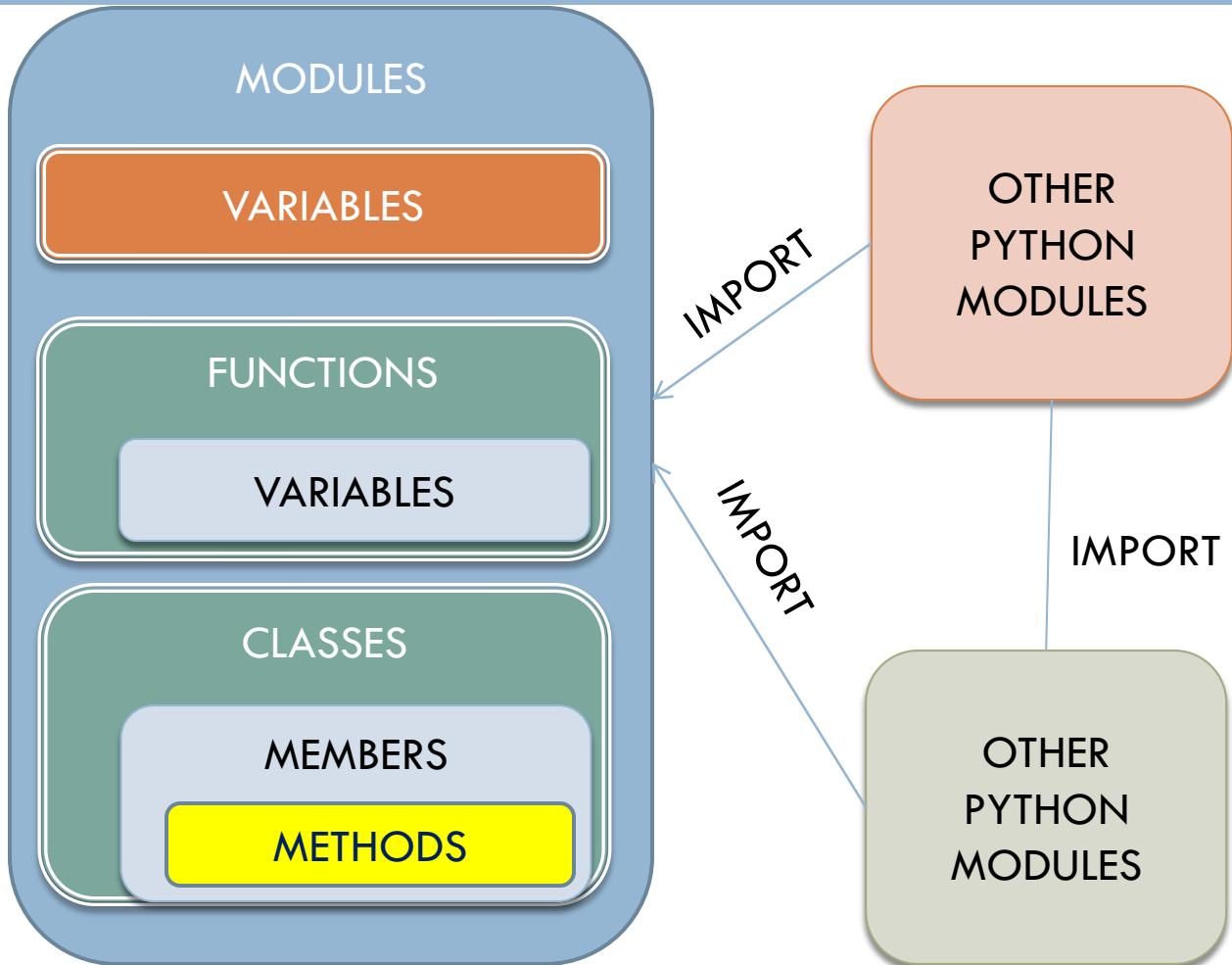
- Act of partitioning a program into individual components(modules) is called modularity. A module is a separate unit in itself.
 - It reduces its complexity to some degree
 - It creates numbers of well-defined, documented boundaries within program.
 - Its contents can be reused in other program, without having to rewrite or recreate them.

Structure of Python module

- A python module is simply a normal python file(.py) and contains functions, constants and other elements.
- Python module may contains following objects:

docstring	Triple quoted comments. Useful for documentation purpose
Variables and constants	For storing values
Classes	To create blueprint of any object
Objects	Object is an instance of class. It represent class in real world
Statements	Instruction
Functions	Group of statements

Composition/Structure of python module



Importing Python modules

- To import entire module
 - **import <module name>**
 - **Example:** `import math`
- To import specific function/object from module:
 - **from <module_name> import <function_name>**
 - **Example:** `from math import sqrt`
- **import *** : can be used to import all names from module into current calling module

Accessing function/constant of imported module

- To use function/constant/variable of imported module we have to specify module name and function name separated by dot(.). This format is known as dot notation.
 - <module_name>.<function_name>
 - **Example:** `print(math.sqrt(25))`

Example : import module_name

The screenshot shows a Python IDE window titled "mod1 - C:/Python27/mod1". The menu bar includes File, Edit, Format, Run, Options, Windows, and Help. The code editor contains the following Python script:

```
import math
print(math.sqrt(25))
print(math.log10(100))
```

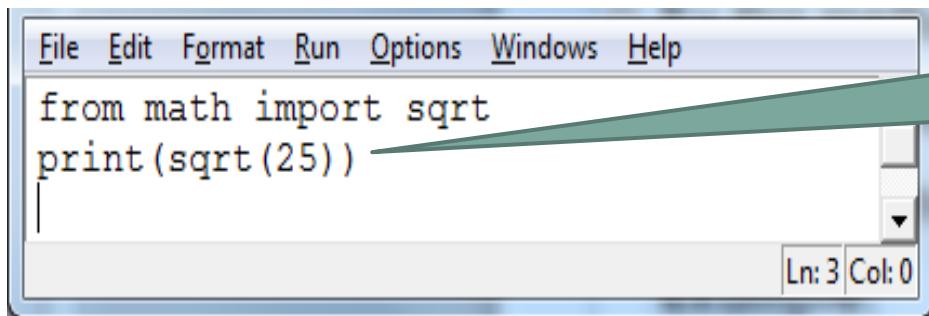
The output window below shows the results of the execution:

```
5.0
2.0
>>> |
```

Here we can see that after **import math**, we have to qualify the name of function with name of package using dot(.) notation

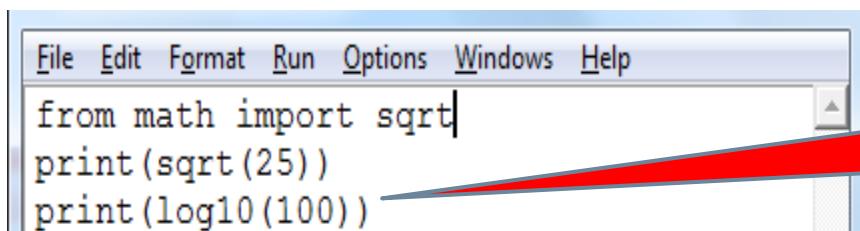
Example: from module import function

- By this method only particular method will be added to our current program. We need not to qualify name of method with name of module. Or example:



```
File Edit Format Run Options Windows Help
from math import sqrt
print(sqrt(25))
Ln: 3 Col: 0
```

Here function
sqrt() is
directly written

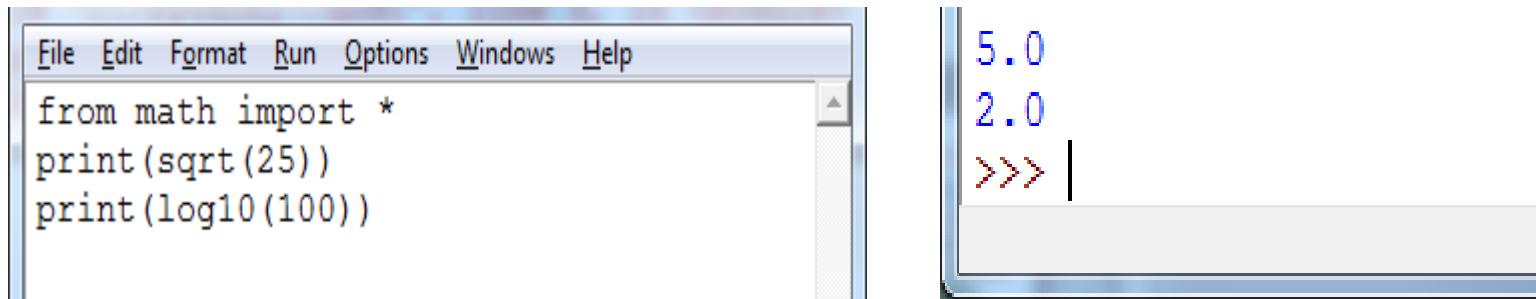


```
File Edit Format Run Options Windows Help
from math import sqrt
print(sqrt(25))
print(log10(100))
Ln: 3 Col: 0
```

This line will not
be executed and
gives an error

Example: from module import *

- It is similar to importing the entire package as “import package” but by this method qualifying each function with module name is not required.



The image shows a screenshot of a Python IDLE window. On the left, the code editor pane contains the following Python code:

```
File Edit Format Run Options Windows Help
from math import *
print(sqrt(25))
print(log10(100))
```

On the right, the interactive shell pane shows the output:

```
5.0
2.0
>>> |
```

We can also import multiple elements of module as :
from math import sqrt, log10

Creating our own Module

- Create new python file(.py) and type the following code as:

```
import math
# the is my first module
mynum=100
def area_rect(length,breadth):
    return length*breadth

def area_square(side):
    return side*side

def area_circle(rad):
    return math.pi*rad*rad
```

Execute the following code to import and use your own module

```
>>> import area
>>> print(area.area_rect(7,8))
56
>>> print(area.mynum)
100
>>> from area import area_square
>>> area_square(8)
64
>>> area_circle(8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'area_circle' is not defined
>>> 
```

Save this file are “area.py”

help() function

- Is used to get detailed information about any module like : name of module, functions inside module, variables inside module and name of file etc.

```
>>> help(area)
Help on module area:

NAME
    area

FUNCTIONS
    area_circle(rad)

    area_rect(length, breadth)

    area_square(side)

DATA
    mynum = 100

FILE
    c:\users\vin\area.py

>>>
```

Namespace

- Is a space that holds a bunch of names. Consider an example:
 - In a CCA competition of vidyalaya, there are students from different classes having similar names, say there are three POOJA GUPTA, one from class X, one from XI and one from XII
 - As long as they are in their class there is no confusion, since in X there is only one POOJA GUPTA, and same with XI and XII
 - But problem arises when the students from X, XI, XII are sitting together, now calling just POOJA GUPTA would create confusion-which class's POOJA GUPTA. So one need to qualify the name as class X's POOJA GUPTA, or XI's or XII's and so on.

Namespace

- From the previous example, we can say that class X has its own namespace where there no two names as POOJA GUPTA; same holds for XI and XII
- In Python terms, namespace can be thought of as a named environment holding logical group of related objects.
- For every python module(.py), Python creates a namespace having its name similar to that of module's name. That is, namespace of module AREA is also AREA.
- **When 2 namespace comes together, to resolve any kind of object name dispute, Python asks you to qualify the name of object as <modulename>.<objectname>**

Processing of **import <module>**

- The code of import module is interpreted and executed
- Defined functions and variables in the module are now available to program in **new namespace created by the name of module**
- **For example, if the imported module is area, now you want to call the function area_circle(), it would be called as area.area_circle()**

Processing of `from module import object`

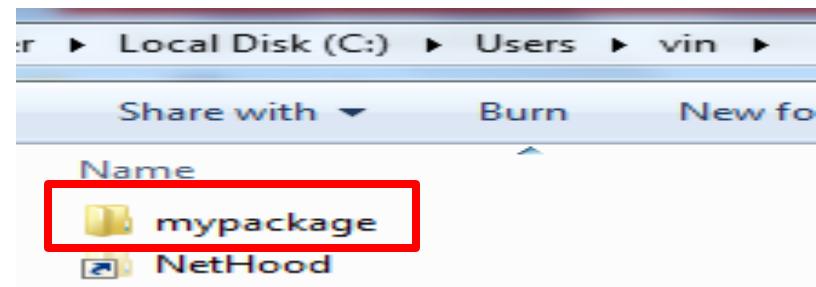
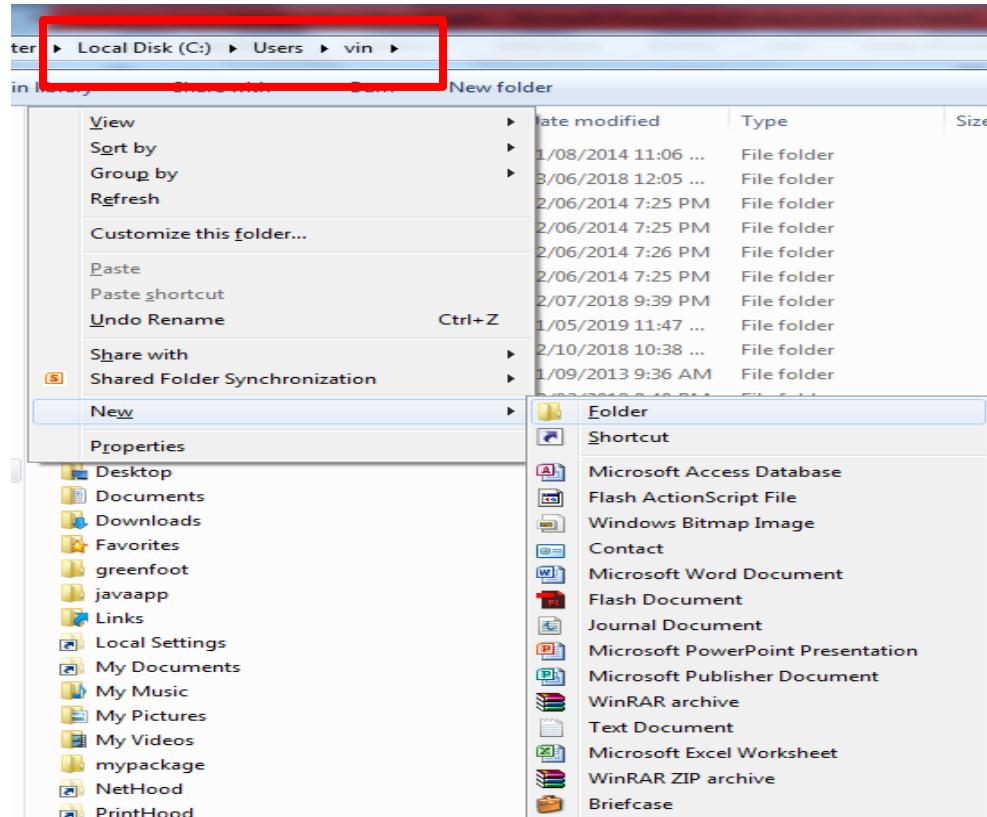
- When we issue `from module import object` command:
 - The code of imported module is interpreted and executed
 - Only the asked function and variables from module are now available in the current namespace i.e. **no new namespace is created that's why we can call object of imported module without qualifying the module name**
 - For example:

```
from math import sqrt
print(sqrt(25))
```
 - *However if the same function name is available in current namespace then local function will hide the imported module's function*
 - *Same will be apply for `from math import *` method*

Creating Package

□ Step 1

- Create a new folder which you want to act as package. The name of folder will be the name of your package



IN THE C:\USERS\VIN
A new Folder “mypackage” is
created.

**Note: you can create folder in
any desired location**

Creating Package

- Step 2: Create modules (.py) and save it in “mypackage” folder

area.py

```
import math
# the is my first module
mynum=100
def area_rect(length,breadth):
    return length*breadth

def area_square(side):
    return side*side

def area_circle(rad):
    return math.pi*rad*rad
```

numcheck.py

```
import math
def even(num):
    if num % 2 == 0:
        return 1
    else:
        return 0

def isprime(num):
    for i in range(2,int(math.sqrt(num)+1)):
        if num % i == 0:
            return 0
    return 1
def palindrome(num):
    mynum = num
    n = 0
    while num!=0:
        r = num % 10
        n = n*10 + r
        num = num // 10
    if mynum == n:
        return 1
    else:
        return 0
```

Creating Package

- Step 2: importing package and modules in python program

```
import mypackage.numcheck  
n = int(input("Enter number "))  
if(mypackage.numcheck.isprime(n)):  
    print("Number is Prime")  
else:  
    print("Number is Composite")
```

Save this file by
“anyname.py” outside
the package folder

RUN THE PROGRAM

Creating Alias of Package/module

- Alias is the another name for imported package/module. It can be used to shorten the package/module name

```
import mypackage.numcheck as mpack
n = int(input("Enter number "))
if(mpack.isprime(n)):
    print("Number is Prime")
else:
    print("Number is Composite")
```

Save this file by
“anyname.py” outside
the package folder

RUN THE PROGRAM

__init__.py file

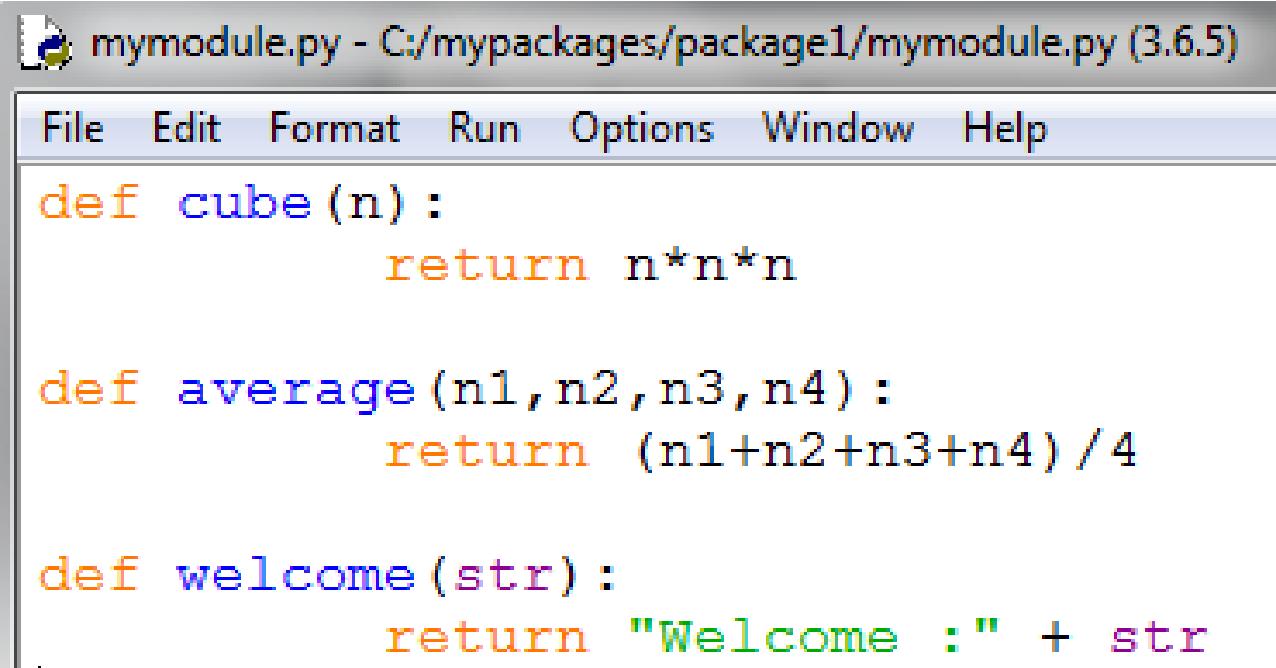
- **__init__.py** (double underscore is prefixed and suffixed) : This file is required to make Python treat directories containing the file as packages.
- **__init__.py** can be just empty file, but it can also execute initialization code for the package.
- **Note: Python 2 requires __init__.py to be inside a folder in order for the folder to be considered a package and made importable but in Python 3.3 and above, it support implicit namespace packages, all folders are packages regardless of the presence of a __init__.py file**
- **So from Python 3.3 it is optional to create __init__.py file**

Example - creating and using package and module and `__init__.py`

- Create a folder to act as a package
- For. e.g. In C: (C Drive) a folder “mypackages” is created
- Create `__init__.py` file (do not write any thing in it) in this folder “mypackages”
- Now create another folder “package1” inside “mypackages”
- Create `__init__.py` file inside “package1” also
- Create a module(.py) for e.g. “mymodule.py” file in “package1” to have some functions in it.

Another example of creating and using package and module and `__init__.py`

□ `mymodule.py`



The screenshot shows a code editor window titled "mymodule.py - C:/mypackages/package1/mymodule.py (3.6.5)". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code itself contains three functions:

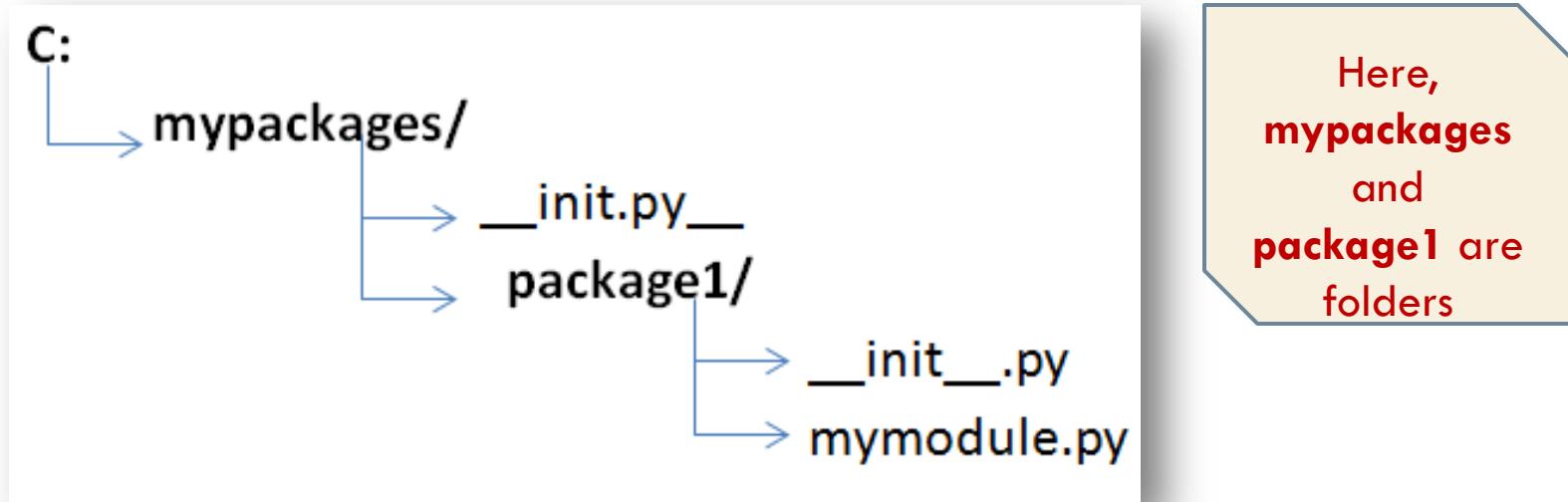
```
def cube(n):
    return n*n*n

def average(n1, n2, n3, n4):
    return (n1+n2+n3+n4) / 4

def welcome(str):
    return "Welcome :" + str
```

Another example of creating and using package and module and `__init__.py`

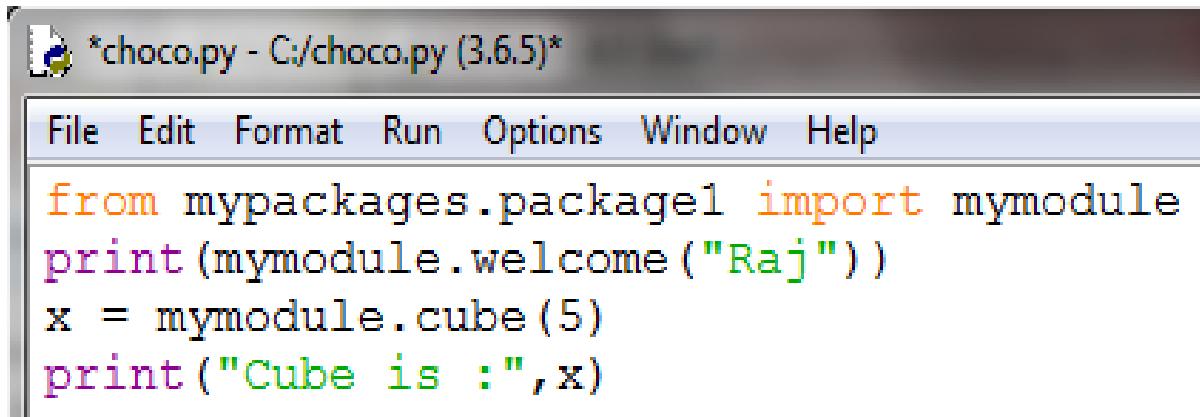
- Now our entire contents will be like this:



- Now in C: (C Drive), create a file for e.g. “choco.py” in which we will import module “mymodule”

Another example of creating and using package and module and `__init__.py`

- To import the packages, we can either use **Absolute addressing** or **Relative Addressing**. (already discussed in data file handling chapter)
- Absolute means following the complete address whereas Relative means with relation to current folder by using Single dot(.)

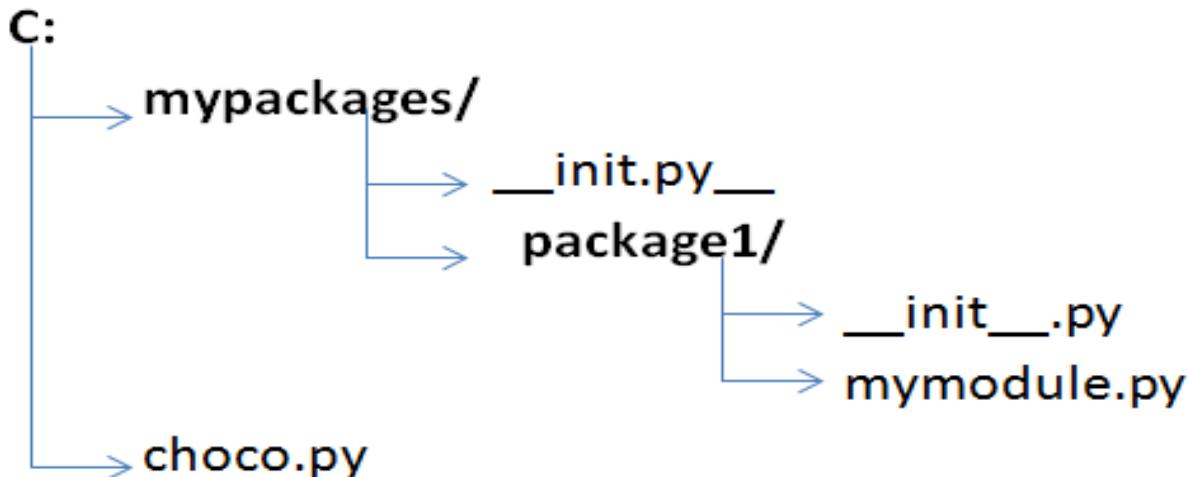


```
*choco.py - C:/choco.py (3.6.5)*
File Edit Format Run Options Window Help
from mypackages.package1 import mymodule
print(mymodule.welcome("Raj"))
x = mymodule(cube(5))
print("Cube is :", x)
```

Absolute path of package

Another example of creating and using package and module and `__init__.py`

- Now our entire content structure will be as:



- Run the `choco.py` file and you will get the output.

TIP : to use package from any location

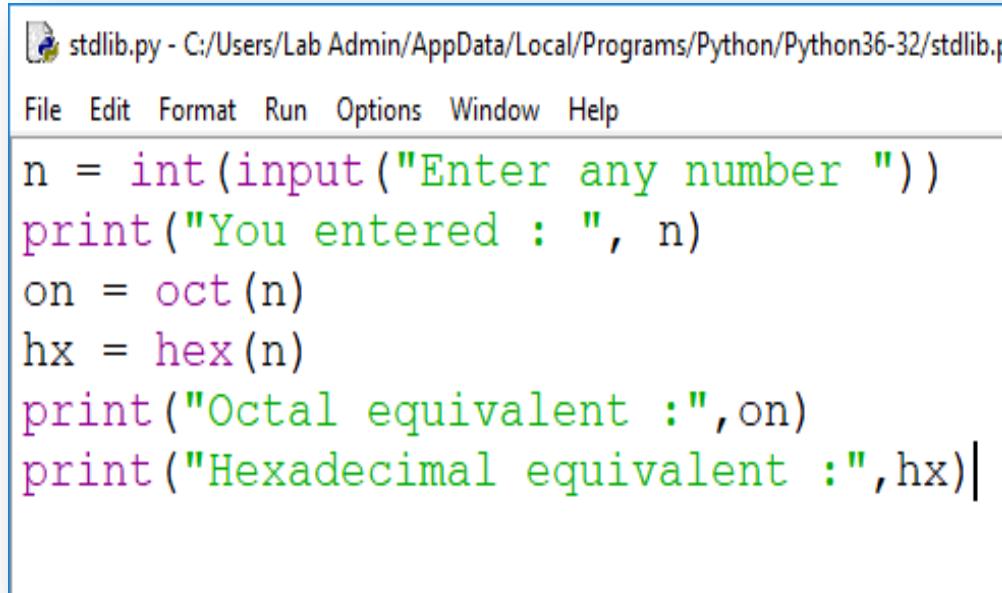
- At this time, the package “mypackages” will be accessible from its location only i.e. the file which wants to import this package must be in same folder/drive where “mypackages” is.
- To enable “mypackages” to be used from any location, Copy this mypackages to Python’s **site-packages** folder inside **Lib** folder of Python’s installation folder. (*Try with copying to Lib folder also*)
- Path of site-packages is :
C:\Users\vin\AppData\Local\Programs\Python\Python 36-32\Lib\site-packages
- After this you can import this package “mypackages” in any python file.

Using Python's Built-in Function

- Python's standard library is very extensive that offers many built-in functions that **we can use without having to import any library.**
- Using Python's Built-in functions
 - `Function_name()`

Mathematical and String functions

- `oct(int)` : return octal string for given number by prefixing “0o”
- `hex(int)` : return octal string for given number by prefixing “0x”



```
stdlib.py - C:/Users/Lab Admin/AppData/Local/Programs/Python/Python36-32/stdlib.py
File Edit Format Run Options Window Help
n = int(input("Enter any number "))
print("You entered : ", n)
on = oct(n)
hx = hex(n)
print("Octal equivalent :",on)
print("Hexadecimal equivalent :",hx)|
```

```
Enter any number 12
You entered : 12
Octal equivalent : 0o14
Hexadecimal equivalent : 0xc
```

Mathematical and String functions

- `int(number)` : function convert the fractional number to integer
- `int(string)` : convert the given string to integer
- `round(number,[nDIGIT])` : return number rounded to nDIGIT after decimal points. If nDIGIT is not given, it returns nearest integer to its input.
- Examples: (next slide)

Mathematical and String functions

```
n1 = float(input("Enter first number "))
n2 = float(input("Enter second number "))
n3 = n1/n2
print("Result :",n3)
```

Enter first number 5
 Enter second number 3
 Result : 1.6666666666666667

```
n1 = float(input("Enter first number "))
n2 = float(input("Enter second number "))
n3 = n1/n2
print("Result :",round(n3,2))
```

Enter first number 5
 Enter second number 3
 Result : 1.67

```
n1 = float(input("Enter first number "))
n2 = float(input("Enter second number "))
n3 = n1/n2
print("Result :",round(n3))
```

Enter first number 5
 Enter second number 3
 Result : 2

Other String function

- We have already used many string function in class XI, here are few new functions
 - **<string>.join()** : if the string based iterator is a string then the <string> is inserted after every character of the string.
 - If the string based iterator is a list or tuple of strings then, the given string/character is joined after each member of the list or tuple. **BUT the tuple or list must have all members as string otherwise Python will raise an error**
- Examples (next slide)

Other String function

```
Python 3.6.4 (v3.6.4:d48ebeb, Dec 19 2017, 06:04:45) [MSC v.1900
on win32
Type "copyright", "credits" or "license()" for more information.
>>> "#".join("INDIA")
'I#N#D#I#A'
>>> "$$".join(("IPL","CCL","FIFA"))
'IPL$$CCL$$FIFA'
>>> "$$".join(("IPL","FIFA",123))
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    "$$".join(("IPL","FIFA",123))
TypeError: sequence item 2: expected str instance, int found
>>>
```

Other String function

- We have already used many string function in class XI, here are few new functions
 - **<string>.split()** : allow to divide string in multiple parts and store it as a LIST. If you do not provide delimiter then by default string will be split using space otherwise using given character.
 - **<str>.replace()** : allows you to replace any part of string with another string.
 - **Example (NEXT SLIDE)**

Example (split() and replace())

```
msg = "KV OEF KANPUR"
msg1 = msg.split()      # by default split the string using space
print(msg1) # output will be in the form of list
print(type(msg1))
```

```
msg2="kv@gmail.com"
msg3=msg2.split("@")    #now split using '@'
print(msg3) # output will be in the form of list
print(type(msg3))
```

```
mymsg="I love Python Programming"
mymsg2=mymsg.replace("Python","C++") #replaces 'Python' with 'C++'
print(mysmg2)
```

```
['KV', 'OEF', 'KANPUR']
<class 'list'>
['kv', 'gmail.com']
<class 'list'>
>>>
RESTART: C:/Users/Lab A
Y
['KV', 'OEF', 'KANPUR']
<class 'list'>
['kv', 'gmail.com']
<class 'list'>
I love C++ Programming
```

Using **URLLIB** and **WEBBROWSER** modules

- Python provides **urllib** module to send http request and receive the result from within your program. To use **urllib** we have to first import **urllib** module.
- **urllib** module is a collection of sub-module like request, error, parse etc. following functions of **urllib** we can use: (**next slide**)

Functions of URLLIB

FUNCTION NAME	PURPOSE
urllib.request.urlopen(<url>)	Opens a website or network object denoted by URL for reading and return file like object using which other functions are often used
urlopen_object.read()	Return HTML or the source code of given url
urlopen_object..getcode()	Returns HTTP status code where 200 means 'all okay' 404 means url not found etc. 301, 302 means some kind of redirections happened
urlopen_object.headers	Stores metadata about the open URL
urlopen_object.info()	Returns same information as by headers
urlopen_object.geturl()	Return URL string

Example:

```
import urllib.request
x=urllib.request.urlopen('https://www.google.com/')
print("URL READ : ",x.geturl())
print("HEADERS : ",x.headers)
print("INFO : ",x.info())
print("CODE : ",x.getcode())
```

```
(base) C:\Users\vin>python use.py
URL READ : https://www.google.com/
HEADERS : Date: Thu, 16 May 2019 17:14:45 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: 1P_JAR=2019-05-16-17; expires=Sat, 15-Jun-2019 17:14:45 GMT; path=/;
domain=.google.com
Set-Cookie: NID=183=hBwj19cMG2hFqDHUVKRsoPE-JvwJRCcz2kSmxL6eWtJNoNozzUj_zwrM-Ia
uIr7XDb5XHUu8G4aM27IKMETI4GY9l0pA57buFyJIY_CZ2-tYtEeUk1sMB7aExBAigAq2XJgYmMmaF_
ufWmwVxbvpku2quvFSBIiFWj8NcqywQ; expires=Fri, 15-Nov-2019 17:14:45 GMT; path=/;
domain=.google.com; HttpOnly
Alt-Svc: quic=":443"; ma=2592000; v="46,44,43,39"
Accept-Ranges: none
Vary: Accept-Encoding
Connection: close
CODE : 200
```

Before execution of the above code, make sure PC is connected to working internet.

WEBBROWSER MODULE

- Provides functionality to open a website in a window or tab or web browser on your computer, from within your program.
- To use webbrowser module we must import the module as:

□ **import webbrowser**

```
import webbrowser
myurl = input("Enter url to open ")
webbrowser.open_new(myurl)
```

```
import webbrowser
text = input("Enter 'Text to Search Online ' :")
webbrowser.open_new("https://www.google.com?q="+text)
```

The Python Library Reference

Release 3.8.2

**Guido van Rossum
and the Python development team**

February 28, 2020

**Python Software Foundation
Email: docs@python.org**

CONTENTS

1	Introduction	3
1.1	Notes on availability	3
2	Built-in Functions	5
3	Built-in Constants	27
3.1	Constants added by the <code>site</code> module	28
4	Built-in Types	29
4.1	Truth Value Testing	29
4.2	Boolean Operations — <code>and</code> , <code>or</code> , <code>not</code>	29
4.3	Comparisons	30
4.4	Numeric Types — <code>int</code> , <code>float</code> , <code>complex</code>	30
4.5	Iterator Types	36
4.6	Sequence Types — <code>list</code> , <code>tuple</code> , <code>range</code>	37
4.7	Text Sequence Type — <code>str</code>	43
4.8	Binary Sequence Types — <code>bytes</code> , <code>bytearray</code> , <code>memoryview</code>	53
4.9	Set Types — <code>set</code> , <code>frozenset</code>	75
4.10	Mapping Types — <code>dict</code>	77
4.11	Context Manager Types	81
4.12	Other Built-in Types	82
4.13	Special Attributes	85
5	Built-in Exceptions	87
5.1	Base classes	88
5.2	Concrete exceptions	88
5.3	Warnings	94
5.4	Exception hierarchy	95
6	Text Processing Services	97
6.1	<code>string</code> — Common string operations	97
6.2	<code>re</code> — Regular expression operations	108
6.3	<code>difflib</code> — Helpers for computing deltas	127
6.4	<code>textwrap</code> — Text wrapping and filling	138
6.5	<code>unicodedata</code> — Unicode Database	142
6.6	<code>stringprep</code> — Internet String Preparation	144
6.7	<code>readline</code> — GNU readline interface	145
6.8	<code>rlcompleter</code> — Completion function for GNU readline	150
7	Binary Data Services	151
7.1	<code>struct</code> — Interpret bytes as packed binary data	151

7.2	<code>codecs</code> — Codec registry and base classes	157
8	Data Types	175
8.1	<code>datetime</code> — Basic date and time types	175
8.2	<code>calendar</code> — General calendar-related functions	210
8.3	<code>collections</code> — Container datatypes	215
8.4	<code>collections.abc</code> — Abstract Base Classes for Containers	232
8.5	<code>heapq</code> — Heap queue algorithm	236
8.6	<code>bisect</code> — Array bisection algorithm	240
8.7	<code>array</code> — Efficient arrays of numeric values	242
8.8	<code>weakref</code> — Weak references	245
8.9	<code>types</code> — Dynamic type creation and names for built-in types	253
8.10	<code>copy</code> — Shallow and deep copy operations	257
8.11	<code>pprint</code> — Data pretty printer	258
8.12	<code>reprlib</code> — Alternate <code>repr()</code> implementation	264
8.13	<code>enum</code> — Support for enumerations	266
9	Numeric and Mathematical Modules	285
9.1	<code>numbers</code> — Numeric abstract base classes	285
9.2	<code>math</code> — Mathematical functions	288
9.3	<code>cmath</code> — Mathematical functions for complex numbers	294
9.4	<code>decimal</code> — Decimal fixed point and floating point arithmetic	298
9.5	<code>fractions</code> — Rational numbers	325
9.6	<code>random</code> — Generate pseudo-random numbers	328
9.7	<code>statistics</code> — Mathematical statistics functions	334
10	Functional Programming Modules	345
10.1	<code>itertools</code> — Functions creating iterators for efficient looping	345
10.2	<code>functools</code> — Higher-order functions and operations on callable objects	360
10.3	<code>operator</code> — Standard operators as functions	368
11	File and Directory Access	377
11.1	<code>pathlib</code> — Object-oriented filesystem paths	377
11.2	<code>os.path</code> — Common pathname manipulations	393
11.3	<code>fileinput</code> — Iterate over lines from multiple input streams	398
11.4	<code>stat</code> — Interpreting <code>stat()</code> results	401
11.5	<code>filecmp</code> — File and Directory Comparisons	406
11.6	<code>tempfile</code> — Generate temporary files and directories	408
11.7	<code>glob</code> — Unix style pathname pattern expansion	412
11.8	<code>fnmatch</code> — Unix filename pattern matching	414
11.9	<code>linecache</code> — Random access to text lines	415
11.10	<code>shutil</code> — High-level file operations	415
12	Data Persistence	425
12.1	<code>pickle</code> — Python object serialization	425
12.2	<code>copyreg</code> — Register <code>pickle</code> support functions	442
12.3	<code>shelve</code> — Python object persistence	443
12.4	<code>marshal</code> — Internal Python object serialization	445
12.5	<code>dbm</code> — Interfaces to Unix “databases”	447
12.6	<code>sqlite3</code> — DB-API 2.0 interface for SQLite databases	451
13	Data Compression and Archiving	473
13.1	<code>zlib</code> — Compression compatible with <code>gzip</code>	473
13.2	<code>gzip</code> — Support for <code>gzip</code> files	477
13.3	<code>bz2</code> — Support for <code>bzip2</code> compression	480

13.4	<code>lzma</code> — Compression using the LZMA algorithm	484
13.5	<code>zipfile</code> — Work with ZIP archives	490
13.6	<code>tarfile</code> — Read and write tar archive files	499
14	File Formats	511
14.1	<code>csv</code> — CSV File Reading and Writing	511
14.2	<code>configparser</code> — Configuration file parser	518
14.3	<code>netrc</code> — netrc file processing	535
14.4	<code>xdrlib</code> — Encode and decode XDR data	536
14.5	<code>plistlib</code> — Generate and parse Mac OS X .plist files	539
15	Cryptographic Services	543
15.1	<code>hashlib</code> — Secure hashes and message digests	543
15.2	<code>hmac</code> — Keyed-Hashing for Message Authentication	553
15.3	<code>secrets</code> — Generate secure random numbers for managing secrets	555
16	Generic Operating System Services	559
16.1	<code>os</code> — Miscellaneous operating system interfaces	559
16.2	<code>io</code> — Core tools for working with streams	611
16.3	<code>time</code> — Time access and conversions	624
16.4	<code>argparse</code> — Parser for command-line options, arguments and sub-commands	633
16.5	<code> getopt</code> — C-style parser for command line options	666
16.6	<code>logging</code> — Logging facility for Python	668
16.7	<code>logging.config</code> — Logging configuration	684
16.8	<code>logging.handlers</code> — Logging handlers	694
16.9	<code>getpass</code> — Portable password input	707
16.10	<code>curses</code> — Terminal handling for character-cell displays	708
16.11	<code>curses.textpad</code> — Text input widget for curses programs	725
16.12	<code>curses.ascii</code> — Utilities for ASCII characters	727
16.13	<code>curses.panel</code> — A panel stack extension for curses	729
16.14	<code>platform</code> — Access to underlying platform's identifying data	730
16.15	<code>errno</code> — Standard errno system symbols	733
16.16	<code>ctypes</code> — A foreign function library for Python	739
17	Concurrent Execution	773
17.1	<code>threading</code> — Thread-based parallelism	773
17.2	<code>multiprocessing</code> — Process-based parallelism	786
17.3	<code>multiprocessing.shared_memory</code> — Provides shared memory for direct access across processes	829
17.4	<code>The concurrent package</code>	834
17.5	<code>concurrent.futures</code> — Launching parallel tasks	834
17.6	<code>subprocess</code> — Subprocess management	840
17.7	<code>sched</code> — Event scheduler	858
17.8	<code>queue</code> — A synchronized queue class	860
17.9	<code>_thread</code> — Low-level threading API	863
17.10	<code>_dummy_thread</code> — Drop-in replacement for the <code>_thread</code> module	866
17.11	<code>dummy_threading</code> — Drop-in replacement for the <code>threading</code> module	866
18	contextvars — Context Variables	867
18.1	Context Variables	867
18.2	Manual Context Management	868
18.3	asyncio support	870
19	Networking and Interprocess Communication	871
19.1	<code>asyncio</code> — Asynchronous I/O	871
19.2	<code>socket</code> — Low-level networking interface	959

19.3	<code>ssl</code> — TLS/SSL wrapper for socket objects	983
19.4	<code>select</code> — Waiting for I/O completion	1018
19.5	<code>selectors</code> — High-level I/O multiplexing	1025
19.6	<code>asyncore</code> — Asynchronous socket handler	1029
19.7	<code>asynchat</code> — Asynchronous socket command/response handler	1033
19.8	<code>signal</code> — Set handlers for asynchronous events	1035
19.9	<code>mmap</code> — Memory-mapped file support	1042
20	Internet Data Handling	1047
20.1	<code>email</code> — An email and MIME handling package	1047
20.2	<code>json</code> — JSON encoder and decoder	1105
20.3	<code>mailcap</code> — Mailcap file handling	1114
20.4	<code>mailbox</code> — Manipulate mailboxes in various formats	1115
20.5	<code>mimetypes</code> — Map filenames to MIME types	1133
20.6	<code>base64</code> — Base16, Base32, Base64, Base85 Data Encodings	1136
20.7	<code>binhex</code> — Encode and decode binhex4 files	1139
20.8	<code>binascii</code> — Convert between binary and ASCII	1140
20.9	<code>quopri</code> — Encode and decode MIME quoted-printable data	1142
20.10	<code>uu</code> — Encode and decode uuencode files	1143
21	Structured Markup Processing Tools	1145
21.1	<code>html</code> — HyperText Markup Language support	1145
21.2	<code>html.parser</code> — Simple HTML and XHTML parser	1145
21.3	<code>html.entities</code> — Definitions of HTML general entities	1150
21.4	<code>XML Processing Modules</code>	1151
21.5	<code>xml.etree.ElementTree</code> — The ElementTree XML API	1152
21.6	<code>xml.dom</code> — The Document Object Model API	1171
21.7	<code>xml.dom.minidom</code> — Minimal DOM implementation	1182
21.8	<code>xml.dom.pulldom</code> — Support for building partial DOM trees	1186
21.9	<code>xml.sax</code> — Support for SAX2 parsers	1188
21.10	<code>xml.sax.handler</code> — Base classes for SAX handlers	1190
21.11	<code>xml.sax.saxutils</code> — SAX Utilities	1195
21.12	<code>xml.sax.xmlreader</code> — Interface for XML parsers	1196
21.13	<code>xml.parsers.expat</code> — Fast XML parsing using Expat	1200
22	Internet Protocols and Support	1211
22.1	<code>webbrowser</code> — Convenient Web-browser controller	1211
22.2	<code>cgi</code> — Common Gateway Interface support	1213
22.3	<code>cgitb</code> — Traceback manager for CGI scripts	1220
22.4	<code>wsgiref</code> — WSGI Utilities and Reference Implementation	1221
22.5	<code>urllib</code> — URL handling modules	1231
22.6	<code>urllib.request</code> — Extensible library for opening URLs	1231
22.7	<code>urllib.response</code> — Response classes used by urllib	1249
22.8	<code>urllib.parse</code> — Parse URLs into components	1250
22.9	<code>urllib.error</code> — Exception classes raised by urllib.request	1258
22.10	<code>urllib.robotparser</code> — Parser for robots.txt	1258
22.11	<code>http</code> — HTTP modules	1260
22.12	<code>http.client</code> — HTTP protocol client	1262
22.13	<code>ftplib</code> — FTP protocol client	1269
22.14	<code>poplib</code> — POP3 protocol client	1274
22.15	<code>imaplib</code> — IMAP4 protocol client	1277
22.16	<code>nntplib</code> — NNTP protocol client	1283
22.17	<code>smtplib</code> — SMTP protocol client	1290
22.18	<code>smtpd</code> — SMTP Server	1297

22.19	<code>telnetlib</code> — Telnet client	1300
22.20	<code>uuid</code> — UUID objects according to RFC 4122	1303
22.21	<code>socketserver</code> — A framework for network servers	1306
22.22	<code>http.server</code> — HTTP servers	1315
22.23	<code>http.cookies</code> — HTTP state management	1320
22.24	<code>http.cookiejar</code> — Cookie handling for HTTP clients	1324
22.25	<code>xmlrpc</code> — XMLRPC server and client modules	1332
22.26	<code>xmlrpc.client</code> — XML-RPC client access	1333
22.27	<code>xmlrpc.server</code> — Basic XML-RPC servers	1341
22.28	<code>ipaddress</code> — IPv4/IPv6 manipulation library	1347
23	Multimedia Services	1361
23.1	<code>audioop</code> — Manipulate raw audio data	1361
23.2	<code>aifc</code> — Read and write AIFF and AIFC files	1364
23.3	<code>sunau</code> — Read and write Sun AU files	1366
23.4	<code>wave</code> — Read and write WAV files	1369
23.5	<code>chunk</code> — Read IFF chunked data	1372
23.6	<code>colorsyst</code> — Conversions between color systems	1373
23.7	<code>imghdr</code> — Determine the type of an image	1374
23.8	<code>sndhdr</code> — Determine type of sound file	1375
23.9	<code>osaudiodev</code> — Access to OSS-compatible audio devices	1375
24	Internationalization	1381
24.1	<code>gettext</code> — Multilingual internationalization services	1381
24.2	<code>locale</code> — Internationalization services	1390
25	Program Frameworks	1399
25.1	<code>turtle</code> — Turtle graphics	1399
25.2	<code>cmd</code> — Support for line-oriented command interpreters	1434
25.3	<code>shlex</code> — Simple lexical analysis	1439
26	Graphical User Interfaces with Tk	1445
26.1	<code>tkinter</code> — Python interface to Tcl/Tk	1445
26.2	<code>tkinter.ttk</code> — Tk themed widgets	1456
26.3	<code>tkinter.tix</code> — Extension widgets for Tk	1474
26.4	<code>tkinter.scrolledtext</code> — Scrolled Text Widget	1479
26.5	<code>IDLE</code>	1479
26.6	Other Graphical User Interface Packages	1490
27	Development Tools	1493
27.1	<code>typing</code> — Support for type hints	1493
27.2	<code>pydoc</code> — Documentation generator and online help system	1513
27.3	<code>doctest</code> — Test interactive Python examples	1514
27.4	<code>unittest</code> — Unit testing framework	1537
27.5	<code>unittest.mock</code> — mock object library	1567
27.6	<code>unittest.mock</code> — getting started	1608
27.7	<code>2to3</code> - Automated Python 2 to 3 code translation	1628
27.8	<code>test</code> — Regression tests package for Python	1634
27.9	<code>test.support</code> — Utilities for the Python test suite	1636
27.10	<code>test.support.script_helper</code> — Utilities for the Python execution tests	1649
28	Debugging and Profiling	1651
28.1	<code>Audit</code> events table	1651
28.2	<code>bdb</code> — Debugger framework	1654
28.3	<code>faulthandler</code> — Dump the Python traceback	1659

28.4	<code>pdb</code> — The Python Debugger	1661
28.5	<code>The Python Profilers</code>	1667
28.6	<code>timeit</code> — Measure execution time of small code snippets	1676
28.7	<code>trace</code> — Trace or track Python statement execution	1681
28.8	<code>tracemalloc</code> — Trace memory allocations	1683
29	Software Packaging and Distribution	1695
29.1	<code>distutils</code> — Building and installing Python modules	1695
29.2	<code>ensurepip</code> — Bootstrapping the <code>pip</code> installer	1696
29.3	<code>venv</code> — Creation of virtual environments	1697
29.4	<code>zipapp</code> — Manage executable Python zip archives	1706
30	Python Runtime Services	1713
30.1	<code>sys</code> — System-specific parameters and functions	1713
30.2	<code>sysconfig</code> — Provide access to Python’s configuration information	1732
30.3	<code>builtins</code> — Built-in objects	1736
30.4	<code>__main__</code> — Top-level script environment	1736
30.5	<code>warnings</code> — Warning control	1737
30.6	<code>dataclasses</code> — Data Classes	1743
30.7	<code>contextlib</code> — Utilities for with-statement contexts	1751
30.8	<code>abc</code> — Abstract Base Classes	1764
30.9	<code>atexit</code> — Exit handlers	1768
30.10	<code>traceback</code> — Print or retrieve a stack traceback	1770
30.11	<code>__future__</code> — Future statement definitions	1776
30.12	<code>gc</code> — Garbage Collector interface	1777
30.13	<code>inspect</code> — Inspect live objects	1780
30.14	<code>site</code> — Site-specific configuration hook	1796
31	Custom Python Interpreters	1801
31.1	<code>code</code> — Interpreter base classes	1801
31.2	<code>codeop</code> — Compile Python code	1803
32	Importing Modules	1805
32.1	<code>zipimport</code> — Import modules from Zip archives	1805
32.2	<code>pkgutil</code> — Package extension utility	1807
32.3	<code>modulefinder</code> — Find modules used by a script	1810
32.4	<code>runpy</code> — Locating and executing Python modules	1811
32.5	<code>importlib</code> — The implementation of <code>import</code>	1813
32.6	Using <code>importlib.metadata</code>	1833
33	Python Language Services	1837
33.1	<code>parser</code> — Access Python parse trees	1837
33.2	<code>ast</code> — Abstract Syntax Trees	1841
33.3	<code>syntable</code> — Access to the compiler’s symbol tables	1847
33.4	<code>symbol</code> — Constants used with Python parse trees	1850
33.5	<code>token</code> — Constants used with Python parse trees	1850
33.6	<code>keyword</code> — Testing for Python keywords	1854
33.7	<code>tokenize</code> — Tokenizer for Python source	1854
33.8	<code>tabnanny</code> — Detection of ambiguous indentation	1858
33.9	<code>pyclbr</code> — Python module browser support	1859
33.10	<code>py_compile</code> — Compile Python source files	1860
33.11	<code>compileall</code> — Byte-compile Python libraries	1862
33.12	<code>dis</code> — Disassembler for Python bytecode	1865
33.13	<code>pickletools</code> — Tools for pickle developers	1879

34 Miscellaneous Services	1881
34.1 <code>formatter</code> — Generic output formatting	1881
35 MS Windows Specific Services	1885
35.1 <code>msilib</code> — Read and write Microsoft Installer files	1885
35.2 <code>msvcrt</code> — Useful routines from the MS VC++ runtime	1891
35.3 <code>winreg</code> — Windows registry access	1893
35.4 <code>winsound</code> — Sound-playing interface for Windows	1902
36 Unix Specific Services	1905
36.1 <code>posix</code> — The most common POSIX system calls	1905
36.2 <code>pwd</code> — The password database	1906
36.3 <code>spwd</code> — The shadow password database	1907
36.4 <code>grp</code> — The group database	1908
36.5 <code>crypt</code> — Function to check Unix passwords	1908
36.6 <code>termios</code> — POSIX style tty control	1910
36.7 <code>tty</code> — Terminal control functions	1912
36.8 <code>pty</code> — Pseudo-terminal utilities	1912
36.9 <code>fcntl</code> — The <code>fcntl</code> and <code>ioctl</code> system calls	1914
36.10 <code>pipes</code> — Interface to shell pipelines	1916
36.11 <code>resource</code> — Resource usage information	1917
36.12 <code>nis</code> — Interface to Sun's NIS (Yellow Pages)	1921
36.13 <code>syslog</code> — Unix syslog library routines	1922
37 Superseded Modules	1925
37.1 <code>optparse</code> — Parser for command line options	1925
37.2 <code>imp</code> — Access the import internals	1952
38 Undocumented Modules	1957
38.1 Platform specific modules	1957
A Glossary	1959
B About these documents	1973
B.1 Contributors to the Python Documentation	1973
C History and License	1975
C.1 History of the software	1975
C.2 Terms and conditions for accessing or otherwise using Python	1976
C.3 Licenses and Acknowledgements for Incorporated Software	1979
D Copyright	1993
Bibliography	1995
Python Module Index	1997
Index	2001

While `reference-index` describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions.

Python’s standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

The Python installers for the Windows platform usually include the entire standard library and often also include many additional components. For Unix-like operating systems Python is normally provided as a collection of packages, so it may be necessary to use the packaging tools provided with the operating system to obtain some or all of the optional components.

In addition to the standard library, there is a growing collection of several thousand components (from individual programs and modules to packages and entire application development frameworks), available from the [Python Package Index](#).

**CHAPTER
ONE**

INTRODUCTION

The “Python library” contains several different kinds of components.

It contains data types that would normally be considered part of the “core” of a language, such as numbers and lists. For these types, the Python language core defines the form of literals and places some constraints on their semantics, but does not fully define the semantics. (On the other hand, the language core does define syntactic properties like the spelling and priorities of operators.)

The library also contains built-in functions and exceptions — objects that can be used by all Python code without the need of an `import` statement. Some of these are defined by the core language, but many are not essential for the core semantics and are only described here.

The bulk of the library, however, consists of a collection of modules. There are many ways to dissect this collection. Some modules are written in C and built in to the Python interpreter; others are written in Python and imported in source form. Some modules provide interfaces that are highly specific to Python, like printing a stack trace; some provide interfaces that are specific to particular operating systems, such as access to specific hardware; others provide interfaces that are specific to a particular application domain, like the World Wide Web. Some modules are available in all versions and ports of Python; others are only available when the underlying system supports or requires them; yet others are available only when a particular configuration option was chosen at the time when Python was compiled and installed.

This manual is organized “from the inside out:” it first describes the built-in functions, data types and exceptions, and finally the modules, grouped in chapters of related modules.

This means that if you start reading this manual from the start, and skip to the next chapter when you get bored, you will get a reasonable overview of the available modules and application areas that are supported by the Python library. Of course, you don’t *have* to read it like a novel — you can also browse the table of contents (in front of the manual), or look for a specific function, module or term in the index (in the back). And finally, if you enjoy learning about random subjects, you choose a random page number (see module `random`) and read a section or two. Regardless of the order in which you read the sections of this manual, it helps to start with chapter *Built-in Functions*, as the remainder of the manual assumes familiarity with this material.

Let the show begin!

1.1 Notes on availability

- An “Availability: Unix” note means that this function is commonly found on Unix systems. It does not make any claims about its existence on a specific operating system.
- If not separately noted, all functions that claim “Availability: Unix” are supported on Mac OS X, which builds on a Unix core.

CHAPTER TWO

BUILT-IN FUNCTIONS

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

Built-in Functions				
<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	

`abs` (*x*)

Return the absolute value of a number. The argument may be an integer or a floating point number. If the argument is a complex number, its magnitude is returned. If *x* defines `__abs__()`, `abs(x)` returns *x*.`__abs__()`.

`all` (*iterable*)

Return True if all elements of the *iterable* are true (or if the iterable is empty). Equivalent to:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

`any` (*iterable*)

Return True if any element of the *iterable* is true. If the iterable is empty, return False. Equivalent to:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

`ascii(object)`

As `repr()`, return a string containing a printable representation of an object, but escape the non-ASCII characters in the string returned by `repr()` using `\x`, `\u` or `\U` escapes. This generates a string similar to that returned by `repr()` in Python 2.

`bin(x)`

Convert an integer number to a binary string prefixed with “0b”. The result is a valid Python expression. If `x` is not a Python `int` object, it has to define an `__index__()` method that returns an integer. Some examples:

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

If prefix “0b” is desired or not, you can use either of the following ways.

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

See also `format()` for more information.

`class bool([x])`

Return a Boolean value, i.e. one of `True` or `False`. `x` is converted using the standard *truth testing procedure*. If `x` is false or omitted, this returns `False`; otherwise it returns `True`. The `bool` class is a subclass of `int` (see *Numeric Types — int, float, complex*). It cannot be subclassed further. Its only instances are `False` and `True` (see *Boolean Values*).

Changed in version 3.7: `x` is now a positional-only parameter.

`breakpoint(*args, **kws)`

This function drops you into the debugger at the call site. Specifically, it calls `sys.breakpointhook()`, passing `args` and `kws` straight through. By default, `sys.breakpointhook()` calls `pdb.set_trace()` expecting no arguments. In this case, it is purely a convenience function so you don’t have to explicitly import `pdb` or type as much code to enter the debugger. However, `sys.breakpointhook()` can be set to some other function and `breakpoint()` will automatically call that, allowing you to drop into the debugger of choice.

Raises an *auditing event* `builtins.breakpoint` with argument `breakpointhook`.

New in version 3.7.

`class bytearray([source[, encoding[, errors]]])`

Return a new array of bytes. The `bytearray` class is a mutable sequence of integers in the range $0 \leq x < 256$. It has most of the usual methods of mutable sequences, described in *Mutable Sequence Types*, as well as most methods that the `bytes` type has, see *Bytes and Bytearray Operations*.

The optional `source` parameter can be used to initialize the array in a few different ways:

- If it is a `string`, you must also give the `encoding` (and optionally, `errors`) parameters; `bytearray()` then converts the string to bytes using `str.encode()`.
- If it is an `integer`, the array will have that size and will be initialized with null bytes.
- If it is an object conforming to the `buffer` interface, a read-only buffer of the object will be used to initialize the bytes array.
- If it is an `iterable`, it must be an iterable of integers in the range $0 \leq x < 256$, which are used as the initial contents of the array.

Without an argument, an array of size 0 is created.

See also [Binary Sequence Types — bytes, bytearray, memoryview](#) and [Bytearray Objects](#).

`class bytes ([source[, encoding[, errors]]])`

Return a new “bytes” object, which is an immutable sequence of integers in the range $0 \leq x < 256$. `bytes` is an immutable version of `bytearray` – it has the same non-mutating methods and the same indexing and slicing behavior.

Accordingly, constructor arguments are interpreted as for `bytearray()`.

Bytes objects can also be created with literals, see strings.

See also [Binary Sequence Types — bytes, bytearray, memoryview, Bytes Objects](#), and [Bytes and Bytearray Operations](#).

`callable (object)`

Return `True` if the `object` argument appears callable, `False` if not. If this returns `True`, it is still possible that a call fails, but if it is `False`, calling `object` will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a `__call__()` method.

New in version 3.2: This function was first removed in Python 3.0 and then brought back in Python 3.2.

`chr (i)`

Return the string representing a character whose Unicode code point is the integer `i`. For example, `chr(97)` returns the string '`a`', while `chr(8364)` returns the string '`€`'. This is the inverse of `ord()`.

The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16). `ValueError` will be raised if `i` is outside that range.

`@classmethod`

Transform a method into a class method.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C:
    @classmethod
    def f(cls, arg1, arg2, ...): ...
```

The `@classmethod` form is a function `decorator` – see function for details.

A class method can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see `staticmethod()`.

For more information on class methods, see types.

`compile (source, filename, mode=0, dont_inherit=False, optimize=-1)`

Compile the `source` into a code or AST object. Code objects can be executed by `exec()` or `eval()`. `source` can either be a normal string, a byte string, or an AST object. Refer to the `ast` module documentation for information on how to work with AST objects.

The `filename` argument should give the file from which the code was read; pass some recognizable value if it wasn't read from a file ('<string>' is commonly used).

The `mode` argument specifies what kind of code must be compiled; it can be '`exec`' if `source` consists of a sequence of statements, '`eval`' if it consists of a single expression, or '`single`' if it consists of a single interactive statement (in the latter case, expression statements that evaluate to something other than `None` will be printed).

The optional arguments `flags` and `dont_inherit` control which future statements affect the compilation of `source`. If neither is present (or both are zero) the code is compiled with those future statements that are in effect in the code that is calling `compile()`. If the `flags` argument is given and `dont_inherit` is not (or is zero) then the future

statements specified by the *flags* argument are used in addition to those that would be used anyway. If *dont_inherit* is a non-zero integer then the *flags* argument is it – the future statements in effect around the call to `compile` are ignored.

Future statements are specified by bits which can be bitwise ORed together to specify multiple statements. The bitfield required to specify a given feature can be found as the `compiler_flag` attribute on the `_Feature` instance in the `__future__` module.

The optional argument *flags* also controls whether the compiled source is allowed to contain top-level `await`, `async for` and `async with`. When the bit `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` is set, the return code object has `CO_COROUTINE` set in `co_code`, and can be interactively executed via `await eval(code_object)`.

The argument *optimize* specifies the optimization level of the compiler; the default value of `-1` selects the optimization level of the interpreter as given by `-O` options. Explicit levels are `0` (no optimization; `__debug__` is true), `1` (asserts are removed, `__debug__` is false) or `2` (docstrings are removed too).

This function raises `SyntaxError` if the compiled source is invalid, and `ValueError` if the source contains null bytes.

If you want to parse Python code into its AST representation, see `ast.parse()`.

Raises an *auditing event* `compile` with arguments `source` and `filename`. This event may also be raised by implicit compilation.

Note: When compiling a string with multi-line code in '`'single'`' or '`'eval'`' mode, input must be terminated by at least one newline character. This is to facilitate detection of incomplete and complete statements in the `code` module.

Warning: It is possible to crash the Python interpreter with a sufficiently large/complex string when compiling to an AST object due to stack depth limitations in Python's AST compiler.

Changed in version 3.2: Allowed use of Windows and Mac newlines. Also input in '`'exec'`' mode does not have to end in a newline anymore. Added the *optimize* parameter.

Changed in version 3.5: Previously, `TypeError` was raised when null bytes were encountered in `source`.

New in version 3.8: `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` can now be passed in `flags` to enable support for top-level `await`, `async for`, and `async with`.

```
class complex([real[, imag]])
```

Return a complex number with the value `real + imag*1j` or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including `complex`). If `imag` is omitted, it defaults to zero and the constructor serves as a numeric conversion like `int` and `float`. If both arguments are omitted, returns `0j`.

For a general Python object `x`, `complex(x)` delegates to `x.__complex__()`. If `__complex__()` is not defined then it falls back to `__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

Note: When converting from a string, the string must not contain whitespace around the central `+` or `-` operator. For example, `complex('1+2j')` is fine, but `complex('1 + 2j')` raises `ValueError`.

The complex type is described in [Numeric Types — `int`, `float`, `complex`](#).

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.8: Falls back to `__index__()` if `__complex__()` and `__float__()` are not defined.

`delattr(object, name)`

This is a relative of `setattr()`. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(x, 'foobar')` is equivalent to `del x foobar`.

```
class dict(**kwargs)
class dict(mapping, **kwargs)
class dict(iterable, **kwargs)
```

Create a new dictionary. The `dict` object is the dictionary class. See `dict` and *Mapping Types — dict* for documentation about this class.

For other containers see the built-in `list`, `set`, and `tuple` classes, as well as the `collections` module.

`dir([object])`

Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object.

If the object has a method named `__dir__()`, this method will be called and must return the list of attributes. This allows objects that implement a custom `__getattr__()` or `__getattribute__()` function to customize the way `dir()` reports their attributes.

If the object does not provide `__dir__()`, the function tries its best to gather information from the object's `__dict__` attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom `__getattr__()`.

The default `dir()` mechanism behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete, information:

- If the object is a module object, the list contains the names of the module's attributes.
- If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
- Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes.

The resulting list is sorted alphabetically. For example:

```
>>> import struct
>>> dir()    # show the names in the module namespace
['__builtins__', '__name__', 'struct']
>>> dir(struct)  # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '__clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
>>> s = Shape()
>>> dir(s)
['area', 'location', 'perimeter']
```

Note: Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its

detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

divmod(*a, b*)

Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division. With mixed operand types, the rules for binary arithmetic operators apply. For integers, the result is the same as (*a* // *b*, *a* % *b*). For floating point numbers the result is (*q*, *a* % *b*), where *q* is usually `math.floor(a / b)` but may be 1 less than that. In any case *q* * *b* + *a* % *b* is very close to *a*, if *a* % *b* is non-zero it has the same sign as *b*, and $0 \leq \text{abs}(a \% b) < \text{abs}(b)$.

enumerate(*iterable*, *start*=0)

Return an enumerate object. *iterable* must be a sequence, an [iterator](#), or some other object which supports iteration. The `__next__()` method of the iterator returned by `enumerate()` returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *iterable*.

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Equivalent to:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

eval(*expression*[, *globals*[, *locals*]])

The arguments are a string and optional *globals* and *locals*. If provided, *globals* must be a dictionary. If provided, *locals* can be any mapping object.

The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* dictionaries as global and local namespace. If the *globals* dictionary is present and does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key before *expression* is parsed. This means that *expression* normally has full access to the standard `builtins` module and restricted environments are propagated. If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed with the *globals* and *locals* in the environment where `eval()` is called. Note, `eval()` does not have access to the *nested scopes* (non-locals) in the enclosing environment.

The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```
>>> x = 1
>>> eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as those created by `compile()`). In this case pass a code object instead of a string. If the code object has been compiled with 'exec' as the *mode* argument, `eval()`'s return value will be `None`.

Hints: dynamic execution of statements is supported by the `exec()` function. The `globals()` and `locals()` functions returns the current global and local dictionary, respectively, which may be useful to pass around for use by `eval()` or `exec()`.

See `ast.literal_eval()` for a function that can safely evaluate strings with expressions containing only literals.

Raises an *auditing event* `exec` with the code object as the argument. Code compilation events may also be raised.

`exec(object[, globals[, locals]])`

This function supports dynamic execution of Python code. *object* must be either a string or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs).¹ If it is a code object, it is simply executed. In all cases, the code that's executed is expected to be valid as file input (see the section “File input” in the Reference Manual). Be aware that the `return` and `yield` statements may not be used outside of function definitions even within the context of code passed to the `exec()` function. The return value is `None`.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only *globals* is provided, it must be a dictionary (and not a subclass of dictionary), which will be used for both the global and the local variables. If *globals* and *locals* are given, they are used for the global and local variables, respectively. If provided, *locals* can be any mapping object. Remember that at module level, *globals* and *locals* are the same dictionary. If `exec` gets two separate objects as *globals* and *locals*, the code will be executed as if it were embedded in a class definition.

If the *globals* dictionary does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key. That way you can control what builtins are available to the executed code by inserting your own `__builtins__` dictionary into *globals* before passing it to `exec()`.

Raises an *auditing event* `exec` with the code object as the argument. Code compilation events may also be raised.

Note: The built-in functions `globals()` and `locals()` return the current global and local dictionary, respectively, which may be useful to pass around for use as the second and third argument to `exec()`.

Note: The default *locals* act as described for function `locals()` below: modifications to the default *locals* dictionary should not be attempted. Pass an explicit *locals* dictionary if you need to see effects of the code on *locals* after function `exec()` returns.

`filter(function, iterable)`

Construct an iterator from those elements of *iterable* for which *function* returns true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *function* is `None`, the identity function is assumed, that is, all elements of *iterable* that are false are removed.

Note that `filter(function, iterable)` is equivalent to the generator expression `(item for item in iterable if function(item))` if *function* is not `None` and `(item for item in iterable if item)` if *function* is `None`.

See `itertools.filterfalse()` for the complementary function that returns elements of *iterable* for which *function* returns false.

`class float([x])`

Return a floating point number constructed from a number or string *x*.

If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be '+' or '-'; a '+' sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or a positive or negative infinity. More precisely, the input must conform to the following grammar after leading and trailing whitespace characters are removed:

```
sign ::= "+" | "-"
```

¹ Note that the parser only accepts the Unix-style end of line convention. If you are reading the code from a file, make sure to use newline conversion mode to convert Windows or Mac-style newlines.

```

infinity      ::=  "Infinity" | "inf"
nan          ::=  "nan"
numeric_value ::=  floatnumber | infinity | nan
numeric_string ::=  [sign] numeric_value

```

Here `floatnumber` is the form of a Python floating-point literal, described in [floating](#). Case is not significant, so, for example, “`inf`”, “`Inf`”, “`INFINITY`” and “`iNfINity`” are all acceptable spellings for positive infinity.

Otherwise, if the argument is an integer or a floating point number, a floating point number with the same value (within Python’s floating point precision) is returned. If the argument is outside the range of a Python float, an [`OverflowError`](#) will be raised.

For a general Python object `x`, `float(x)` delegates to `x.__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

If no argument is given, `0.0` is returned.

Examples:

```

>>> float('+1.23')
1.23
>>> float(' -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf

```

The float type is described in [Numeric Types — `int`, `float`, `complex`](#).

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.7: `x` is now a positional-only parameter.

Changed in version 3.8: Falls back to `__index__()` if `__float__()` is not defined.

`format(value[, format_spec])`

Convert a `value` to a “formatted” representation, as controlled by `format_spec`. The interpretation of `format_spec` will depend on the type of the `value` argument, however there is a standard formatting syntax that is used by most built-in types: [Format Specification Mini-Language](#).

The default `format_spec` is an empty string which usually gives the same effect as calling `str(value)`.

A call to `format(value, format_spec)` is translated to `type(value).__format__(value, format_spec)` which bypasses the instance dictionary when searching for the value’s `__format__()` method. A [`TypeError`](#) exception is raised if the method search reaches `object` and the `format_spec` is non-empty, or if either the `format_spec` or the return value are not strings.

Changed in version 3.4: `object().__format__(format_spec)` raises [`TypeError`](#) if `format_spec` is not an empty string.

`class frozenset([iterable])`

Return a new `frozenset` object, optionally with elements taken from `iterable`. `frozenset` is a built-in class. See `frozenset` and [Set Types — `set`, `frozenset`](#) for documentation about this class.

For other containers see the built-in `set`, `list`, `tuple`, and `dict` classes, as well as the `collections` module.

`getattr(object, name[, default])`

Return the value of the named attribute of `object`. `name` must be a string. If the string is the name of one of the ob-

ject's attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribute does not exist, `default` is returned if provided, otherwise `AttributeError` is raised.

`globals()`

Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

`hasattr(object, name)`

The arguments are an object and a string. The result is `True` if the string is the name of one of the object's attributes, `False` if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an `AttributeError` or not.)

`hash(object)`

Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

Note: For objects with custom `__hash__()` methods, note that `hash()` truncates the return value based on the bit width of the host machine. See `__hash__()` for details.

`help([object])`

Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated.

Note that if a slash(/) appears in the parameter list of a function, when invoking `help()`, it means that the parameters prior to the slash are positional-only. For more info, see the FAQ entry on positional-only parameters.

This function is added to the built-in namespace by the `site` module.

Changed in version 3.4: Changes to `pydoc` and `inspect` mean that the reported signatures for callables are now more comprehensive and consistent.

`hex(x)`

Convert an integer number to a lowercase hexadecimal string prefixed with "0x". If `x` is not a Python `int` object, it has to define an `__index__()` method that returns an integer. Some examples:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

If you want to convert an integer number to an uppercase or lower hexadecimal string with prefix or not, you can use either of the following ways:

```
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
>>> f'{255:#x}', f'{255:x}', f'{255:X}'
('0xff', 'ff', 'FF')
```

See also `format()` for more information.

See also `int()` for converting a hexadecimal string to an integer using a base of 16.

Note: To obtain a hexadecimal string representation for a float, use the `float.hex()` method.

`id`(*object*)

Return the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.

CPython implementation detail: This is the address of the object in memory.

`input`(*[prompt]*)

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, `EOFError` is raised. Example:

```
>>> s = input('---> ')
---> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the `readline` module was loaded, then `input()` will use it to provide elaborate line editing and history features.

Raises an *auditing event* `builtins.input` with argument *prompt* before reading input

Raises an auditing event `builtins.input/result` with the result after successfully reading input.

`class int([x])`**`class int(x, base=10)`**

Return an integer object constructed from a number or string *x*, or return 0 if no arguments are given. If *x* defines `__int__()`, `int(x)` returns *x*.`__int__()`. If *x* defines `__index__()`, it returns *x*.`__index__()`. If *x* defines `__trunc__()`, it returns *x*.`__trunc__()`. For floating point numbers, this truncates towards zero.

If *x* is not a number or if *base* is given, then *x* must be a string, `bytes`, or `bytearray` instance representing an integer literal in radix *base*. Optionally, the literal can be preceded by + or - (with no space in between) and surrounded by whitespace. A base-n literal consists of the digits 0 to n-1, with a to z (or A to Z) having values 10 to 35. The default *base* is 10. The allowed values are 0 and 2–36. Base-2, -8, and -16 literals can be optionally prefixed with 0b/0B, 0o/0O, or 0x/0X, as with integer literals in code. Base 0 means to interpret exactly as a code literal, so that the actual base is 2, 8, 10, or 16, and so that `int('010', 0)` is not legal, while `int('010')` is, as well as `int('010', 8)`.

The integer type is described in *Numeric Types — int, float, complex*.

Changed in version 3.4: If *base* is not an instance of `int` and the *base* object has a `base.__index__` method, that method is called to obtain an integer for the base. Previous versions used `base.__int__` instead of `base.__index__`.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.7: *x* is now a positional-only parameter.

Changed in version 3.8: Falls back to `__index__()` if `__int__()` is not defined.

`isinstance`(*object, classinfo*)

Return True if the *object* argument is an instance of the *classinfo* argument, or of a (direct, indirect or *virtual*) subclass thereof. If *object* is not an object of the given type, the function always returns False. If *classinfo* is a tuple of type objects (or recursively, other such tuples), return True if *object* is an instance of any of the types. If *classinfo* is not a type or tuple of types and such tuples, a `TypeError` exception is raised.

`issubclass`(*class, classinfo*)

Return True if *class* is a subclass (direct, indirect or *virtual*) of *classinfo*. A class is considered a subclass of itself.

classinfo may be a tuple of class objects, in which case every entry in *classinfo* will be checked. In any other case, a *TypeError* exception is raised.

`iter(object[, sentinel])`

Return an *iterator* object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, *object* must be a collection object which supports the iteration protocol (the `__iter__()` method), or it must support the sequence protocol (the `__getitem__()` method with integer arguments starting at 0). If it does not support either of those protocols, *TypeError* is raised. If the second argument, *sentinel*, is given, then *object* must be a callable object. The iterator created in this case will call *object* with no arguments for each call to its `__next__()` method; if the value returned is equal to *sentinel*, *StopIteration* will be raised, otherwise the value will be returned.

See also *Iterator Types*.

One useful application of the second form of `iter()` is to build a block-reader. For example, reading fixed-width blocks from a binary database file until the end of file is reached:

```
from functools import partial
with open('mydata.db', 'rb') as f:
    for block in iter(partial(f.read, 64), b''):
        process_block(block)
```

`len(s)`

Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

`class list([iterable])`

Rather than being a function, `list` is actually a mutable sequence type, as documented in *Lists and Sequence Types — list, tuple, range*.

`locals()`

Update and return a dictionary representing the current local symbol table. Free variables are returned by `locals()` when it is called in function blocks, but not in class blocks. Note that at the module level, `locals()` and `globals()` are the same dictionary.

Note: The contents of this dictionary should not be modified; changes may not affect the values of local and free variables used by the interpreter.

`map(function, iterable, ...)`

Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases where the function inputs are already arranged into argument tuples, see `itertools.starmap()`.

`max(iterable, *[, key, default])`

`max(arg1, arg2, *args[, key])`

Return the largest item in an iterable or the largest of two or more arguments.

If one positional argument is provided, it should be an *iterable*. The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for `list.sort()`. The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a *ValueError* is raised.

If multiple items are maximal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc, reverse=True)[0]` and `heapq.nlargest(1, iterable, key=keyfunc)`.

New in version 3.4: The *default* keyword-only argument.

Changed in version 3.8: The *key* can be None.

class memoryview(*obj*)

Return a “memory view” object created from the given argument. See [Memory Views](#) for more information.

min(*iterable*, *[, *key*, *default*])

min(*arg1*, *arg2*, **args*[, *key*])

Return the smallest item in an iterable or the smallest of two or more arguments.

If one positional argument is provided, it should be an *iterable*. The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for [*list.sort\(\)*](#). The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a [ValueError](#) is raised.

If multiple items are minimal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as [*sorted\(iterable, key=keyfunc\)\[0\]*](#) and [*heapq.nsmallest\(1, iterable, key=keyfunc\)*](#).

New in version 3.4: The *default* keyword-only argument.

Changed in version 3.8: The *key* can be None.

next(*iterator*[, *default*])

Retrieve the next item from the *iterator* by calling its [*__next__\(\)*](#) method. If *default* is given, it is returned if the iterator is exhausted, otherwise [StopIteration](#) is raised.

class object

Return a new featureless object. [*object*](#) is a base for all classes. It has the methods that are common to all instances of Python classes. This function does not accept any arguments.

Note: [*object*](#) does *not* have a [*__dict__*](#), so you can't assign arbitrary attributes to an instance of the [*object*](#) class.

oct(*x*)

Convert an integer number to an octal string prefixed with “0o”. The result is a valid Python expression. If *x* is not a Python [*int*](#) object, it has to define an [*__index__\(\)*](#) method that returns an integer. For example:

```
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'
```

If you want to convert an integer number to octal string either with prefix “0o” or not, you can use either of the following ways.

```
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
>>> f'{10:#o}', f'{10:o}'
('0o12', '12')
```

See also [*format\(\)*](#) for more information.

open (*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

Open *file* and return a corresponding *file object*. If the file cannot be opened, an *OSError* is raised.

file is a *path-like object* giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless *closefd* is set to *False*.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to '*r*' which means open for reading in text mode. Other common values are '*w*' for writing (truncating the file if it already exists), '*x*' for exclusive creation and '*a*' for appending (which on *some* Unix systems, means that *all* writes append to the end of the file regardless of the current seek position). In text mode, if *encoding* is not specified the encoding used is platform dependent: `locale.getpreferredencoding(False)` is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave *encoding* unspecified.) The available modes are:

Character	Meaning
' <i>r</i> '	open for reading (default)
' <i>w</i> '	open for writing, truncating the file first
' <i>x</i> '	open for exclusive creation, failing if the file already exists
' <i>a</i> '	open for writing, appending to the end of the file if it exists
' <i>b</i> '	binary mode
' <i>t</i> '	text mode (default)
'+'	open for updating (reading and writing)

The default mode is '*r*' (open for reading text, synonym of '*rt*'). Modes '*w+*' and '*w+b*' open and truncate the file. Modes '*r+*' and '*r+b*' open the file with no truncation.

As mentioned in the *Overview*, Python distinguishes between binary and text I/O. Files opened in binary mode (including '*b*' in the *mode* argument) return contents as *bytes* objects without any decoding. In text mode (the default, or when '*t*' is included in the *mode* argument), the contents of the file are returned as *str*, the bytes having been first decoded using a platform-dependent encoding or using the specified *encoding* if given.

There is an additional mode character permitted, '*U*', which no longer has any effect, and is considered deprecated. It previously enabled *universal newlines* in text mode, which became the default behaviour in Python 3.0. Refer to the documentation of the *newline* parameter for further details.

Note: Python doesn't depend on the underlying operating system's notion of text files; all the processing is done by Python itself, and is therefore platform-independent.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size in bytes of a fixed-size chunk buffer. When no *buffering* argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on `io.DEFAULT_BUFFER_SIZE`. On many systems, the buffer will typically be 4096 or 8192 bytes long.
- "Interactive" text files (files for which `isatty()` returns `True`) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever `locale.getpreferredencoding()` returns), but any *text encoding* supported by Python can be used. See the `codecs` module for the list of supported encodings.

errors is an optional string that specifies how encoding and decoding errors are to be handled—this cannot be used in binary mode. A variety of standard error handlers are available (listed under *Error Handlers*), though any error

handling name that has been registered with `codecs.register_error()` is also valid. The standard names include:

- 'strict' to raise a `ValueError` exception if there is an encoding error. The default value of `None` has the same effect.
- 'ignore' ignores errors. Note that ignoring encoding errors can lead to data loss.
- 'replace' causes a replacement marker (such as '?) to be inserted where there is malformed data.
- 'surrogateescape' will represent any incorrect bytes as code points in the Unicode Private Use Area ranging from U+DC80 to U+DCFF. These private code points will then be turned back into the same bytes when the `surrogateescape` error handler is used when writing data. This is useful for processing files in an unknown encoding.
- 'xmlcharrefreplace' is only supported when writing to a file. Characters not supported by the encoding are replaced with the appropriate XML character reference &#nnn;.
- 'backslashreplace' replaces malformed data by Python's backslashed escape sequences.
- 'namereplace' (also only supported when writing) replaces unsupported characters with \N{...} escape sequences.

`newline` controls how *universal newlines* mode works (it only applies to text mode). It can be `None`, '`'`', '`\n`', '`\r`', and '`\r\n`'. It works as follows:

- When reading input from the stream, if `newline` is `None`, universal newlines mode is enabled. Lines in the input can end in '`\n`', '`\r`', or '`\r\n`', and these are translated into '`\n`' before being returned to the caller. If it is '`'`', universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- When writing output to the stream, if `newline` is `None`, any '`\n`' characters written are translated to the system default line separator, `os.linesep`. If `newline` is '`'` or '`\n`', no translation takes place. If `newline` is any of the other legal values, any '`\n`' characters written are translated to the given string.

If `closefd` is `False` and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given `closefd` must be `True` (the default) otherwise an error will be raised.

A custom opener can be used by passing a callable as `opener`. The underlying file descriptor for the file object is then obtained by calling `opener` with `(file, flags)`. `opener` must return an open file descriptor (passing `os.open` as `opener` results in functionality similar to passing `None`).

The newly created file is *non-inheritable*.

The following example uses the `dir_fd` parameter of the `os.open()` function to open a file relative to a given directory:

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spamspam.txt', file=f)
...
>>> os.close(dir_fd) # don't leak a file descriptor
```

The type of `file object` returned by the `open()` function depends on the mode. When `open()` is used to open a file in a text mode ('`w`', '`r`', '`wt`', '`rt`', etc.), it returns a subclass of `io.TextIOWrapper` (specifically `io.TextIOWrapper`). When used to open a file in a binary mode with buffering, the returned class is a subclass of

`io.BufferedIOBase`. The exact class varies: in read binary mode, it returns an `io.BufferedReader`; in write binary and append binary modes, it returns an `io.BufferedWriter`, and in read/write mode, it returns an `io.BufferedRandom`. When buffering is disabled, the raw stream, a subclass of `io.RawIOBase`, `io.FileIO`, is returned.

See also the file handling modules, such as, `fileinput`, `io` (where `open()` is declared), `os`, `os.path`, `tempfile`, and `shutil`.

Raises an *auditing event* open with arguments `file`, `mode`, `flags`.

The `mode` and `flags` arguments may have been modified or inferred from the original call.

Changed in version 3.3:

- The `opener` parameter was added.
- The '`x`' mode was added.
- `IOError` used to be raised, it is now an alias of `OSError`.
- `FileExistsError` is now raised if the file opened in exclusive creation mode ('`x`') already exists.

Changed in version 3.4:

- The file is now non-inheritable.

Deprecated since version 3.4, will be removed in version 3.9: The '`U`' mode.

Changed in version 3.5:

- If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see [PEP 475](#) for the rationale).
- The '`namereplace`' error handler was added.

Changed in version 3.6:

- Support added to accept objects implementing `os.PathLike`.
- On Windows, opening a console buffer may return a subclass of `io.RawIOBase` other than `io.FileIO`.

`ord(c)`

Given a string representing one Unicode character, return an integer representing the Unicode code point of that character. For example, `ord('a')` returns the integer 97 and `ord('€')` (Euro sign) returns 8364. This is the inverse of `chr()`.

`pow(base, exp[, mod])`

Return `base` to the power `exp`; if `mod` is present, return `base` to the power `exp`, modulo `mod` (computed more efficiently than `pow(base, exp) % mod`). The two-argument form `pow(base, exp)` is equivalent to using the power operator: `base**exp`.

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For `int` operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns 100, but `10**-2` returns 0.01.

For `int` operands `base` and `exp`, if `mod` is present, `mod` must also be of integer type and `mod` must be nonzero. If `mod` is present and `exp` is negative, `base` must be relatively prime to `mod`. In that case, `pow(inv_base, -exp, mod)` is returned, where `inv_base` is an inverse to `base` modulo `mod`.

Here's an example of computing an inverse for 38 modulo 97:

```
>>> pow(38, -1, mod=97)
23
>>> 23 * 38 % 97 == 1
True
```

Changed in version 3.8: For `int` operands, the three-argument form of `pow` now allows the second argument to be negative, permitting computation of modular inverses.

Changed in version 3.9: Allow keyword arguments. Formerly, only positional arguments were supported.

`print(*objects, sep='', end='\n', file=sys.stdout, flush=False)`

Print `objects` to the text stream `file`, separated by `sep` and followed by `end`. `sep`, `end`, `file` and `flush`, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by `sep` and followed by `end`. Both `sep` and `end` must be strings; they can also be `None`, which means to use the default values. If no `objects` are given, `print()` will just write `end`.

The `file` argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Whether output is buffered is usually determined by `file`, but if the `flush` keyword argument is true, the stream is forcibly flushed.

Changed in version 3.3: Added the `flush` keyword argument.

`class property(fget=None, fset=None, fdel=None, doc=None)`

Return a property attribute.

`fget` is a function for getting an attribute value. `fset` is a function for setting an attribute value. `fdel` is a function for deleting an attribute value. And `doc` creates a docstring for the attribute.

A typical use is to define a managed attribute `x`:

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

x = property(getx, setx, delx, "I'm the 'x' property.")
```

If `c` is an instance of `C`, `c.x` will invoke the getter, `c.x = value` will invoke the setter and `del c.x` the deleter.

If given, `doc` will be the docstring of the property attribute. Otherwise, the property will copy `fget`'s docstring (if it exists). This makes it possible to create read-only properties easily using `property()` as a `decorator`:

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

(continues on next page)

(continued from previous page)

```

self._voltage = 100000

@property
def voltage(self):
    """Get the current voltage."""
    return self._voltage

```

The `@property` decorator turns the `voltage()` method into a “getter” for a read-only attribute with the same name, and it sets the docstring for `voltage` to “Get the current voltage.”

A property object has `getter`, `setter`, and `deleter` methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```

class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x

```

This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original property (`x` in this case.)

The returned property object also has the attributes `fget`, `fset`, and `fdel` corresponding to the constructor arguments.

Changed in version 3.5: The docstrings of property objects are now writeable.

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

Rather than being a function, `range` is actually an immutable sequence type, as documented in [Ranges and Sequence Types — list, tuple, range](#).

`repr(object)`

Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`, otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a `__repr__()` method.

`reversed(seq)`

Return a reverse [iterator](#). `seq` must be an object which has a `__reversed__()` method or supports the sequence protocol (the `__len__()` method and the `__getitem__(i)` method with integer arguments starting at 0).

`round(number[, ndigits])`

Return `number` rounded to `ndigits` precision after the decimal point. If `ndigits` is omitted or is `None`, it returns the nearest integer to its input.

For the built-in types supporting `round()`, values are rounded to the closest multiple of 10 to the power minus `ndigits`; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are 0, and `round(1.5)` is 2). Any integer value is valid for `ndigits` (positive, zero, or negative). The return value is an integer if `ndigits` is omitted or `None`. Otherwise the return value has the same type as `number`.

For a general Python object `number`, `round` delegates to `number.__round__`.

Note: The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives 2.67 instead of the expected 2.68. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See [tut-fp-issues](#) for more information.

class `set`([*iterable*])

Return a new `set` object, optionally with elements taken from *iterable*. `set` is a built-in class. See [set](#) and [Set Types — `set`, `frozenset`](#) for documentation about this class.

For other containers see the built-in `frozenset`, `list`, `tuple`, and `dict` classes, as well as the `collections` module.

`setattr`(*object*, *name*, *value*)

This is the counterpart of `getattr()`. The arguments are an object, a string and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

class `slice`(*stop*)

class `slice`(*start*, *stop*[, *step*])

Return a `slice` object representing the set of indices specified by range (`start`, `stop`, `step`). The `start` and `step` arguments default to `None`. Slice objects have read-only data attributes `start`, `stop` and `step` which merely return the argument values (or their default). They have no other explicit functionality; however they are used by Numerical Python and other third party extensions. Slice objects are also generated when extended indexing syntax is used. For example: `a[start:stop:step]` or `a[start:stop, i]`. See [itertools.islice\(\)](#) for an alternate version that returns an iterator.

`sorted`(*iterable*, *, *key=None*, *reverse=False*)

Return a new sorted list from the items in *iterable*.

Has two optional arguments which must be specified as keyword arguments.

key specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

Use `functools.cmp_to_key()` to convert an old-style `cmp` function to a `key` function.

The built-in `sorted()` function is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

For sorting examples and a brief sorting tutorial, see [sortinghowto](#).

`@staticmethod`

Transform a method into a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...
```

The `@staticmethod` form is a function *decorator* – see function for details.

A static method can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`).

Static methods in Python are similar to those found in Java or C++. Also see `classmethod()` for a variant that is useful for creating alternate class constructors.

Like all decorators, it is also possible to call `staticmethod` as a regular function and do something with its result. This is needed in some cases where you need a reference to a function from a class body and you want to avoid the automatic transformation to instance method. For these cases, use this idiom:

```
class C:
    builtin_open = staticmethod(open)
```

For more information on static methods, see types.

```
class str(object=’’)
class str(object=b”, encoding=’utf-8’, errors=’strict’)
    Return a str version of object. See str() for details.
```

`str` is the built-in string `class`. For general information about strings, see *Text Sequence Type — str*.

`sum(iterable, /, start=0)`

Sums `start` and the items of an `iterable` from left to right and returns the total. The `iterable`'s items are normally numbers, and the `start` value is not allowed to be a string.

For some use cases, there are good alternatives to `sum()`. The preferred, fast way to concatenate a sequence of strings is by calling `''.join(sequence)`. To add floating point values with extended precision, see `math.fsum()`. To concatenate a series of iterables, consider using `itertools.chain()`.

Changed in version 3.8: The `start` parameter can be specified as a keyword argument.

`super([type[, object-or-type]])`

Return a proxy object that delegates method calls to a parent or sibling class of `type`. This is useful for accessing inherited methods that have been overridden in a class.

The `object-or-type` determines the *method resolution order* to be searched. The search starts from the class right after the `type`.

For example, if `__mro__` of `object-or-type` is `D -> B -> C -> A -> object` and the value of `type` is `B`, then `super()` searches `C -> A -> object`.

The `__mro__` attribute of the `object-or-type` lists the method resolution search order used by both `getattr()` and `super()`. The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

If the second argument is omitted, the super object returned is unbound. If the second argument is an object, `isinstance(obj, type)` must be true. If the second argument is a type, `issubclass(type2, type)` must be true (this is useful for classmethods).

There are two typical use cases for `super`. In a class hierarchy with single inheritance, `super` can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of `super` in other programming languages.

The second use case is to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement “diamond diagrams” where multiple base classes implement the same method. Good design dictates that this method have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

For both use cases, a typical superclass call looks like this:

```
class C(B):
    def method(self, arg):
        super().method(arg)      # This does the same thing as:
                                # super(C, self).method(arg)
```

In addition to method lookups, `super()` also works for attribute lookups. One possible use case for this is calling [descriptors](#) in a parent or sibling class.

Note that `super()` is implemented as part of the binding process for explicit dotted attribute lookups such as `super().__getitem__(name)`. It does so by implementing its own `__getattribute__()` method for searching classes in a predictable order that supports cooperative multiple inheritance. Accordingly, `super()` is undefined for implicit lookups using statements or operators such as `super()[name]`.

Also note that, aside from the zero argument form, `super()` is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references. The zero argument form only works inside a class definition, as the compiler fills in the necessary details to correctly retrieve the class being defined, as well as accessing the current instance for ordinary methods.

For practical suggestions on how to design cooperative classes using `super()`, see [guide to using super\(\)](#).

`class tuple([iterable])`

Rather than being a function, `tuple` is actually an immutable sequence type, as documented in [Tuples and Sequence Types — list, tuple, range](#).

`class type(object)`

`class type(name, bases, dict)`

With one argument, return the type of an `object`. The return value is a type object and generally the same object as returned by `object.__class__`.

The `isinstance()` built-in function is recommended for testing the type of an object, because it takes subclasses into account.

With three arguments, return a new type object. This is essentially a dynamic form of the `class` statement. The `name` string is the class name and becomes the `__name__` attribute; the `bases` tuple itemizes the base classes and becomes the `__bases__` attribute; and the `dict` dictionary is the namespace containing definitions for class body and is copied to a standard dictionary to become the `__dict__` attribute. For example, the following two statements create identical `type` objects:

```
>>> class X:
...     a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

See also [Type Objects](#).

Changed in version 3.6: Subclasses of `type` which don't override `type.__new__` may no longer use the one-argument form to get the type of an object.

`vars([object])`

Return the `__dict__` attribute for a module, class, instance, or any other object with a `__dict__` attribute.

Objects such as modules and instances have an updateable `__dict__` attribute; however, other objects may have write restrictions on their `__dict__` attributes (for example, classes use a `types.MappingProxyType` to prevent direct dictionary updates).

Without an argument, `vars()` acts like `locals()`. Note, the locals dictionary is only useful for reads since updates to the locals dictionary are ignored.

`zip(*iterables)`

Make an iterator that aggregates elements from each of the iterables.

Returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator. Equivalent to:

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using `zip(*[iter(s)]*n)`. This repeats the *same* iterator *n* times so that each output tuple has the result of *n* calls to the iterator. This has the effect of dividing the input into *n*-length chunks.

`zip()` should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables. If those values are important, use `itertools.zip_longest()` instead.

`zip()` in conjunction with the `*` operator can be used to unzip a list:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True
```

`__import__(name, globals=None, locals=None, fromlist=(), level=0)`

Note: This is an advanced function that is not needed in everyday Python programming, unlike `importlib.import_module()`.

This function is invoked by the `import` statement. It can be replaced (by importing the `builtins` module and assigning to `builtins.__import__()`) in order to change semantics of the `import` statement, but doing so is **strongly** discouraged as it is usually simpler to use import hooks (see [PEP 302](#)) to attain the same goals and does not cause issues with code which assumes the default import implementation is in use. Direct use of `__import__()` is also discouraged in favor of `importlib.import_module()`.

The function imports the module *name*, potentially using the given *globals* and *locals* to determine how to interpret the name in a package context. The *fromlist* gives the names of objects or submodules that should be imported from the module given by *name*. The standard implementation does not use its *locals* argument at all, and uses its *globals* only to determine the package context of the `import` statement.

level specifies whether to use absolute or relative imports. 0 (the default) means only perform absolute imports. Positive values for *level* indicate the number of parent directories to search relative to the directory of the module calling `__import__()` (see [PEP 328](#) for the details).

When the *name* variable is of the form `package.module`, normally, the top-level package (the name up till the

first dot) is returned, *not* the module named by *name*. However, when a non-empty *fromlist* argument is given, the module named by *name* is returned.

For example, the statement `import spam` results in bytecode resembling the following code:

```
spam = __import__('spam', globals(), locals(), [], 0)
```

The statement `import spam.ham` results in this call:

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

Note how `__import__()` returns the toplevel module here because this is the object that is bound to a name by the `import` statement.

On the other hand, the statement `from spam.ham import eggs, sausage as saus` results in

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], 0)
eggs = _temp.eggs
saus = _temp.sausage
```

Here, the `spam.ham` module is returned from `__import__()`. From this object, the names to import are retrieved and assigned to their respective names.

If you simply want to import a module (potentially within a package) by name, use `importlib.import_module()`.

Changed in version 3.3: Negative values for *level* are no longer supported (which also changes the default value to 0).

BUILT-IN CONSTANTS

A small number of constants live in the built-in namespace. They are:

False

The false value of the `bool` type. Assignments to `False` are illegal and raise a `SyntaxError`.

True

The true value of the `bool` type. Assignments to `True` are illegal and raise a `SyntaxError`.

None

The sole value of the type `NoneType`. `None` is frequently used to represent the absence of a value, as when default arguments are not passed to a function. Assignments to `None` are illegal and raise a `SyntaxError`.

NotImplemented

Special value which should be returned by the binary special methods (e.g. `__eq__()`, `__lt__()`, `__add__()`, `__rsub__()`, etc.) to indicate that the operation is not implemented with respect to the other type; may be returned by the in-place binary special methods (e.g. `__imul__()`, `__iand__()`, etc.) for the same purpose. Its truth value is true.

Note: When a binary (or in-place) method returns `NotImplemented` the interpreter will try the reflected operation on the other type (or some other fallback, depending on the operator). If all attempts return `NotImplemented`, the interpreter will raise an appropriate exception. Incorrectly returning `NotImplemented` will result in a misleading error message or the `NotImplemented` value being returned to Python code.

See [Implementing the arithmetic operations](#) for examples.

Note: `NotImplementedError` and `NotImplemented` are not interchangeable, even though they have similar names and purposes. See [NotImplementedError](#) for details on when to use it.

Ellipsis

The same as the ellipsis literal “`...`”. Special value used mostly in conjunction with extended slicing syntax for user-defined container data types.

`__debug__`

This constant is true if Python was not started with an `-O` option. See also the `assert` statement.

Note: The names `None`, `False`, `True` and `__debug__` cannot be reassigned (assignments to them, even as an attribute name, raise `SyntaxError`), so they can be considered “true” constants.

3.1 Constants added by the `site` module

The `site` module (which is imported automatically during startup, except if the `-S` command-line option is given) adds several constants to the built-in namespace. They are useful for the interactive interpreter shell and should not be used in programs.

`quit` (`code=None`)

`exit` (`code=None`)

Objects that when printed, print a message like “Use `quit()` or Ctrl-D (i.e. EOF) to exit”, and when called, raise `SystemExit` with the specified exit code.

`copyright`

`credits`

Objects that when printed or called, print the text of copyright or credits, respectively.

`license`

Object that when printed, prints the message “Type `license()` to see the full license text”, and when called, displays the full license text in a pager-like fashion (one screen at a time).

BUILT-IN TYPES

The following sections describe the standard types that are built into the interpreter.

The principal built-in types are numerics, sequences, mappings, classes, instances and exceptions.

Some collection classes are mutable. The methods that add, subtract, or rearrange their members in place, and don't return a specific item, never return the collection instance itself but `None`.

Some operations are supported by several object types; in particular, practically all objects can be compared for equality, tested for truth value, and converted to a string (with the `repr()` function or the slightly different `str()` function). The latter function is implicitly used when an object is written by the `print()` function.

4.1 Truth Value Testing

Any object can be tested for truth value, for use in an `if` or `while` condition or as operand of the Boolean operations below.

By default, an object is considered true unless its class defines either a `__bool__()` method that returns `False` or a `__len__()` method that returns zero, when called with the object.¹ Here are most of the built-in objects considered false:

- constants defined to be false: `None` and `False`.
- zero of any numeric type: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- empty sequences and collections: `' '`, `()`, `[]`, `{}`, `set()`, `range(0)`

Operations and built-in functions that have a Boolean result always return `0` or `False` for false and `1` or `True` for true, unless otherwise stated. (Important exception: the Boolean operations `or` and `and` always return one of their operands.)

4.2 Boolean Operations — `and`, `or`, `not`

These are the Boolean operations, ordered by ascending priority:

Operation	Result	Notes
<code>x or y</code>	if <code>x</code> is false, then <code>y</code> , else <code>x</code>	(1)
<code>x and y</code>	if <code>x</code> is false, then <code>x</code> , else <code>y</code>	(2)
<code>not x</code>	if <code>x</code> is false, then <code>True</code> , else <code>False</code>	(3)

¹ Additional information on these special methods may be found in the Python Reference Manual (customization).

Notes:

- (1) This is a short-circuit operator, so it only evaluates the second argument if the first one is false.
- (2) This is a short-circuit operator, so it only evaluates the second argument if the first one is true.
- (3) `not` has a lower priority than non-Boolean operators, so `not a == b` is interpreted as `not (a == b)`, and `a == not b` is a syntax error.

4.3 Comparisons

There are eight comparison operations in Python. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, `x < y <= z` is equivalent to `x < y` and `y <= z`, except that `y` is evaluated only once (but in both cases `z` is not evaluated at all when `x < y` is found to be false).

This table summarizes the comparison operations:

Operation	Meaning
<code><</code>	strictly less than
<code><=</code>	less than or equal
<code>></code>	strictly greater than
<code>>=</code>	greater than or equal
<code>==</code>	equal
<code>!=</code>	not equal
<code>is</code>	object identity
<code>is not</code>	negated object identity

Objects of different types, except different numeric types, never compare equal. The `==` operator is always defined but for some object types (for example, class objects) is equivalent to `is`. The `<`, `<=`, `>` and `>=` operators are only defined where they make sense; for example, they raise a `TypeError` exception when one of the arguments is a complex number.

Non-identical instances of a class normally compare as non-equal unless the class defines the `__eq__()` method.

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines enough of the methods `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` (in general, `__lt__()` and `__eq__()` are sufficient, if you want the conventional meanings of the comparison operators).

The behavior of the `is` and `is not` operators cannot be customized; also they can be applied to any two objects and never raise an exception.

Two more operations with the same syntactic priority, `in` and `not in`, are supported by types that are `iterable` or implement the `__contains__()` method.

4.4 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 the additional numeric types `fractions.Fraction`, for rationals, and `decimal.Decimal`, for 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 (for priorities of the operations, see operator-summary):

Operation	Result	Notes	Full documentation
<code>x + y</code>	sum of <code>x</code> and <code>y</code>		
<code>x - y</code>	difference of <code>x</code> and <code>y</code>		
<code>x * y</code>	product of <code>x</code> and <code>y</code>		
<code>x / y</code>	quotient of <code>x</code> and <code>y</code>		
<code>x // y</code>	floored quotient of <code>x</code> and <code>y</code>	(1)	
<code>x % y</code>	remainder of <code>x / y</code>	(2)	
<code>-x</code>	<code>x</code> negated		
<code>+x</code>	<code>x</code> unchanged		
<code>abs(x)</code>	absolute value or magnitude of <code>x</code>		<code>abs()</code>
<code>int(x)</code>	<code>x</code> converted to integer	(3)(6)	<code>int()</code>
<code>float(x)</code>	<code>x</code> converted to floating point	(4)(6)	<code>float()</code>
<code>complex(re, im)</code>	a complex number with real part <code>re</code> , imaginary part <code>im</code> . <code>im</code> defaults to zero.	(6)	<code>complex()</code>
<code>c.conjugate()</code>	conjugate of the complex number <code>c</code>		
<code>divmod(x, y)</code>	the pair <code>(x // y, x % y)</code>	(2)	<code>divmod()</code>
<code>pow(x, y)</code>	<code>x</code> to the power <code>y</code>	(5)	<code>pow()</code>
<code>x ** y</code>	<code>x</code> to the power <code>y</code>	(5)	

Notes:

- (1) Also referred to as integer division. The resultant value is a whole integer, though the result’s type is not necessarily `int`. The result is always rounded towards minus infinity: `1//2` is 0, `(-1)//2` is -1, `1//(-2)` is -1, and `(-1)//(-2)` is 0.
- (2) Not for complex numbers. Instead convert to floats using `abs()` if appropriate.
- (3) Conversion from floating point to integer may round or truncate as in C; see functions `math.floor()` and `math.ceil()` for well-defined conversions.
- (4) `float` also accepts the strings “nan” and “inf” with an optional prefix “+” or “-” for Not a Number (NaN) and positive or negative infinity.
- (5) Python defines `pow(0, 0)` and `0 ** 0` to be 1, as is common for programming languages.
- (6) The numeric literals accepted include the digits 0 to 9 or any Unicode equivalent (code points with the Nd property). See <http://www.unicode.org/Public/12.1.0/ucd/extracted/DerivedNumericType.txt> for a complete list of code points with the Nd property.

All `numbers.Real` types (`int` and `float`) also include the following operations:

² As a consequence, the list `[1, 2]` is considered equal to `[1.0, 2.0]`, and similarly for tuples.

Operation	Result
<code>math.trunc(x)</code>	x truncated to <code>Integral</code>
<code>round(x[, n])</code>	x rounded to n digits, rounding half to even. If n is omitted, it defaults to 0.
<code>math.floor(x)</code>	the greatest <code>Integral</code> $\leq x$
<code>math.ceil(x)</code>	the least <code>Integral</code> $\geq x$

For additional numeric operations see the `math` and `cmath` modules.

4.4.1 Bitwise Operations on Integer Types

Bitwise operations only make sense for integers. The result of bitwise operations is calculated as though carried out in two's complement with an infinite number of sign bits.

The priorities of the binary bitwise operations are all lower than the numeric operations and higher than the comparisons; the unary operation `~` has the same priority as the other unary numeric operations (`+` and `-`).

This table lists the bitwise operations sorted in ascending priority:

Operation	Result	Notes
<code>x y</code>	bitwise <i>or</i> of x and y	(4)
<code>x ^ y</code>	bitwise <i>exclusive or</i> of x and y	(4)
<code>x & y</code>	bitwise <i>and</i> of x and y	(4)
<code>x << n</code>	x shifted left by n bits	(1)(2)
<code>x >> n</code>	x shifted right by n bits	(1)(3)
<code>~x</code>	the bits of x inverted	

Notes:

- (1) Negative shift counts are illegal and cause a `ValueError` to be raised.
- (2) A left shift by n bits is equivalent to multiplication by `pow(2, n)` without overflow check.
- (3) A right shift by n bits is equivalent to division by `pow(2, n)` without overflow check.
- (4) Performing these calculations with at least one extra sign extension bit in a finite two's complement representation (a working bit-width of $1 + \max(x.bit_length(), y.bit_length())$ or more) is sufficient to get the same result as if there were an infinite number of sign bits.

4.4.2 Additional Methods on Integer Types

The `int` type implements the `numbers.Integral` abstract base class. In addition, it provides a few more methods:

`int.bit_length()`

Return the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros:

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

More precisely, if x is nonzero, then `x.bit_length()` is the unique positive integer k such that $2^{k-1} \leq \text{abs}(x) < 2^k$. Equivalently, when `abs(x)` is small enough to have a correctly rounded logarithm, then $k = 1 + \text{int}(\log(\text{abs}(x), 2))$. If x is zero, then `x.bit_length()` returns 0.

Equivalent to:

```
def bit_length(self):
    s = bin(self)           # binary representation: bin(-37) --> '-0b100101'
    s = s.lstrip('-0b')     # remove leading zeros and minus sign
    return len(s)           # len('100101') --> 6
```

New in version 3.1.

`int.to_bytes(length, byteorder, *, signed=False)`

Return an array of bytes representing an integer.

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() + 7) // 8, byteorder='little')
b'\xe8\x03'
```

The integer is represented using *length* bytes. An *OverflowError* is raised if the integer is not representable with the given number of bytes.

The *byteorder* argument determines the byte order used to represent the integer. If *byteorder* is "big", the most significant byte is at the beginning of the byte array. If *byteorder* is "little", the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value.

The *signed* argument determines whether two's complement is used to represent the integer. If *signed* is False and a negative integer is given, an *OverflowError* is raised. The default value for *signed* is False.

New in version 3.2.

`classmethod int.from_bytes(bytes, byteorder, *, signed=False)`

Return the integer represented by the given array of bytes.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

The argument *bytes* must either be a *bytes-like object* or an iterable producing bytes.

The *byteorder* argument determines the byte order used to represent the integer. If *byteorder* is "big", the most significant byte is at the beginning of the byte array. If *byteorder* is "little", the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value.

The *signed* argument indicates whether two's complement is used to represent the integer.

New in version 3.2.

`int.as_integer_ratio()`

Return a pair of integers whose ratio is exactly equal to the original integer and with a positive denominator. The integer ratio of integers (whole numbers) is always the integer as the numerator and 1 as the denominator.

New in version 3.8.

4.4.3 Additional Methods on Float

The float type implements the `numbers.Real` abstract base class. float also has the following additional methods.

`float.as_integer_ratio()`

Return a pair of integers whose ratio is exactly equal to the original float and with a positive denominator. Raises `OverflowError` on infinities and a `ValueError` on NaNs.

`float.is_integer()`

Return True if the float instance is finite with integral value, and False otherwise:

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

Two methods support conversion to and from hexadecimal strings. Since Python’s floats are stored internally as binary numbers, converting a float to or from a *decimal* string usually involves a small rounding error. In contrast, hexadecimal strings allow exact representation and specification of floating-point numbers. This can be useful when debugging, and in numerical work.

`float.hex()`

Return a representation of a floating-point number as a hexadecimal string. For finite floating-point numbers, this representation will always include a leading `0x` and a trailing `p` and exponent.

`classmethod float.fromhex(s)`

Class method to return the float represented by a hexadecimal string *s*. The string *s* may have leading and trailing whitespace.

Note that `float.hex()` is an instance method, while `float.fromhex()` is a class method.

A hexadecimal string takes the form:

```
[sign] ['0x'] integer ['. fraction] ['p' exponent]
```

where the optional `sign` may be either + or -, `integer` and `fraction` are strings of hexadecimal digits, and `exponent` is a decimal integer with an optional leading sign. Case is not significant, and there must be at least one hexadecimal digit in either the integer or the fraction. This syntax is similar to the syntax specified in section 6.4.4.2 of the C99 standard, and also to the syntax used in Java 1.5 onwards. In particular, the output of `float.hex()` is usable as a hexadecimal floating-point literal in C or Java code, and hexadecimal strings produced by C’s `%a` format character or Java’s `Double.toHexString` are accepted by `float.fromhex()`.

Note that the exponent is written in decimal rather than hexadecimal, and that it gives the power of 2 by which to multiply the coefficient. For example, the hexadecimal string `0x3.a7p10` represents the floating-point number $(3 + 7 \cdot 16^{-2}) \cdot 2^{10}$, or 3740.0:

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

Applying the reverse conversion to `3740.0` gives a different hexadecimal string representing the same number:

```
>>> float.hex(3740.0)
'0x1.d3800000000000p+11'
```

4.4.4 Hashing of numeric types

For numbers x and y , possibly of different types, it's a requirement that $\text{hash}(x) == \text{hash}(y)$ whenever $x == y$ (see the `__hash__()` method documentation for more details). For ease of implementation and efficiency across a variety of numeric types (including `int`, `float`, `decimal.Decimal` and `fractions.Fraction`) Python's hash for numeric types is based on a single mathematical function that's defined for any rational number, and hence applies to all instances of `int` and `fractions.Fraction`, and all finite instances of `float` and `decimal.Decimal`. Essentially, this function is given by reduction modulo P for a fixed prime P . The value of P is made available to Python as the `modulus` attribute of `sys.hash_info`.

CPython implementation detail: Currently, the prime used is $P = 2^{31} - 1$ on machines with 32-bit C longs and $P = 2^{61} - 1$ on machines with 64-bit C longs.

Here are the rules in detail:

- If $x = m / n$ is a nonnegative rational number and n is not divisible by P , define $\text{hash}(x)$ as $m * \text{invmod}(n, P) \% P$, where `invmod(n, P)` gives the inverse of n modulo P .
- If $x = m / n$ is a nonnegative rational number and n is divisible by P (but m is not) then n has no inverse modulo P and the rule above doesn't apply; in this case define $\text{hash}(x)$ to be the constant value `sys.hash_info.inf`.
- If $x = m / n$ is a negative rational number define $\text{hash}(x)$ as $-\text{hash}(-x)$. If the resulting hash is -1 , replace it with -2 .
- The particular values `sys.hash_info.inf`, `-sys.hash_info.inf` and `sys.hash_info.nan` are used as hash values for positive infinity, negative infinity, or nans (respectively). (All hashable nans have the same hash value.)
- For a `complex` number z , the hash values of the real and imaginary parts are combined by computing $\text{hash}(z.real) + \text{sys.hash_info.imag} * \text{hash}(z.imag)$, reduced modulo $2^{sys.hash_info.width}$ so that it lies in range $(-2^{sys.hash_info.width - 1}, 2^{sys.hash_info.width - 1})$. Again, if the result is -1 , it's replaced with -2 .

To clarify the above rules, here's some example Python code, equivalent to the built-in hash, for computing the hash of a rational number, `float`, or `complex`:

```
import sys, math

def hash_fraction(m, n):
    """Compute the hash of a rational number m / n.

    Assumes m and n are integers, with n positive.
    Equivalent to hash(fractions.Fraction(m, n)).

    """
    P = sys.hash_info.modulus
    # Remove common factors of P. (Unnecessary if m and n already coprime.)
    while m % P == n % P == 0:
        m, n = m // P, n // P

    if n % P == 0:
        hash_value = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: pow(n, P-1, P) is 1, so
        hash_value = m * pow(n, P-2, P) % P
```

(continues on next page)

(continued from previous page)

```

# pow(n, P-2, P) gives the inverse of n modulo P.
hash_value = (abs(m) % P) * pow(n, P - 2, P) % P
if m < 0:
    hash_value = -hash_value
if hash_value == -1:
    hash_value = -2
return hash_value

def hash_float(x):
    """Compute the hash of a float x."""

    if math.isnan(x):
        return sys.hash_info.nan
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""

    hash_value = hash_float(z.real) + sys.hash_info.imag * hash_float(z.imag)
    # do a signed reduction modulo 2**sys.hash_info.width
    M = 2**sys.hash_info.width - 1
    hash_value = (hash_value & (M - 1)) - (hash_value & M)
    if hash_value == -1:
        hash_value = -2
    return hash_value

```

4.5 Iterator Types

Python supports a concept of iteration over containers. This is implemented using two distinct methods; these are used to allow user-defined classes to support iteration. Sequences, described below in more detail, always support the iteration methods.

One method needs to be defined for container objects to provide iteration support:

`container.__iter__()`

Return an iterator object. The object is required to support the iterator protocol described below. If a container supports different types of iteration, additional methods can be provided to specifically request iterators for those iteration types. (An example of an object supporting multiple forms of iteration would be a tree structure which supports both breadth-first and depth-first traversal.) This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

The iterator objects themselves are required to support the following two methods, which together form the *iterator protocol*:

`iterator.__iter__()`

Return the iterator object itself. This is required to allow both containers and iterators to be used with the `for` and `in` statements. This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

`iterator.__next__()`

Return the next item from the container. If there are no further items, raise the `StopIteration` exception. This method corresponds to the `tp_iternext` slot of the type structure for Python objects in the Python/C API.

Python defines several iterator objects to support iteration over general and specific sequence types, dictionaries, and other more specialized forms. The specific types are not important beyond their implementation of the iterator protocol.

Once an iterator's `__next__()` method raises `StopIteration`, it must continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken.

4.5.1 Generator Types

Python's *generators* provide a convenient way to implement the iterator protocol. If a container object's `__iter__()` method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the `__iter__()` and `__next__()` methods. More information about generators can be found in the documentation for the `yield` expression.

4.6 Sequence Types — `list`, `tuple`, `range`

There are three basic sequence types: lists, tuples, and range objects. Additional sequence types tailored for processing of *binary data* and *text strings* are described in dedicated sections.

4.6.1 Common Sequence Operations

The operations in the following table are supported by most sequence types, both mutable and immutable. The `collections.abc.Sequence` ABC is provided to make it easier to correctly implement these operations on custom sequence types.

This table lists the sequence operations sorted in ascending priority. In the table, *s* and *t* are sequences of the same type, *n*, *i*, *j* and *k* are integers and *x* is an arbitrary object that meets any type and value restrictions imposed by *s*.

The `in` and `not in` operations have the same priorities as the comparison operations. The `+` (concatenation) and `*` (repetition) operations have the same priority as the corresponding numeric operations.³

Operation	Result	Notes
<code>x in s</code>	True if an item of <i>s</i> is equal to <i>x</i> , else False	(1)
<code>x not in s</code>	False if an item of <i>s</i> is equal to <i>x</i> , else True	(1)
<code>s + t</code>	the concatenation of <i>s</i> and <i>t</i>	(6)(7)
<code>s * n</code> or <code>n * s</code>	equivalent to adding <i>s</i> to itself <i>n</i> times	(2)(7)
<code>s[i]</code>	<i>i</i> th item of <i>s</i> , origin 0	(3)
<code>s[i:j]</code>	slice of <i>s</i> from <i>i</i> to <i>j</i>	(3)(4)
<code>s[i:j:k]</code>	slice of <i>s</i> from <i>i</i> to <i>j</i> with step <i>k</i>	(3)(5)
<code>len(s)</code>	length of <i>s</i>	
<code>min(s)</code>	smallest item of <i>s</i>	
<code>max(s)</code>	largest item of <i>s</i>	
<code>s.index(x[, i[, j]])</code>	index of the first occurrence of <i>x</i> in <i>s</i> (at or after index <i>i</i> and before index <i>j</i>)	(8)
<code>s.count(x)</code>	total number of occurrences of <i>x</i> in <i>s</i>	

Sequences of the same type also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see comparisons in the language reference.)

Notes:

³ They must have since the parser can't tell the type of the operands.

- (1) While the `in` and `not in` operations are used only for simple containment testing in the general case, some specialised sequences (such as `str`, `bytes` and `bytearray`) also use them for subsequence testing:

```
>>> "gg" in "eggs"
True
```

- (2) Values of n less than 0 are treated as 0 (which yields an empty sequence of the same type as s). Note that items in the sequence s are not copied; they are referenced multiple times. This often haunts new Python programmers; consider:

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

What has happened is that `[]` is a one-element list containing an empty list, so all three elements of `[] * 3` are references to this single empty list. Modifying any of the elements of `lists` modifies this single list. You can create a list of different lists this way:

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

Further explanation is available in the FAQ entry [faq-multidimensional-list](#).

- (3) If i or j is negative, the index is relative to the end of sequence s : `len(s) + i` or `len(s) + j` is substituted. But note that -0 is still 0.
- (4) The slice of s from i to j is defined as the sequence of items with index k such that $i \leq k < j$. If i or j is greater than `len(s)`, use `len(s)`. If i is omitted or `None`, use 0. If j is omitted or `None`, use `len(s)`. If i is greater than or equal to j , the slice is empty.
- (5) The slice of s from i to j with step k is defined as the sequence of items with index $x = i + n*k$ such that $0 \leq n < (j-i)/k$. In other words, the indices are $i, i+k, i+2*k, i+3*k$ and so on, stopping when j is reached (but never including j). When k is positive, i and j are reduced to `len(s)` if they are greater. When k is negative, i and j are reduced to `len(s) - 1` if they are greater. If i or j are omitted or `None`, they become “end” values (which end depends on the sign of k). Note, k cannot be zero. If k is `None`, it is treated like 1.
- (6) Concatenating immutable sequences always results in a new object. This means that building up a sequence by repeated concatenation will have a quadratic runtime cost in the total sequence length. To get a linear runtime cost, you must switch to one of the alternatives below:
 - if concatenating `str` objects, you can build a list and use `str.join()` at the end or else write to an `io.StringIO` instance and retrieve its value when complete
 - if concatenating `bytes` objects, you can similarly use `bytes.join()` or `io.BytesIO`, or you can do in-place concatenation with a `bytearray` object. `bytearray` objects are mutable and have an efficient overallocation mechanism
 - if concatenating `tuple` objects, extend a `list` instead
 - for other types, investigate the relevant class documentation
- (7) Some sequence types (such as `range`) only support item sequences that follow specific patterns, and hence don’t support sequence concatenation or repetition.

- (8) `index` raises `ValueError` when `x` is not found in `s`. Not all implementations support passing the additional arguments `i` and `j`. These arguments allow efficient searching of subsections of the sequence. Passing the extra arguments is roughly equivalent to using `s[i:j].index(x)`, only without copying any data and with the returned index being relative to the start of the sequence rather than the start of the slice.

4.6.2 Immutable Sequence Types

The only operation that immutable sequence types generally implement that is not also implemented by mutable sequence types is support for the `hash()` built-in.

This support allows immutable sequences, such as `tuple` instances, to be used as `dict` keys and stored in `set` and `frozenset` instances.

Attempting to hash an immutable sequence that contains unhashable values will result in `TypeError`.

4.6.3 Mutable Sequence Types

The operations in the following table are defined on mutable sequence types. The `collections.abc.MutableSequence` ABC is provided to make it easier to correctly implement these operations on custom sequence types.

In the table `s` is an instance of a mutable sequence type, `t` is any iterable object and `x` is an arbitrary object that meets any type and value restrictions imposed by `s` (for example, `bytearray` only accepts integers that meet the value restriction `0 <= x <= 255`).

Operation	Result	Notes
<code>s[i] = x</code>	item <code>i</code> of <code>s</code> is replaced by <code>x</code>	
<code>s[i:j] = t</code>	slice of <code>s</code> from <code>i</code> to <code>j</code> is replaced by the contents of the iterable <code>t</code>	
<code>del s[i:j]</code>	same as <code>s[i:j] = []</code>	
<code>s[i:j:k] = t</code>	the elements of <code>s[i:j:k]</code> are replaced by those of <code>t</code>	(1)
<code>del s[i:j:k]</code>	removes the elements of <code>s[i:j:k]</code> from the list	
<code>s.append(x)</code>	appends <code>x</code> to the end of the sequence (same as <code>s[len(s):len(s)] = [x]</code>)	
<code>s.clear()</code>	removes all items from <code>s</code> (same as <code>del s[:]</code>)	(5)
<code>s.copy()</code>	creates a shallow copy of <code>s</code> (same as <code>s[:]</code>)	(5)
<code>s.extend(t)</code> or <code>s += t</code>	extends <code>s</code> with the contents of <code>t</code> (for the most part the same as <code>s[len(s):len(s)] = t</code>)	
<code>s *= n</code>	updates <code>s</code> with its contents repeated <code>n</code> times	(6)
<code>s.insert(i, x)</code>	inserts <code>x</code> into <code>s</code> at the index given by <code>i</code> (same as <code>s[i:i] = [x]</code>)	
<code>s.pop([i])</code>	retrieves the item at <code>i</code> and also removes it from <code>s</code>	(2)
<code>s.remove(x)</code>	remove the first item from <code>s</code> where <code>s[i]</code> is equal to <code>x</code>	(3)
<code>s.reverse()</code>	reverses the items of <code>s</code> in place	(4)

Notes:

- (1) `t` must have the same length as the slice it is replacing.
- (2) The optional argument `i` defaults to `-1`, so that by default the last item is removed and returned.
- (3) `remove()` raises `ValueError` when `x` is not found in `s`.
- (4) The `reverse()` method modifies the sequence in place for economy of space when reversing a large sequence. To remind users that it operates by side effect, it does not return the reversed sequence.

(5) `clear()` and `copy()` are included for consistency with the interfaces of mutable containers that don't support slicing operations (such as `dict` and `set`). `copy()` is not part of the `collections.abc.MutableSequence` ABC, but most concrete mutable sequence classes provide it.

New in version 3.3: `clear()` and `copy()` methods.

(6) The value `n` is an integer, or an object implementing `__index__()`. Zero and negative values of `n` clear the sequence. Items in the sequence are not copied; they are referenced multiple times, as explained for `s * n` under *Common Sequence Operations*.

4.6.4 Lists

Lists are mutable sequences, typically used to store collections of homogeneous items (where the precise degree of similarity will vary by application).

`class list([iterable])`

Lists may be constructed in several ways:

- Using a pair of square brackets to denote the empty list: `[]`
- Using square brackets, separating items with commas: `[a], [a, b, c]`
- Using a list comprehension: `[x for x in iterable]`
- Using the type constructor: `list()` or `list(iterable)`

The constructor builds a list whose items are the same and in the same order as `iterable`'s items. `iterable` may be either a sequence, a container that supports iteration, or an iterator object. If `iterable` is already a list, a copy is made and returned, similar to `iterable[:]`. For example, `list('abc')` returns `['a', 'b', 'c']` and `list((1, 2, 3))` returns `[1, 2, 3]`. If no argument is given, the constructor creates a new empty list, `[]`.

Many other operations also produce lists, including the `sorted()` built-in.

Lists implement all of the `common` and `mutable` sequence operations. Lists also provide the following additional method:

`sort(*, key=None, reverse=False)`

This method sorts the list in place, using only `<` comparisons between items. Exceptions are not suppressed – if any comparison operations fail, the entire sort operation will fail (and the list will likely be left in a partially modified state).

`sort()` accepts two arguments that can only be passed by keyword (*keyword-only arguments*):

`key` specifies a function of one argument that is used to extract a comparison key from each list element (for example, `key=str.lower`). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of `None` means that list items are sorted directly without calculating a separate key value.

The `functools.cmp_to_key()` utility is available to convert a 2.x style `cmp` function to a `key` function.

`reverse` is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

This method modifies the sequence in place for economy of space when sorting a large sequence. To remind users that it operates by side effect, it does not return the sorted sequence (use `sorted()` to explicitly request a new sorted list instance).

The `sort()` method is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

For sorting examples and a brief sorting tutorial, see [sortinghowto](#).

CPython implementation detail: While a list is being sorted, the effect of attempting to mutate, or even inspect, the list is undefined. The C implementation of Python makes the list appear empty for the duration, and raises [ValueError](#) if it can detect that the list has been mutated during a sort.

4.6.5 Tuples

Tuples are immutable sequences, typically used to store collections of heterogeneous data (such as the 2-tuples produced by the [enumerate\(\)](#) built-in). Tuples are also used for cases where an immutable sequence of homogeneous data is needed (such as allowing storage in a [set](#) or [dict](#) instance).

```
class tuple([iterable])
```

Tuples may be constructed in a number of ways:

- Using a pair of parentheses to denote the empty tuple: ()
- Using a trailing comma for a singleton tuple: a, or (a,)
- Separating items with commas: a, b, c or (a, b, c)
- Using the [tuple\(\)](#) built-in: tuple() or tuple(iterable)

The constructor builds a tuple whose items are the same and in the same order as *iterable*'s items. *iterable* may be either a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a tuple, it is returned unchanged. For example, tuple('abc') returns ('a', 'b', 'c') and tuple([1, 2, 3]) returns (1, 2, 3). If no argument is given, the constructor creates a new empty tuple, ().

Note that it is actually the comma which makes a tuple, not the parentheses. The parentheses are optional, except in the empty tuple case, or when they are needed to avoid syntactic ambiguity. For example, f(a, b, c) is a function call with three arguments, while f((a, b, c)) is a function call with a 3-tuple as the sole argument.

Tuples implement all of the [common](#) sequence operations.

For heterogeneous collections of data where access by name is clearer than access by index, [collections.namedtuple\(\)](#) may be a more appropriate choice than a simple tuple object.

4.6.6 Ranges

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))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

Ranges implement all of the [common](#) sequence operations except concatenation and repetition (due to the fact that range objects can only represent sequences that follow a strict pattern and repetition and concatenation will usually violate that pattern).

start

The value of the *start* parameter (or 0 if the parameter was not supplied)

stop

The value of the *stop* parameter

step

The value of the *step* parameter (or 1 if the parameter was not supplied)

The advantage of the `range` type over a regular `list` or `tuple` is that a `range` object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the `start`, `stop` and `step` values, calculating individual items and subranges as needed).

Range objects implement the `collections.abc.Sequence` ABC, and provide features such as containment tests, element index lookup, slicing and support for negative indices (see [Sequence Types — `list`, `tuple`, `range`](#)):

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

Testing range objects for equality with `==` and `!=` compares them as sequences. That is, two range objects are considered equal if they represent the same sequence of values. (Note that two range objects that compare equal might have different `start`, `stop` and `step` attributes, for example `range(0) == range(2, 1, 3)` or `range(0, 3, 2) == range(0, 4, 2)`.)

Changed in version 3.2: Implement the Sequence ABC. Support slicing and negative indices. Test `int` objects for membership in constant time instead of iterating through all items.

Changed in version 3.3: Define ‘==’ and ‘!=’ to compare range objects based on the sequence of values they define (instead of comparing based on object identity).

New in version 3.3: The `start`, `stop` and `step` attributes.

See also:

- The [linspace recipe](#) shows how to implement a lazy version of range suitable for floating point applications.

4.7 Text Sequence Type — `str`

Textual data in Python is handled with `str` objects, or *strings*. Strings are immutable *sequences* of Unicode code points. String literals are written in a variety of ways:

- Single quotes: ' allows embedded "double" quotes'
- Double quotes: " allows embedded 'single' quotes".
- Triple quoted: '''Three single quotes''', """Three double quotes"""

Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.

String literals that are part of a single expression and have only whitespace between them will be implicitly converted to a single string literal. That is, ("spam " "eggs") == "spam eggs".

See [strings](#) for more about the various forms of string literal, including supported escape sequences, and the `r` (“raw”) prefix that disables most escape sequence processing.

Strings may also be created from other objects using the `str` constructor.

Since there is no separate “character” type, indexing a string produces strings of length 1. That is, for a non-empty string `s`, `s[0] == s[0:1]`.

There is also no mutable string type, but `str.join()` or `io.StringIO` can be used to efficiently construct strings from multiple fragments.

Changed in version 3.3: For backwards compatibility with the Python 2 series, the `u` prefix is once again permitted on string literals. It has no effect on the meaning of string literals and cannot be combined with the `r` prefix.

```
class str(object=’’)
class str(object=b”, encoding=’utf-8’, errors=’strict’)
```

Return a `string` version of `object`. If `object` is not provided, returns the empty string. Otherwise, the behavior of `str()` depends on whether `encoding` or `errors` is given, as follows.

If neither `encoding` nor `errors` is given, `str(object)` returns `object.__str__()`, which is the “informal” or nicely printable string representation of `object`. For string objects, this is the string itself. If `object` does not have a `__str__()` method, then `str()` falls back to returning `repr(object)`.

If at least one of `encoding` or `errors` is given, `object` should be a *bytes-like object* (e.g. `bytes` or `bytearray`). In this case, if `object` is a `bytes` (or `bytearray`) object, then `str(bytes, encoding, errors)` is equivalent to `bytes.decode(encoding, errors)`. Otherwise, the bytes object underlying the buffer object is obtained before calling `bytes.decode()`. See [Binary Sequence Types — bytes, bytearray, memoryview](#) and [bufferobjects](#) for information on buffer objects.

Passing a `bytes` object to `str()` without the `encoding` or `errors` arguments falls under the first case of returning the informal string representation (see also the `-b` command-line option to Python). For example:

```
>>> str(b’Zoot!’)
"b'Zoot!'"
```

For more information on the `str` class and its methods, see [Text Sequence Type — `str`](#) and the [String Methods](#) section below. To output formatted strings, see the f-strings and [Format String Syntax](#) sections. In addition, see the [Text Processing Services](#) section.

4.7.1 String Methods

Strings implement all of the [common](#) sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see [`str.format\(\)`](#), [Format String Syntax](#) and [Custom String Formatting](#)) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle ([printf-style String Formatting](#)).

The [Text Processing Services](#) section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the `re` module).

`str.capitalize()`

Return a copy of the string with its first character capitalized and the rest lowercased.

Changed in version 3.8: The first character is now put into titlecase rather than uppercase. This means that characters like digraphs will only have their first letter capitalized, instead of the full character.

`str.casefold()`

Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

CASEFOLDING is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter 'ß' is equivalent to "ss". Since it is already lowercase, `lower()` would do nothing to 'ß'; `casefold()` converts it to "ss".

The casefolding algorithm is described in section 3.13 of the Unicode Standard.

New in version 3.3.

`str.center(width[, fillchar])`

Return centered in a string of length `width`. Padding is done using the specified `fillchar` (default is an ASCII space). The original string is returned if `width` is less than or equal to `len(s)`.

`str.count(sub[, start[, end]])`

Return the number of non-overlapping occurrences of substring `sub` in the range `[start, end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

`str.encode(encoding="utf-8", errors="strict")`

Return an encoded version of the string as a bytes object. Default encoding is 'utf-8'. `errors` may be given to set a different error handling scheme. The default for `errors` is 'strict', meaning that encoding errors raise a [UnicodeError](#). Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other name registered via `codecs.register_error()`, see section [Error Handlers](#). For a list of possible encodings, see section [Standard Encodings](#).

Changed in version 3.1: Support for keyword arguments added.

`str.endswith(suffix[, start[, end]])`

Return True if the string ends with the specified `suffix`, otherwise return False. `suffix` can also be a tuple of suffixes to look for. With optional `start`, test beginning at that position. With optional `end`, stop comparing at that position.

`str.expandtabs(tabsize=8)`

Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every `tabsize` characters (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the string, the current column is set to zero and the string is examined character by character. If the character is a tab (\t), one or more space characters are inserted in the result until the

current column is equal to the next tab position. (The tab character itself is not copied.) If the character is a newline (\n) or return (\r), it is copied and the current column is reset to zero. Any other character is copied unchanged and the current column is incremented by one regardless of how the character is represented when printed.

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012      0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012  0123   01234'
```

`str.find(sub[, start[, end]])`

Return the lowest index in the string where substring *sub* is found within the slice *s[start:end]*. Optional arguments *start* and *end* are interpreted as in slice notation. Return -1 if *sub* is not found.

Note: The `find()` method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the `in` operator:

```
>>> 'Py' in 'Python'
True
```

`str.format(*args, **kwargs)`

Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces {}. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

See [Format String Syntax](#) for a description of the various formatting options that can be specified in format strings.

Note: When formatting a number (`int`, `float`, `complex`, `decimal.Decimal` and subclasses) with the `n` type (ex: `'{ :n }'.format(1234)`), the function temporarily sets the `LC_CTYPE` locale to the `LC_NUMERIC` locale to decode `decimal_point` and `thousands_sep` fields of `localeconv()` if they are non-ASCII or longer than 1 byte, and the `LC_NUMERIC` locale is different than the `LC_CTYPE` locale. This temporary change affects other threads.

Changed in version 3.7: When formatting a number with the `n` type, the function sets temporarily the `LC_CTYPE` locale to the `LC_NUMERIC` locale in some cases.

`str.format_map(mapping)`

Similar to `str.format(**mapping)`, except that *mapping* is used directly and not copied to a `dict`. This is useful if for example *mapping* is a `dict` subclass:

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

New in version 3.2.

`str.index(sub[, start[, end]])`

Like `find()`, but raise `ValueError` when the substring is not found.

`str.isalnum()`

Return True if all characters in the string are alphanumeric and there is at least one character, False otherwise. A character `c` is alphanumeric if one of the following returns True: `c.isalpha()`, `c.isdecimal()`, `c.isdigit()`, or `c.isnumeric()`.

`str.isalpha()`

Return True if all characters in the string are alphabetic and there is at least one character, False otherwise. Alphabetic characters are those characters defined in the Unicode character database as “Letter”, i.e., those with general category property being one of “Lm”, “Lt”, “Lu”, “Ll”, or “Lo”. Note that this is different from the “Alphabetic” property defined in the Unicode Standard.

`str.isascii()`

Return True if the string is empty or all characters in the string are ASCII, False otherwise. ASCII characters have code points in the range U+0000-U+007F.

New in version 3.7.

`str.isdecimal()`

Return True if all characters in the string are decimal characters and there is at least one character, False otherwise. Decimal characters are those that can be used to form numbers in base 10, e.g. U+0660, ARABIC-INDIC DIGIT ZERO. Formally a decimal character is a character in the Unicode General Category “Nd”.

`str.isdigit()`

Return True if all characters in the string are digits and there is at least one character, False otherwise. Digits include decimal characters and digits that need special handling, such as the compatibility superscript digits. This covers digits which cannot be used to form numbers in base 10, like the Kharosthi numbers. Formally, a digit is a character that has the property value Numeric_Type=Digit or Numeric_Type=Decimal.

`str.isidentifier()`

Return True if the string is a valid identifier according to the language definition, section identifiers.

Call `keyword.iskeyword()` to test whether string `s` is a reserved identifier, such as `def` and `class`.

Example:

```
>>> from keyword import iskeyword
>>> 'hello'.isidentifier(), iskeyword('hello')
True, False
>>> 'def'.isidentifier(), iskeyword('def')
True, True
```

`str.islower()`

Return True if all cased characters⁴ in the string are lowercase and there is at least one cased character, False otherwise.

`str.isnumeric()`

Return True if all characters in the string are numeric characters, and there is at least one character, False otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH. Formally, numeric characters are those with the property value Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric.

`str.isprintable()`

Return True if all characters in the string are printable or the string is empty, False otherwise. Nonprintable characters are those characters defined in the Unicode character database as “Other” or “Separator”, excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when `repr()` is invoked on a string. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.)

⁴ Cased characters are those with general category property being one of “Lu” (Letter, uppercase), “Ll” (Letter, lowercase), or “Lt” (Letter, titlecase).

str.isspace()

Return True if there are only whitespace characters in the string and there is at least one character, False otherwise.

A character is *whitespace* if in the Unicode character database (see [unicodedata](#)), either its general category is Zs (“Separator, space”), or its bidirectional class is one of WS, B, or S.

str.istitle()

Return True if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

str.isupper()

Return True if all cased characters⁴ in the string are uppercase and there is at least one cased character, False otherwise.

str.join(iterable)

Return a string which is the concatenation of the strings in *iterable*. A [TypeError](#) will be raised if there are any non-string values in *iterable*, including [bytes](#) objects. The separator between elements is the string providing this method.

str.ljust(width[, fillchar])

Return the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

str.lower()

Return a copy of the string with all the cased characters⁴ converted to lowercase.

The lowercasing algorithm used is described in section 3.13 of the Unicode Standard.

str.lstrip([chars])

Return a copy of the string with leading characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or None, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped:

```
>>> '    spacious    '.lstrip()
'spacious'
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

static str.maketrans(x[, y[, z]])

This static method returns a translation table usable for `str.translate()`.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters (strings of length 1) to Unicode ordinals, strings (of arbitrary lengths) or None. Character keys will then be converted to ordinals.

If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in *x* will be mapped to the character at the same position in *y*. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

str.partition(sep)

Split the string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

str.replace(old, new[, count])

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

`str.rfind(sub[, start[, end]])`

Return the highest index in the string where substring *sub* is found, such that *sub* is contained within *s[start:end]*. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

`str.rindex(sub[, start[, end]])`

Like `rfind()` but raises `ValueError` when the substring *sub* is not found.

`str.rjust(width[, fillchar])`

Return the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.rpartition(sep)`

Split the string at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

`str.rsplit(sep=None, maxsplit=-1)`

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any whitespace string is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below.

`str.rstrip([chars])`

Return a copy of the string with trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> '    spacious    '.rstrip()
'    spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

`str.split(sep=None, maxsplit=-1)`

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most *maxsplit*+1 elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `[]`.

For example:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
```

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

For example:

```
>>> '1 2 3'.split()
['1', '2', '3']
```

(continues on next page)

(continued from previous page)

```
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> ' 1 2 3 '.split()
['1', '2', '3']
```

str.splitlines([keepends])

Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless *keepends* is given and true.

This method splits on the following line boundaries. In particular, the boundaries are a superset of *universal new-lines*.

Representation	Description
\n	Line Feed
\r	Carriage Return
\r\n	Carriage Return + Line Feed
\v or \x0b	Line Tabulation
\f or \x0c	Form Feed
\x1c	File Separator
\x1d	Group Separator
\x1e	Record Separator
\x85	Next Line (C1 Control Code)
\u2028	Line Separator
\u2029	Paragraph Separator

Changed in version 3.2: \v and \f added to list of line boundaries.

For example:

```
>>> 'ab c\nnde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\nnde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

Unlike *split()* when a delimiter string *sep* is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

For comparison, *split('\n')* gives:

```
>>> ''.split('\n')
[']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

str.startswith(prefix[, start[, end]])

Return True if string starts with the *prefix*, otherwise return False. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test string beginning at that position. With optional *end*, stop comparing string at that position.

`str.strip([chars])`

Return a copy of the string with the leading and trailing characters removed. The `chars` argument is a string specifying the set of characters to be removed. If omitted or `None`, the `chars` argument defaults to removing whitespace. The `chars` argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```
>>> '    spacious    '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

The outermost leading and trailing `chars` argument values are stripped from the string. Characters are removed from the leading end until reaching a string character that is not contained in the set of characters in `chars`. A similar action takes place on the trailing end. For example:

```
>>> comment_string = '#..... Section 3.2.1 Issue #32 .....'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

`str.swapcase()`

Return a copy of the string with uppercase characters converted to lowercase and vice versa. Note that it is not necessarily true that `s.swapcase().swapcase() == s`.

`str.title()`

Return a titlecased version of the string where words start with an uppercase character and the remaining characters are lowercase.

For example:

```
>>> 'Hello world'.title()
'Hello World'
```

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

```
>>> "they're bill's friends from the UK".title()
"They'Re Bill'S Friends From The Uk"
```

A workaround for apostrophes can be constructed using regular expressions:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                  lambda mo: mo.group(0).capitalize(),
...                  s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

`str.translate(table)`

Return a copy of the string in which each character has been mapped through the given translation table. The table must be an object that implements indexing via `__getitem__()`, typically a `mapping` or `sequence`. When indexed by a Unicode ordinal (an integer), the table object can do any of the following: return a Unicode ordinal or a string, to map the character to one or more other characters; return `None`, to delete the character from the return string; or raise a `LookupError` exception, to map the character to itself.

You can use `str.maketrans()` to create a translation map from character-to-character mappings in different formats.

See also the [codecs](#) module for a more flexible approach to custom character mappings.

`str.upper()`

Return a copy of the string with all the cased characters⁴ converted to uppercase. Note that `s.upper().isupper()` might be `False` if `s` contains uncased characters or if the Unicode category of the resulting character(s) is not “Lu” (Letter, uppercase), but e.g. “Lt” (Letter, titlecase).

The uppercasing algorithm used is described in section 3.13 of the Unicode Standard.

`str.zfill(width)`

Return a copy of the string left filled with ASCII '0' digits to make a string of length `width`. A leading sign prefix ('+'/'-') is handled by inserting the padding *after* the sign character rather than before. The original string is returned if `width` is less than or equal to `len(s)`.

For example:

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

4.7.2 printf-style String Formatting

Note: The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer formatted string literals, the `str.format()` interface, or [template strings](#) may help avoid these errors. Each of these alternatives provides their own trade-offs and benefits of simplicity, flexibility, and/or extensibility.

String objects have one unique built-in operation: the `%` operator (modulo). This is also known as the string *formatting* or *interpolation* operator. Given `format % values` (where `format` is a string), `%` conversion specifications in `format` are replaced with zero or more elements of `values`. The effect is similar to using the `sprintf()` in the C language.

If `format` requires a single argument, `values` may be a single non-tuple object.⁵ Otherwise, `values` must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The '%' character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, `(somename)`).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an '*' (asterisk), the actual width is read from the next element of the tuple in `values`, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a '.' (dot) followed by the precision. If specified as '*' (an asterisk), the actual precision is read from the next element of the tuple in `values`, and the value to convert comes after the precision.
6. Length modifier (optional).
7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the string *must* include a parenthesised mapping key into that dictionary inserted immediately after the '%' character. The mapping key selects the value to be formatted from the mapping. For example:

⁵ To format only a tuple you should therefore provide a singleton tuple whose only element is the tuple to be formatted.

```
>>> print('%(language)s has %(number)03d quote types.' %
...      {'language': "Python", "number": 2})
Python has 002 quote types.
```

In this case no * specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

Flag	Meaning
'#'	The value conversion will use the “alternate form” (where defined below).
'0'	The conversion will be zero padded for numeric values.
'-'	The converted value is left adjusted (overrides the '0' conversion if both are given).
' '	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
'+'	A sign character ('+' or '-') will precede the conversion (overrides a “space” flag).

A length modifier (h, l, or L) may be present, but is ignored as it is not necessary for Python – so e.g. %ld is identical to %d.

The conversion types are:

Conversion	Meaning	Notes
'd'	Signed integer decimal.	
'i'	Signed integer decimal.	
'o'	Signed octal value.	(1)
'u'	Obsolete type – it is identical to 'd'.	(6)
'x'	Signed hexadecimal (lowercase).	(2)
'X'	Signed hexadecimal (uppercase).	(2)
'e'	Floating point exponential format (lowercase).	(3)
'E'	Floating point exponential format (uppercase).	(3)
'f'	Floating point decimal format.	(3)
'F'	Floating point decimal format.	(3)
'g'	Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'G'	Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'c'	Single character (accepts integer or single character string).	
'r'	String (converts any Python object using <code>repr()</code>).	(5)
's'	String (converts any Python object using <code>str()</code>).	(5)
'a'	String (converts any Python object using <code>ascii()</code>).	(5)
'%'	No argument is converted, results in a '%' character in the result.	

Notes:

- (1) The alternate form causes a leading octal specifier ('0o') to be inserted before the first digit.
- (2) The alternate form causes a leading '0x' or '0X' (depending on whether the 'x' or 'X' format was used) to be inserted before the first digit.
- (3) The alternate form causes the result to always contain a decimal point, even if no digits follow it.
The precision determines the number of digits after the decimal point and defaults to 6.
- (4) The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

The precision determines the number of significant digits before and after the decimal point and defaults to 6.

- (5) If precision is N, the output is truncated to N characters.
- (6) See [PEP 237](#).

Since Python strings have an explicit length, %s conversions do not assume that '\0' is the end of the string.

Changed in version 3.1: %f conversions for numbers whose absolute value is over 1e50 are no longer replaced by %g conversions.

4.8 Binary Sequence Types — `bytes`, `bytearray`, `memoryview`

The core built-in types for manipulating binary data are `bytes` and `bytearray`. They are supported by `memoryview` which uses the buffer protocol to access the memory of other binary objects without needing to make a copy.

The `array` module supports efficient storage of basic data types like 32-bit integers and IEEE754 double-precision floating values.

4.8.1 Bytes Objects

Bytes objects are immutable sequences of single bytes. Since many major binary protocols are based on the ASCII text encoding, bytes objects offer several methods that are only valid when working with ASCII compatible data and are closely related to string objects in a variety of other ways.

`class bytes([source[, encoding[, errors]]])`

Firstly, the syntax for bytes literals is largely the same as that for string literals, except that a `b` prefix is added:

- Single quotes: `b'still allows embedded "double" quotes'`
- Double quotes: `b"still allows embedded 'single' quotes".`
- Triple quoted: `b'''3 single quotes'''`, `b"""\3 double quotes"""`

Only ASCII characters are permitted in bytes literals (regardless of the declared source code encoding). Any binary values over 127 must be entered into bytes literals using the appropriate escape sequence.

As with string literals, bytes literals may also use a `r` prefix to disable processing of escape sequences. See strings for more about the various forms of bytes literal, including supported escape sequences.

While bytes literals and representations are based on ASCII text, bytes objects actually behave like immutable sequences of integers, with each value in the sequence restricted such that $0 \leq x < 256$ (attempts to violate this restriction will trigger `ValueError`). This is done deliberately to emphasise that while many binary formats include ASCII based elements and can be usefully manipulated with some text-oriented algorithms, this is not generally the case for arbitrary binary data (blindly applying text processing algorithms to binary data formats that are not ASCII compatible will usually lead to data corruption).

In addition to the literal forms, bytes objects can be created in a number of other ways:

- A zero-filled bytes object of a specified length: `bytes(10)`
- From an iterable of integers: `bytes(range(20))`
- Copying existing binary data via the buffer protocol: `bytes(obj)`

Also see the `bytes` built-in.

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing binary data. Accordingly, the bytes type has an additional class method to read data in that format:

classmethod fromhex(string)

This `bytes` class method returns a bytes object, decoding the given string object. The string must contain two hexadecimal digits per byte, with ASCII whitespace being ignored.

```
>>> bytes.fromhex('2Ef0 F1f2 ')
b'\xf0\xf1\xf2'
```

Changed in version 3.7: `bytes.fromhex()` now skips all ASCII whitespace in the string, not just spaces.

A reverse conversion function exists to transform a bytes object into its hexadecimal representation.

hex([sep[, bytes_per_sep]])

Return a string object containing two hexadecimal digits for each byte in the instance.

```
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

If you want to make the hex string easier to read, you can specify a single character separator `sep` parameter to include in the output. By default between each byte. A second optional `bytes_per_sep` parameter controls the spacing. Positive values calculate the separator position from the right, negative values from the left.

```
>>> value = b'\xf0\xf1\xf2'
>>> value.hex('-')
'f0-f1-f2'
>>> value.hex('_', 2)
'f0_f1f2'
>>> b'UUDDLRLRAB'.hex(' ', -4)
'55554444 4c524c52 4142'
```

New in version 3.5.

Changed in version 3.8: `bytes.hex()` now supports optional `sep` and `bytes_per_sep` parameters to insert separators between bytes in the hex output.

Since bytes objects are sequences of integers (akin to a tuple), for a bytes object `b`, `b[0]` will be an integer, while `b[0:1]` will be a bytes object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytes objects uses the literal format (`b'...'`) since it is often more useful than e.g. `bytes([46, 46, 46])`. You can always convert a bytes object into a list of integers using `list(b)`.

Note: For Python 2.x users: In the Python 2.x series, a variety of implicit conversions between 8-bit strings (the closest thing 2.x offers to a built-in binary data type) and Unicode strings were permitted. This was a backwards compatibility workaround to account for the fact that Python originally only supported 8-bit text, and Unicode text was a later addition. In Python 3.x, those implicit conversions are gone - conversions between 8-bit binary data and Unicode text must be explicit, and bytes and string objects will always compare unequal.

4.8.2 Bytearray Objects

`bytearray` objects are a mutable counterpart to `bytes` objects.

class `bytearray`(`[source[, encoding[, errors]]]`)

There is no dedicated literal syntax for bytearray objects, instead they are always created by calling the constructor:

- Creating an empty instance: `bytearray()`
- Creating a zero-filled instance with a given length: `bytearray(10)`
- From an iterable of integers: `bytearray(range(20))`
- Copying existing binary data via the buffer protocol: `bytearray(b'Hi !')`

As bytearray objects are mutable, they support the *mutable* sequence operations in addition to the common bytes and bytearray operations described in *Bytes and Bytearray Operations*.

Also see the `bytearray` built-in.

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing binary data. Accordingly, the bytearray type has an additional class method to read data in that format:

classmethod `fromhex`(`string`)

This `bytearray` class method returns bytearray object, decoding the given string object. The string must contain two hexadecimal digits per byte, with ASCII whitespace being ignored.

```
>>> bytearray.fromhex('2Ef0 F1f2 ')
bytearray(b'.\xf0\xf1\xf2')
```

Changed in version 3.7: `bytearray.fromhex()` now skips all ASCII whitespace in the string, not just spaces.

A reverse conversion function exists to transform a bytearray object into its hexadecimal representation.

`hex([sep[, bytes_per_sep]])`

Return a string object containing two hexadecimal digits for each byte in the instance.

```
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

New in version 3.5.

Changed in version 3.8: Similar to `bytes.hex()`, `bytearray.hex()` now supports optional `sep` and `bytes_per_sep` parameters to insert separators between bytes in the hex output.

Since bytearray objects are sequences of integers (akin to a list), for a bytearray object `b`, `b[0]` will be an integer, while `b[0:1]` will be a bytearray object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytearray objects uses the bytes literal format (`bytearray(b'...')`) since it is often more useful than e.g. `bytearray([46, 46, 46])`. You can always convert a bytearray object into a list of integers using `list(b)`.

4.8.3 Bytes and Bytearray Operations

Both bytes and bytearray objects support the [common](#) sequence operations. They interoperate not just with operands of the same type, but with any [bytes-like object](#). Due to this flexibility, they can be freely mixed in operations without causing errors. However, the return type of the result may depend on the order of operands.

Note: The methods on bytes and bytearray objects don't accept strings as their arguments, just as the methods on strings don't accept bytes as their arguments. For example, you have to write:

```
a = "abc"  
b = a.replace("a", "f")
```

and:

```
a = b"abc"  
b = a.replace(b"a", b"f")
```

Some bytes and bytearray operations assume the use of ASCII compatible binary formats, and hence should be avoided when working with arbitrary binary data. These restrictions are covered below.

Note: Using these ASCII based operations to manipulate binary data that is not stored in an ASCII based format may lead to data corruption.

The following methods on bytes and bytearray objects can be used with arbitrary binary data.

`bytes.count(sub[, start[, end]])`
`bytearray.count(sub[, start[, end]])`

Return the number of non-overlapping occurrences of subsequence *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

The subsequence to search for may be any [bytes-like object](#) or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

`bytes.decode(encoding="utf-8", errors="strict")`
`bytearray.decode(encoding="utf-8", errors="strict")`

Return a string decoded from the given bytes. Default encoding is 'utf-8'. *errors* may be given to set a different error handling scheme. The default for *errors* is 'strict', meaning that encoding errors raise a [UnicodeError](#). Other possible values are 'ignore', 'replace' and any other name registered via `codecs.register_error()`, see section [Error Handlers](#). For a list of possible encodings, see section [Standard Encodings](#).

Note: Passing the *encoding* argument to `str` allows decoding any [bytes-like object](#) directly, without needing to make a temporary bytes or bytearray object.

Changed in version 3.1: Added support for keyword arguments.

`bytes.endswith(suffix[, start[, end]])`
`bytearray.endswith(suffix[, start[, end]])`

Return True if the binary data ends with the specified *suffix*, otherwise return False. *suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

The suffix(es) to search for may be any [bytes-like object](#).

```
bytes.find(sub[, start[, end]])  
bytarray.find(sub[, start[, end]])
```

Return the lowest index in the data where the subsequence *sub* is found, such that *sub* is contained in the slice *s[start:end]*. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Note: The `find()` method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the `in` operator:

```
>>> b'Py' in b'Python'  
True
```

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

```
bytes.index(sub[, start[, end]])  
bytarray.index(sub[, start[, end]])
```

Like `find()`, but raise `ValueError` when the subsequence is not found.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

```
bytes.join(iterable)  
bytarray.join(iterable)
```

Return a bytes or bytarray object which is the concatenation of the binary data sequences in *iterable*. A `TypeError` will be raised if there are any values in *iterable* that are not *bytes-like objects*, including `str` objects. The separator between elements is the contents of the bytes or bytarray object providing this method.

```
static bytes.maketrans(from, to)  
static bytarray.maketrans(from, to)
```

This static method returns a translation table usable for `bytes.translate()` that will map each character in *from* into the character at the same position in *to*; *from* and *to* must both be *bytes-like objects* and have the same length.

New in version 3.1.

```
bytes.partition(sep)  
bytarray.partition(sep)
```

Split the sequence at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself or its bytarray copy, and the part after the separator. If the separator is not found, return a 3-tuple containing a copy of the original sequence, followed by two empty bytes or bytarray objects.

The separator to search for may be any *bytes-like object*.

```
bytes.replace(old, new[, count])  
bytarray.replace(old, new[, count])
```

Return a copy of the sequence with all occurrences of subsequence *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

The subsequence to search for and its replacement may be any *bytes-like object*.

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
bytes.rfind(sub[, start[, end]])
```

`bytearray.rfind(sub[, start[, end]])`

Return the highest index in the sequence where the subsequence *sub* is found, such that *sub* is contained within *s* [start:end]. Optional arguments *start* and *end* are interpreted as in slice notation. Return -1 on failure.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

`bytes.rindex(sub[, start[, end]])`

`bytearray.rindex(sub[, start[, end]])`

Like `rfind()` but raises `ValueError` when the subsequence *sub* is not found.

The subsequence to search for may be any *bytes-like object* or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

`bytes.rpartition(sep)`

`bytearray.rpartition(sep)`

Split the sequence at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself or its bytearray copy, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty bytes or bytearray objects, followed by a copy of the original sequence.

The separator to search for may be any *bytes-like object*.

`bytes.startswith(prefix[, start[, end]])`

`bytearray.startswith(prefix[, start[, end]])`

Return True if the binary data starts with the specified *prefix*, otherwise return False. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

The prefix(es) to search for may be any *bytes-like object*.

`bytes.translate(table, /, delete=b")`

`bytearray.translate(table, /, delete=b")`

Return a copy of the bytes or bytearray object where all bytes occurring in the optional argument *delete* are removed, and the remaining bytes have been mapped through the given translation table, which must be a bytes object of length 256.

You can use the `bytes.maketrans()` method to create a translation table.

Set the *table* argument to None for translations that only delete characters:

```
>>> b'read this short text'.translate(None, b'aeiou')
b'r d ths shrt txt'
```

Changed in version 3.6: *delete* is now supported as a keyword argument.

The following methods on bytes and bytearray objects have default behaviours that assume the use of ASCII compatible binary formats, but can still be used with arbitrary binary data by passing appropriate arguments. Note that all of the bytearray methods in this section do *not* operate in place, and instead produce new objects.

`bytes.center(width[, fillbyte])`

`bytearray.center(width[, fillbyte])`

Return a copy of the object centered in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
bytes.ljust(width[, fillbyte])
bytarray.ljust(width[, fillbyte])
```

Return a copy of the object left justified in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
bytes.lstrip([chars])
bytarray.lstrip([chars])
```

Return a copy of the sequence with specified leading bytes removed. The *chars* argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or `None`, the *chars* argument defaults to removing ASCII whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped:

```
>>> b'    spacious    '.lstrip()
b'spacious'
>>> b'www.example.com'.lstrip(b'cmowz.')
b'example.com'
```

The binary sequence of byte values to remove may be any *bytes-like object*.

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
bytes.rjust(width[, fillbyte])
bytarray.rjust(width[, fillbyte])
```

Return a copy of the object right justified in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
bytes.rsplit(sep=None, maxsplit=-1)
bytarray.rsplit(sep=None, maxsplit=-1)
```

Split the binary sequence into subsequences of the same type, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any subsequence consisting solely of ASCII whitespace is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below.

```
bytes.rstrip([chars])
bytarray.rstrip([chars])
```

Return a copy of the sequence with specified trailing bytes removed. The *chars* argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or `None`, the *chars* argument defaults to removing ASCII whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> b'    spacious    '.rstrip()
b'    spacious'
```

(continues on next page)

(continued from previous page)

```
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

The binary sequence of byte values to remove may be any *bytes-like object*.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.split(sep=None, maxsplit=-1)`
`bytearray.split(sep=None, maxsplit=-1)`

Split the binary sequence into subsequences of the same type, using *sep* as the delimiter string. If *maxsplit* is given and non-negative, at most *maxsplit* splits are done (thus, the list will have at most *maxsplit*+1 elements). If *maxsplit* is not specified or is *-1*, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty subsequences (for example, `b'1,,2'.split(b',')` returns `[b'1', b'', b'2']`). The *sep* argument may consist of a multibyte sequence (for example, `b'1<>2<>3'.split(b'<>')` returns `[b'1', b'2', b'3']`). Splitting an empty sequence with a specified separator returns `[b'']` or `[bytearray(b'')]` depending on the type of object being split. The *sep* argument may be any *bytes-like object*.

For example:

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,,2,,3,'.split(b',')
[b'1', b'2', b'', b'3', b'']
```

If *sep* is not specified or is *None*, a different splitting algorithm is applied: runs of consecutive ASCII whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the sequence has leading or trailing whitespace. Consequently, splitting an empty sequence or a sequence consisting solely of ASCII whitespace without a specified separator returns `[]`.

For example:

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b'    1    2    3    '.split()
[b'1', b'2', b'3']
```

`bytes.strip([chars])`
`bytearray.strip([chars])`

Return a copy of the sequence with specified leading and trailing bytes removed. The *chars* argument is a binary sequence specifying the set of byte values to be removed - the name refers to the fact this method is usually used with ASCII characters. If omitted or *None*, the *chars* argument defaults to removing ASCII whitespace. The *chars* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```
>>> b'    spacious    '.strip()
b'spacious'
>>> b'www.example.com'.strip(b'cmowz.')
b'example'
```

The binary sequence of byte values to remove may be any *bytes-like object*.

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

The following methods on bytes and bytarray objects assume the use of ASCII compatible binary formats and should not be applied to arbitrary binary data. Note that all of the bytarray methods in this section do *not* operate in place, and instead produce new objects.

```
bytes.capitalize()
bytarray.capitalize()
```

Return a copy of the sequence with each byte interpreted as an ASCII character, and the first byte capitalized and the rest lowercased. Non-ASCII byte values are passed through unchanged.

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
bytes.expandtabs (tabsize=8)
bytarray.expandtabs (tabsize=8)
```

Return a copy of the sequence where all ASCII tab characters are replaced by one or more ASCII spaces, depending on the current column and the given tab size. Tab positions occur every *tabsize* bytes (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the sequence, the current column is set to zero and the sequence is examined byte by byte. If the byte is an ASCII tab character (b'\t'), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the current byte is an ASCII newline (b'\n') or carriage return (b'\r'), it is copied and the current column is reset to zero. Any other byte value is copied unchanged and the current column is incremented by one regardless of how the byte value is represented when printed:

```
>>> b'01\t012\t0123\t01234'.expandtabs()
b'01      012      0123    01234'
>>> b'01\t012\t0123\t01234'.expandtabs(4)
b'01  012  0123  01234'
```

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

```
bytes.isalnum()
bytarray.isalnum()
```

Return True if all bytes in the sequence are alphabetical ASCII characters or ASCII decimal digits and the sequence is not empty, False otherwise. Alphabetic ASCII characters are those byte values in the sequence b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'. ASCII decimal digits are those byte values in the sequence b'0123456789'.

For example:

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

```
bytes.isalpha()
bytarray.isalpha()
```

Return True if all bytes in the sequence are alphabetic ASCII characters and the sequence is

not empty, `False` otherwise. Alphabetic ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

For example:

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

`bytes.isascii()`
`bytearray.isascii()`

Return `True` if the sequence is empty or all bytes in the sequence are ASCII, `False` otherwise. ASCII bytes are in the range 0-0x7F.

New in version 3.7.

`bytes.isdigit()`
`bytearray.isdigit()`

Return `True` if all bytes in the sequence are ASCII decimal digits and the sequence is not empty, `False` otherwise. ASCII decimal digits are those byte values in the sequence `b'0123456789'`.

For example:

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

`bytes.islower()`
`bytearray.islower()`

Return `True` if there is at least one lowercase ASCII character in the sequence and no uppercase ASCII characters, `False` otherwise.

For example:

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

`bytes.isspace()`
`bytearray.isspace()`

Return `True` if all bytes in the sequence are ASCII whitespace and the sequence is not empty, `False` otherwise. ASCII whitespace characters are those byte values in the sequence `b' \t\n\r\x0b\f'` (space, tab, newline, carriage return, vertical tab, form feed).

`bytes.istitle()`
`bytearray.istitle()`

Return `True` if the sequence is ASCII titlecase and the sequence is not empty, `False` otherwise. See `bytes.title()` for more details on the definition of “titlecase”.

For example:

```
>>> b'Hello World'.istitle()
True
```

(continues on next page)

(continued from previous page)

```
>>> b'Hello world'.istitle()
False
```

`bytes.isupper()`
`bytearray.isupper()`

Return True if there is at least one uppercase alphabetic ASCII character in the sequence and no lowercase ASCII characters, False otherwise.

For example:

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

`bytes.lower()`
`bytearray.lower()`

Return a copy of the sequence with all the uppercase ASCII characters converted to their corresponding lowercase counterpart.

For example:

```
>>> b'Hello World'.lower()
b'hello world'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.splitlines(keepends=False)`
`bytearray.splitlines(keepends=False)`

Return a list of the lines in the binary sequence, breaking at ASCII line boundaries. This method uses the [universal newlines](#) approach to splitting lines. Line breaks are not included in the resulting list unless `keepends` is given and true.

For example:

```
>>> b'ab c\nnde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\nnde fg\rkl\r\n'.splitlines(keepends=True)
[b'ab c\n', b'\n', b'de fg\r', b'kl\r\n']
```

Unlike `split()` when a delimiter string `sep` is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([b''], [b'Two lines', b''])
>>> b"".splitlines(), b"One line\n".splitlines()
([], [b'One line'])
```

`bytes.swapcase()`

`bytearray.swapcase()`

Return a copy of the sequence with all the lowercase ASCII characters converted to their corresponding uppercase counterpart and vice-versa.

For example:

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Unlike `str.swapcase()`, it is always the case that `bin.swapcase().swapcase() == bin` for the binary versions. Case conversions are symmetrical in ASCII, even though that is not generally true for arbitrary Unicode code points.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.title()`

`bytearray.title()`

Return a titlecased version of the binary sequence where words start with an uppercase ASCII character and the remaining characters are lowercase. Uncased byte values are left unmodified.

For example:

```
>>> b'Hello world'.title()
b'Hello World'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. All other byte values are uncased.

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

```
>>> b"they're bill's friends from the UK".title()
b"They'Re Bill'S Friends From The Uk"
```

A workaround for apostrophes can be constructed using regular expressions:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(rb"[A-Za-z]+('[A-Za-z]+)?",
...                  lambda mo: mo.group(0)[0:1].upper() +
...                             mo.group(0)[1:].lower(),
...                  s)
...
>>> titlecase(b"they're bill's friends.")
b"They're Bill's Friends."
```

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.upper()`

`bytearray.upper()`

Return a copy of the sequence with all the lowercase ASCII characters converted to their corresponding uppercase counterpart.

For example:

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`bytes.zfill(width)`

`bytearray.zfill(width)`

Return a copy of the sequence left filled with ASCII `b'0'` digits to make a sequence of length `width`. A leading sign prefix (`b'+' / b'-'`) is handled by inserting the padding *after* the sign character rather than before. For `bytes` objects, the original sequence is returned if `width` is less than or equal to `len(seq)`.

For example:

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

Note: The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

4.8.4 printf-style Bytes Formatting

Note: The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). If the value being printed may be a tuple or dictionary, wrap it in a tuple.

Bytes objects (`bytes/bytarray`) have one unique built-in operation: the `%` operator (modulo). This is also known as the bytes *formatting* or *interpolation* operator. Given `format % values` (where `format` is a bytes object), `%` conversion specifications in `format` are replaced with zero or more elements of `values`. The effect is similar to using the `sprintf()` in the C language.

If `format` requires a single argument, `values` may be a single non-tuple object.⁵ Otherwise, `values` must be a tuple with exactly the number of items specified by the format bytes object, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The `'%'` character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, `(somename)`).
3. Conversion flags (optional), which affect the result of some conversion types.

4. Minimum field width (optional). If specified as an '*' (asterisk), the actual width is read from the next element of the tuple in *values*, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a '.' (dot) followed by the precision. If specified as '*' (an asterisk), the actual precision is read from the next element of the tuple in *values*, and the value to convert comes after the precision.
6. Length modifier (optional).
7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the bytes object *must* include a parenthesised mapping key into that dictionary inserted immediately after the '%' character. The mapping key selects the value to be formatted from the mapping. For example:

```
>>> print(b'%(%(language)s has %(number)03d quote types.' %
...       {b'language': b"Python", b"number": 2})
b'Python has 002 quote types.'
```

In this case no * specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

Flag	Meaning
' #'	The value conversion will use the “alternate form” (where defined below).
' 0 '	The conversion will be zero padded for numeric values.
' - '	The converted value is left adjusted (overrides the ' 0 ' conversion if both are given).
' '	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
' + '	A sign character (' +' or ' - ') will precede the conversion (overrides a “space” flag).

A length modifier (h, l, or L) may be present, but is ignored as it is not necessary for Python – so e.g. %ld is identical to %d.

The conversion types are:

Conversion	Meaning	Notes
'd'	Signed integer decimal.	
'i'	Signed integer decimal.	
'o'	Signed octal value.	(1)
'u'	Obsolete type – it is identical to 'd'.	(8)
'x'	Signed hexadecimal (lowercase).	(2)
'X'	Signed hexadecimal (uppercase).	(2)
'e'	Floating point exponential format (lowercase).	(3)
'E'	Floating point exponential format (uppercase).	(3)
'f'	Floating point decimal format.	(3)
'F'	Floating point decimal format.	(3)
'g'	Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'G'	Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'c'	Single byte (accepts integer or single byte objects).	
'b'	Bytes (any object that follows the buffer protocol or has <code>__bytes__()</code>).	(5)
's'	's' is an alias for 'b' and should only be used for Python2/3 code bases.	(6)
'a'	Bytes (converts any Python object using <code>repr(obj).encode('ascii', 'backslashreplace')</code>).	(5)
'r'	'r' is an alias for 'a' and should only be used for Python2/3 code bases.	(7)
'%'	No argument is converted, results in a '%' character in the result.	

Notes:

- (1) The alternate form causes a leading octal specifier ('0o') to be inserted before the first digit.
- (2) The alternate form causes a leading '0x' or '0X' (depending on whether the 'x' or 'X' format was used) to be inserted before the first digit.
- (3) The alternate form causes the result to always contain a decimal point, even if no digits follow it.
The precision determines the number of digits after the decimal point and defaults to 6.
- (4) The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.
The precision determines the number of significant digits before and after the decimal point and defaults to 6.
- (5) If precision is N, the output is truncated to N characters.
- (6) b'%s' is deprecated, but will not be removed during the 3.x series.
- (7) b'%r' is deprecated, but will not be removed during the 3.x series.
- (8) See [PEP 237](#).

Note: The bytarray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

See also:

[PEP 461](#) - Adding % formatting to bytes and bytarray

New in version 3.5.

4.8.5 Memory Views

`memoryview` objects allow Python code to access the internal data of an object that supports the buffer protocol without copying.

```
class memoryview(obj)
```

Create a `memoryview` that references `obj`. `obj` must support the buffer protocol. Built-in objects that support the buffer protocol include `bytes` and `bytearray`.

A `memoryview` has the notion of an *element*, which is the atomic memory unit handled by the originating object `obj`. For many simple types such as `bytes` and `bytearray`, an element is a single byte, but other types such as `array.array` may have bigger elements.

`len(view)` is equal to the length of `tolist`. If `view.ndim = 0`, the length is 1. If `view.ndim = 1`, the length is equal to the number of elements in the view. For higher dimensions, the length is equal to the length of the nested list representation of the view. The `itemsize` attribute will give you the number of bytes in a single element.

A `memoryview` supports slicing and indexing to expose its data. One-dimensional slicing will result in a subview:

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

If `format` is one of the native format specifiers from the `struct` module, indexing with an integer or a tuple of integers is also supported and returns a single *element* with the correct type. One-dimensional memoryviews can be indexed with an integer or a one-integer tuple. Multi-dimensional memoryviews can be indexed with tuples of exactly `ndim` integers where `ndim` is the number of dimensions. Zero-dimensional memoryviews can be indexed with the empty tuple.

Here is an example with a non-byte format:

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::-2].tolist()
[-11111111, -33333333]
```

If the underlying object is writable, the memoryview supports one-dimensional slice assignment. Resizing is not allowed:

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
```

(continues on next page)

(continued from previous page)

```
>>> v[1:4] = b'123'
>>> data
bytarray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different structures
>>> v[2:6] = b'spam'
>>> data
bytarray(b'z1spam')
```

One-dimensional memoryviews of hashable (read-only) types with formats ‘B’, ‘b’ or ‘c’ are also hashable. The hash is defined as `hash(m) == hash(m.tobytes())`:

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

Changed in version 3.3: One-dimensional memoryviews can now be sliced. One-dimensional memoryviews with formats ‘B’, ‘b’ or ‘c’ are now hashable.

Changed in version 3.4: `memoryview` is now registered automatically with `collections.abc.Sequence`

Changed in version 3.5: memoryviews can now be indexed with tuple of integers.

`memoryview` has several methods:

`__eq__(exporter)`

A memoryview and a [PEP 3118](#) exporter are equal if their shapes are equivalent and if all corresponding values are equal when the operands’ respective format codes are interpreted using `struct` syntax.

For the subset of `struct` format strings currently supported by `tolist()`, `v` and `w` are equal if `v.tolist() == w.tolist()`:

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

If either format string is not supported by the `struct` module, then the objects will always compare as unequal (even if the format strings and buffer contents are identical):

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

Note that, as with floating point numbers, `v is w` does *not* imply `v == w` for `memoryview` objects.

Changed in version 3.3: Previous versions compared the raw memory disregarding the item format and the logical array structure.

`tobytes(order=None)`

Return the data in the buffer as a bytestring. This is equivalent to calling the `bytes` constructor on the `memoryview`.

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

For non-contiguous arrays the result is equal to the flattened list representation with all elements converted to bytes. `tobytes()` supports all format strings, including those that are not in `struct` module syntax.

New in version 3.8: `order` can be {'C', 'F', 'A'}. When `order` is 'C' or 'F', the data of the original array is converted to C or Fortran order. For contiguous views, 'A' returns an exact copy of the physical memory. In particular, in-memory Fortran order is preserved. For non-contiguous views, the data is converted to C first. `order=None` is the same as `order='C'`.

`hex([sep[, bytes_per_sep]])`

Return a string object containing two hexadecimal digits for each byte in the buffer.

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

New in version 3.5.

Changed in version 3.8: Similar to `bytes.hex()`, `memoryview.hex()` now supports optional `sep` and `bytes_per_sep` parameters to insert separators between bytes in the hex output.

`tolist()`

Return the data in the buffer as a list of elements.

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

Changed in version 3.3: `tolist()` now supports all single character native formats in `struct` module syntax as well as multi-dimensional representations.

`toreadonly()`

Return a readonly version of the memoryview object. The original memoryview object is unchanged.

```
>>> m = memoryview(bytarray(b'abc'))
>>> mm = m.toreadonly()
>>> mm.tolist()
[89, 98, 99]
>>> mm[0] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> m[0] = 43
>>> mm.tolist()
[43, 98, 99]
```

New in version 3.8.

`release()`

Release the underlying buffer exposed by the memoryview object. Many objects take special actions when a view is held on them (for example, a `bytarray` would temporarily forbid resizing); therefore, calling `release()` is handy to remove these restrictions (and free any dangling resources) as soon as possible.

After this method has been called, any further operation on the view raises a `ValueError` (except `release()` itself which can be called multiple times):

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

The context management protocol can be used for a similar effect, using the `with` statement:

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

New in version 3.2.

`cast(format[, shape])`

Cast a memoryview to a new format or shape. `shape` defaults to `[byte_length//new_itemsize]`, which means that the result view will be one-dimensional. The return value is a new memoryview, but the buffer itself is not copied. Supported casts are 1D -> C-contiguous and C-contiguous -> 1D.

The destination format is restricted to a single element native format in `struct` syntax. One of the formats must be a byte format ('B', 'b' or 'c'). The byte length of the result must be the same as the original length.

Cast 1D/long to 1D/unsigned bytes:

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

Cast 1D/unsigned bytes to 1D/char:

```
>>> b = bytearray(b'xyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview: invalid value for format "B"
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

Cast 1D/bytes to 3D/ints to 1D/signed char:

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
```

(continues on next page)

(continued from previous page)

```
>>> z.nbytes
48
```

Cast 1D/unsigned long to 2D/unsigned long:

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

New in version 3.3.

Changed in version 3.5: The source format is no longer restricted when casting to a byte view.

There are also several readonly attributes available:

obj

The underlying object of the memoryview:

```
>>> b = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

New in version 3.3.

nbytes

`nbytes == product(shape) * itemsize == len(m.tobytes())`. This is the amount of space in bytes that the array would use in a contiguous representation. It is not necessarily equal to `len(m)`:

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

Multi-dimensional arrays:

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
```

(continues on next page)

(continued from previous page)

```
>>> len(y)
3
>>> y.nbytes
96
```

New in version 3.3.

readonly

A bool indicating whether the memory is read only.

format

A string containing the format (in `struct` module style) for each element in the view. A memoryview can be created from exporters with arbitrary format strings, but some methods (e.g. `tolist()`) are restricted to native single element formats.

Changed in version 3.3: format 'B' is now handled according to the struct module syntax. This means that `memoryview(b'abc')[0] == b'abc'[0] == 97.`

itemsize

The size in bytes of each element of the memoryview:

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim

An integer indicating how many dimensions of a multi-dimensional array the memory represents.

shape

A tuple of integers the length of `ndim` giving the shape of the memory as an N-dimensional array.

Changed in version 3.3: An empty tuple instead of `None` when `ndim = 0`.

strides

A tuple of integers the length of `ndim` giving the size in bytes to access each element for each dimension of the array.

Changed in version 3.3: An empty tuple instead of `None` when `ndim = 0`.

suboffsets

Used internally for PIL-style arrays. The value is informational only.

c_contiguous

A bool indicating whether the memory is C-*contiguous*.

New in version 3.3.

f_contiguous

A bool indicating whether the memory is Fortran *contiguous*.

New in version 3.3.

contiguous

A bool indicating whether the memory is *contiguous*.

New in version 3.3.

4.9 Set Types — `set`, `frozenset`

A `set` object is an unordered collection of distinct *hashable* objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference. (For other containers see the built-in `dict`, `list`, and `tuple` classes, and the `collections` module.)

Like other collections, sets support `x` in `set`, `len(set)`, and `for x in set`. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

There are currently two built-in set types, `set` and `frozenset`. The `set` type is mutable — the contents can be changed using methods like `add()` and `remove()`. Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. The `frozenset` type is immutable and *hashable* — its contents cannot be altered after it is created; it can therefore be used as a dictionary key or as an element of another set.

Non-empty sets (not frozensets) can be created by placing a comma-separated list of elements within braces, for example: `{'jack', 'sjoerd'}`, in addition to the `set` constructor.

The constructors for both classes work the same:

```
class set([iterable])
class frozenset([iterable])
```

Return a new set or frozenset object whose elements are taken from *iterable*. The elements of a set must be *hashable*.

To represent sets of sets, the inner sets must be `frozenset` objects. If *iterable* is not specified, a new empty set is returned.

Instances of `set` and `frozenset` provide the following operations:

`len(s)`

Return the number of elements in set *s* (cardinality of *s*).

`x in s`

Test *x* for membership in *s*.

`x not in s`

Test *x* for non-membership in *s*.

`isdisjoint(other)`

Return `True` if the set has no elements in common with *other*. Sets are disjoint if and only if their intersection is the empty set.

`issubset(other)`

`set <= other`

Test whether every element in the set is in *other*.

`set < other`

Test whether the set is a proper subset of *other*, that is, `set <= other` and `set != other`.

`issuperset(other)`

`set >= other`

Test whether every element in *other* is in the set.

`set > other`

Test whether the set is a proper superset of *other*, that is, `set >= other` and `set != other`.

`union(*others)`

`set | other | ...`

Return a new set with elements from the set and all others.

`intersection(*others)`

set & other & ...

Return a new set with elements common to the set and all others.

difference(*others)

set - other - ...

Return a new set with elements in the set that are not in the others.

symmetric_difference(other)

set ^ other

Return a new set with elements in either the set or *other* but not both.

copy()

Return a shallow copy of the set.

Note, the non-operator versions of `union()`, `intersection()`, `difference()`, and `symmetric_difference()`, `issubset()`, and `issuperset()` methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like `set('abc') & 'cbs'` in favor of the more readable `set('abc').intersection('cbs')`.

Both `set` and `frozenset` support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

Instances of `set` are compared to instances of `frozenset` based on their members. For example, `set('abc') == frozenset('abc')` returns `True` and so does `set('abc') in set([frozenset('abc')])`.

The subset and equality comparisons do not generalize to a total ordering function. For example, any two nonempty disjoint sets are not equal and are not subsets of each other, so *all* of the following return `False`: `a < b`, `a == b`, or `a > b`.

Since sets only define partial ordering (subset relationships), the output of the `list.sort()` method is undefined for lists of sets.

Set elements, like dictionary keys, must be *hashable*.

Binary operations that mix `set` instances with `frozenset` return the type of the first operand. For example: `frozenset('ab') | set('bc')` returns an instance of `frozenset`.

The following table lists operations available for `set` that do not apply to immutable instances of `frozenset`:

update(*others)

set |= other | ...

Update the set, adding elements from all others.

intersection_update(*others)

set &= other & ...

Update the set, keeping only elements found in it and all others.

difference_update(*others)

set -= other | ...

Update the set, removing elements found in others.

symmetric_difference_update(other)

set ^= other

Update the set, keeping only elements found in either set, but not in both.

add(elem)

Add element *elem* to the set.

`remove (elem)`

Remove element *elem* from the set. Raises `KeyError` if *elem* is not contained in the set.

`discard (elem)`

Remove element *elem* from the set if it is present.

`pop ()`

Remove and return an arbitrary element from the set. Raises `KeyError` if the set is empty.

`clear ()`

Remove all elements from the set.

Note, the non-operator versions of the `update()`, `intersection_update()`, `difference_update()`, and `symmetric_difference_update()` methods will accept any iterable as an argument.

Note, the *elem* argument to the `__contains__()`, `remove()`, and `discard()` methods may be a set. To support searching for an equivalent frozenset, a temporary one is created from *elem*.

4.10 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(**kwargs)
class dict(mapping, **kwargs)
class dict(iterable, **kwargs)
```

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 from the positional argument. If a key being added is already present, the value from the keyword argument replaces the value from the positional argument.

To illustrate, the following examples all return a dictionary equal to `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
```

(continues on next page)

(continued from previous page)

```
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Providing keyword arguments as in the first example only works for keys that are valid Python identifiers. Otherwise, any valid keys can be used.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

list(d)

Return a list of all the keys used in the dictionary *d*.

len(d)

Return the number of items in the dictionary *d*.

d[key]

Return the item of *d* with key *key*. Raises a *KeyError* if *key* is not in the map.

If a subclass of dict defines a method `__missing__(self, key)` and *key* is not present, the `d[key]` operation calls that method with the key *key* as argument. The `d[key]` operation then returns or raises whatever is returned or raised by the `__missing__(key)` call. No other operations or methods invoke `__missing__(self, key)`. If `__missing__(self, key)` is not defined, *KeyError* is raised. `__missing__(self, key)` must be a method; it cannot be an instance variable:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

The example above shows part of the implementation of `collections.Counter`. A different `__missing__(self, key)` method is used by `collections.defaultdict`.

d[key] = value

Set `d[key]` to *value*.

del d[key]

Remove `d[key]` from *d*. Raises a *KeyError* if *key* is not in the map.

key in d

Return True if *d* has a key *key*, else False.

key not in d

Equivalent to `not key in d`.

iter(d)

Return an iterator over the keys of the dictionary. This is a shortcut for `iter(d.keys())`.

clear()

Remove all items from the dictionary.

copy()

Return a shallow copy of the dictionary.

classmethod fromkeys (iterable[, value])

Create a new dictionary with keys from *iterable* and values set to *value*.

fromkeys () is a class method that returns a new dictionary. *value* defaults to `None`. All of the values refer to just a single instance, so it generally doesn't make sense for *value* to be a mutable object such as an empty list. To get distinct values, use a dict comprehension instead.

get (key[, default])

Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

items ()

Return a new view of the dictionary's items ((key, value) pairs). See the [documentation of view objects](#).

keys ()

Return a new view of the dictionary's keys. See the [documentation of view objects](#).

pop (key[, default])

If *key* is in the dictionary, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a `KeyError` is raised.

popitem ()

Remove and return a (key, value) pair from the dictionary. Pairs are returned in LIFO (last-in, first-out) order.

popitem () is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling *popitem ()* raises a `KeyError`.

Changed in version 3.7: LIFO order is now guaranteed. In prior versions, *popitem ()* would return an arbitrary key/value pair.

reversed (d)

Return a reverse iterator over the keys of the dictionary. This is a shortcut for `reversed (d.keys ())`.

New in version 3.8.

setdefault (key[, default])

If *key* is in the dictionary, return its value. If not, insert *key* with a value of *default* and return *default*. *default* defaults to `None`.

update ([other])

Update the dictionary with the key/value pairs from *other*, overwriting existing keys. Return `None`.

update () accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: `d.update (red=1, blue=2)`.

values ()

Return a new view of the dictionary's values. See the [documentation of view objects](#).

An equality comparison between one `dict.values ()` view and another will always return `False`. This also applies when comparing `dict.values ()` to itself:

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

Dictionaries compare equal if and only if they have the same (key, value) pairs (regardless of ordering). Order comparisons ('<', '<=', '>=' , '>') raise `TypeError`.

Dictionaries preserve insertion order. Note that updating a key does not affect the order. Keys added after deletion are inserted at the end.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

Changed in version 3.7: Dictionary order is guaranteed to be insertion order. This behavior was an implementation detail of CPython from 3.6.

Dictionaries and dictionary views are reversible.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(reversed(d))
['four', 'three', 'two', 'one']
>>> list(reversed(d.values()))
[4, 3, 2, 1]
>>> list(reversed(d.items()))
[('four', 4), ('three', 3), ('two', 2), ('one', 1)]
```

Changed in version 3.8: Dictionaries are now reversible.

See also:

`types.MappingProxyType` can be used to create a read-only view of a `dict`.

4.10.1 Dictionary view objects

The objects returned by `dict.keys()`, `dict.values()` and `dict.items()` are *view objects*. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

Dictionary views can be iterated over to yield their respective data, and support membership tests:

`len(dictview)`

Return the number of entries in the dictionary.

`iter(dictview)`

Return an iterator over the keys, values or items (represented as tuples of `(key, value)`) in the dictionary.

Keys and values are iterated over in insertion order. This allows the creation of `(value, key)` pairs using `zip()`: `pairs = zip(d.values(), d.keys())`. Another way to create the same list is `pairs = [(v, k) for (k, v) in d.items()]`.

Iterating views while adding or deleting entries in the dictionary may raise a `RuntimeError` or fail to iterate over all entries.

Changed in version 3.7: Dictionary order is guaranteed to be insertion order.

x in dictview

Return True if *x* is in the underlying dictionary's keys, values or items (in the latter case, *x* should be a (key, value) tuple).

reversed(dictview)

Return a reverse iterator over the keys, values or items of the dictionary. The view will be iterated in reverse order of the insertion.

Changed in version 3.8: Dictionary views are now reversible.

Keys views are set-like since their entries are unique and hashable. If all values are hashable, so that (key, value) pairs are unique and hashable, then the items view is also set-like. (Values views are not treated as set-like since the entries are generally not unique.) For set-like views, all of the operations defined for the abstract base class `collections.abc.Set` are available (for example, ==, <, or ^).

An example of dictionary view usage:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order (insertion order)
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```

4.11 Context Manager Types

Python's `with` statement supports the concept of a runtime context defined by a context manager. This is implemented using a pair of methods that allow user-defined classes to define a runtime context that is entered before the statement body is executed and exited when the statement ends:

`contextmanager.__enter__()`

Enter the runtime context and return either this object or another object related to the runtime context. The value returned by this method is bound to the identifier in the `as` clause of `with` statements using this context manager.

An example of a context manager that returns itself is a [file object](#). File objects return themselves from `__enter__()` to allow `open()` to be used as the context expression in a `with` statement.

An example of a context manager that returns a related object is the one returned by `decimal.localcontext()`. These managers set the active decimal context to a copy of the original decimal context and then return the copy. This allows changes to be made to the current decimal context in the body of the `with` statement without affecting code outside the `with` statement.

`contextmanager.__exit__(exc_type, exc_val, exc_tb)`

Exit the runtime context and return a Boolean flag indicating if any exception that occurred should be suppressed. If an exception occurred while executing the body of the `with` statement, the arguments contain the exception type, value and traceback information. Otherwise, all three arguments are `None`.

Returning a true value from this method will cause the `with` statement to suppress the exception and continue execution with the statement immediately following the `with` statement. Otherwise the exception continues propagating after this method has finished executing. Exceptions that occur during execution of this method will replace any exception that occurred in the body of the `with` statement.

The exception passed in should never be reraised explicitly - instead, this method should return a false value to indicate that the method completed successfully and does not want to suppress the raised exception. This allows context management code to easily detect whether or not an `__exit__()` method has actually failed.

Python defines several context managers to support easy thread synchronisation, prompt closure of files or other objects, and simpler manipulation of the active decimal arithmetic context. The specific types are not treated specially beyond their implementation of the context management protocol. See the [contextlib](#) module for some examples.

Python's [generators](#) and the [contextlib.contextmanager](#) decorator provide a convenient way to implement these protocols. If a generator function is decorated with the [contextlib.contextmanager](#) decorator, it will return a context manager implementing the necessary `__enter__()` and `__exit__()` methods, rather than the iterator produced by an undecorated generator function.

Note that there is no specific slot for any of these methods in the type structure for Python objects in the Python/C API. Extension types wanting to define these methods must provide them as a normal Python accessible method. Compared to the overhead of setting up the runtime context, the overhead of a single class dictionary lookup is negligible.

4.12 Other Built-in Types

The interpreter supports several other kinds of objects. Most of these support only one or two operations.

4.12.1 Modules

The only special operation on a module is attribute access: `m.name`, where `m` is a module and `name` accesses a name defined in `m`'s symbol table. Module attributes can be assigned to. (Note that the `import` statement is not, strictly speaking, an operation on a module object; `import foo` does not require a module object named `foo` to exist, rather it requires an (external) `definition` for a module named `foo` somewhere.)

A special attribute of every module is `__dict__`. This is the dictionary containing the module's symbol table. Modifying this dictionary will actually change the module's symbol table, but direct assignment to the `__dict__` attribute is not possible (you can write `m.__dict__['a'] = 1`, which defines `m.a` to be 1, but you can't write `m.__dict__ = {}`). Modifying `__dict__` directly is not recommended.

Modules built into the interpreter are written like this: `<module 'sys' (built-in)>`. If loaded from a file, they are written as `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`.

4.12.2 Classes and Class Instances

See objects and class for these.

4.12.3 Functions

Function objects are created by function definitions. The only operation on a function object is to call it: `func(argument-list)`.

There are really two flavors of function objects: built-in functions and user-defined functions. Both support the same operation (to call the function), but the implementation is different, hence the different object types.

See function for more information.

4.12.4 Methods

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as `append()` on lists) and class instance methods. Built-in methods are described with the types that support them.

If you access a method (a function defined in a class namespace) through an instance, you get a special object: a *bound method* (also called *instance method*) object. When called, it will add the `self` argument to the argument list. Bound methods have two special read-only attributes: `m.__self__` is the object on which the method operates, and `m.__func__` is the function implementing the method. Calling `m(arg-1, arg-2, ..., arg-n)` is completely equivalent to calling `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`.

Like function objects, bound method objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (`meth.__func__`), setting method attributes on bound methods is disallowed. Attempting to set an attribute on a method results in an `AttributeError` being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method' # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

See types for more information.

4.12.5 Code Objects

Code objects are used by the implementation to represent “pseudo-compiled” executable Python code such as a function body. They differ from function objects because they don’t contain a reference to their global execution environment. Code objects are returned by the built-in `compile()` function and can be extracted from function objects through their `__code__` attribute. See also the `code` module.

A code object can be executed or evaluated by passing it (instead of a source string) to the `exec()` or `eval()` built-in functions.

See types for more information.

4.12.6 Type Objects

Type objects represent the various object types. An object’s type is accessed by the built-in function `type()`. There are no special operations on types. The standard module `types` defines names for all standard built-in types.

Types are written like this: `<class 'int'>`.

4.12.7 The Null Object

This object is returned by functions that don’t explicitly return a value. It supports no special operations. There is exactly one null object, named `None` (a built-in name). `type(None)()` produces the same singleton.

It is written as `None`.

4.12.8 The Ellipsis Object

This object is commonly used by slicing (see slicings). It supports no special operations. There is exactly one ellipsis object, named `Ellipsis` (a built-in name). `type(Ellipsis)()` produces the `Ellipsis` singleton.

It is written as `Ellipsis` or `...`.

4.12.9 The NotImplemented Object

This object is returned from comparisons and binary operations when they are asked to operate on types they don’t support. See comparisons for more information. There is exactly one `NotImplemented` object. `type(NotImplemented)()` produces the singleton instance.

It is written as `NotImplemented`.

4.12.10 Boolean Values

Boolean values are the two constant objects `False` and `True`. They are used to represent truth values (although other values can also be considered false or true). In numeric contexts (for example when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively. The built-in function `bool()` can be used to convert any value to a Boolean, if the value can be interpreted as a truth value (see section *Truth Value Testing* above).

They are written as `False` and `True`, respectively.

4.12.11 Internal Objects

See types for this information. It describes stack frame objects, traceback objects, and slice objects.

4.13 Special Attributes

The implementation adds a few special read-only attributes to several object types, where they are relevant. Some of these are not reported by the `dir()` built-in function.

`object.__dict__`

A dictionary or other mapping object used to store an object's (writable) attributes.

`instance.__class__`

The class to which a class instance belongs.

`class.__bases__`

The tuple of base classes of a class object.

`definition.__name__`

The name of the class, function, method, descriptor, or generator instance.

`definition.__qualname__`

The *qualified name* of the class, function, method, descriptor, or generator instance.

New in version 3.3.

`class.__mro__`

This attribute is a tuple of classes that are considered when looking for base classes during method resolution.

`class.mro()`

This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class instantiation, and its result is stored in `__mro__`.

`class.__subclasses__()`

Each class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. Example:

```
>>> int.__subclasses__()  
[<class 'bool'>]
```


BUILT-IN EXCEPTIONS

In Python, all exceptions must be instances of a class that derives from `BaseException`. In a `try` statement with an `except` clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which *it* is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an “associated value” indicating the detailed cause of the error. This may be a string or a tuple of several items of information (e.g., an error code and a string explaining the code). The associated value is usually passed as arguments to the exception class’s constructor.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition “just like” the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be subclassed to define new exceptions; programmers are encouraged to derive new exceptions from the `Exception` class or one of its subclasses, and not from `BaseException`. More information on defining exceptions is available in the Python Tutorial under `tut-userexceptions`.

When raising (or re-raising) an exception in an `except` or `finally` clause `__context__` is automatically set to the last exception caught; if the new exception is not handled the traceback that is eventually displayed will include the originating exception(s) and the final exception.

When raising a new exception (rather than using a bare `raise` to re-raise the exception currently being handled), the implicit exception context can be supplemented with an explicit cause by using `from` with `raise`:

```
raise new_exc from original_exc
```

The expression following `from` must be an exception or `None`. It will be set as `__cause__` on the raised exception. Setting `__cause__` also implicitly sets the `__suppress_context__` attribute to `True`, so that using `raise new_exc from None` effectively replaces the old exception with the new one for display purposes (e.g. converting `KeyError` to `AttributeError`), while leaving the old exception available in `__context__` for introspection when debugging.

The default traceback display code shows these chained exceptions in addition to the traceback for the exception itself. An explicitly chained exception in `__cause__` is always shown when present. An implicitly chained exception in `__context__` is shown only if `__cause__` is `None` and `__suppress_context__` is `false`.

In either case, the exception itself is always shown after any chained exceptions so that the final line of the traceback always shows the last exception that was raised.

5.1 Base classes

The following exceptions are used mostly as base classes for other exceptions.

`exception BaseException`

The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that, use `Exception`). If `str()` is called on an instance of this class, the representation of the argument(s) to the instance are returned, or the empty string when there were no arguments.

`args`

The tuple of arguments given to the exception constructor. Some built-in exceptions (like `OSError`) expect a certain number of arguments and assign a special meaning to the elements of this tuple, while others are usually called only with a single string giving an error message.

`with_traceback(tb)`

This method sets `tb` as the new traceback for the exception and returns the exception object. It is usually used in exception handling code like this:

```
try:  
    ...  
except SomeException:  
    tb = sys.exc_info()[2]  
    raise OtherException(...).with_traceback(tb)
```

`exception Exception`

All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.

`exception ArithmeticError`

The base class for those built-in exceptions that are raised for various arithmetic errors: `OverflowError`, `ZeroDivisionError`, `FloatingPointError`.

`exception BufferError`

Raised when a buffer related operation cannot be performed.

`exception LookupError`

The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: `IndexError`, `KeyError`. This can be raised directly by `codecs.lookup()`.

5.2 Concrete exceptions

The following exceptions are the exceptions that are usually raised.

`exception AssertionError`

Raised when an `assert` statement fails.

`exception AttributeError`

Raised when an attribute reference (see attribute-references) or assignment fails. (When an object does not support attribute references or attribute assignments at all, `TypeError` is raised.)

`exception EOFError`

Raised when the `input()` function hits an end-of-file condition (EOF) without reading any data. (N.B.: the `io.IOBase.read()` and `io.IOBase.readline()` methods return an empty string when they hit EOF.)

`exception FloatingPointError`

Not currently used.

exception GeneratorExit

Raised when a [generator](#) or [coroutine](#) is closed; see `generator.close()` and `coroutine.close()`. It directly inherits from [BaseException](#) instead of [Exception](#) since it is technically not an error.

exception ImportError

Raised when the `import` statement has troubles trying to load a module. Also raised when the “from list” in `from ... import` has a name that cannot be found.

The `name` and `path` attributes can be set using keyword-only arguments to the constructor. When set they represent the name of the module that was attempted to be imported and the path to any file which triggered the exception, respectively.

Changed in version 3.3: Added the `name` and `path` attributes.

exception ModuleNotFoundError

A subclass of [ImportError](#) which is raised by `import` when a module could not be located. It is also raised when `None` is found in `sys.modules`.

New in version 3.6.

exception IndexError

Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not an integer, [TypeError](#) is raised.)

exception KeyError

Raised when a mapping (dictionary) key is not found in the set of existing keys.

exception KeyboardInterrupt

Raised when the user hits the interrupt key (normally Control-C or Delete). During execution, a check for interrupts is made regularly. The exception inherits from [BaseException](#) so as to not be accidentally caught by code that catches [Exception](#) and thus prevent the interpreter from exiting.

exception MemoryError

Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C’s `malloc()` function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

exception NameError

Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

exception NotImplementedError

This exception is derived from [RuntimeError](#). In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method, or while the class is being developed to indicate that the real implementation still needs to be added.

Note: It should not be used to indicate that an operator or method is not meant to be supported at all – in that case either leave the operator / method undefined or, if a subclass, set it to `None`.

Note: `NotImplementedError` and `NotImplemented` are not interchangeable, even though they have similar names and purposes. See [NotImplemented](#) for details on when to use it.

exception OSError([arg])

exception OSError(*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]])

This exception is raised when a system function returns a system-related error, including I/O failures such as “file not found” or “disk full” (not for illegal argument types or other incidental errors).

The second form of the constructor sets the corresponding attributes, described below. The attributes default to *None* if not specified. For backwards compatibility, if three arguments are passed, the *args* attribute contains only a 2-tuple of the first two constructor arguments.

The constructor often actually returns a subclass of *OSError*, as described in *OS exceptions* below. The particular subclass depends on the final *errno* value. This behaviour only occurs when constructing *OSError* directly or via an alias, and is not inherited when subclassing.

errno

A numeric error code from the C variable *errno*.

winerror

Under Windows, this gives you the native Windows error code. The *errno* attribute is then an approximate translation, in POSIX terms, of that native error code.

Under Windows, if the *winerror* constructor argument is an integer, the *errno* attribute is determined from the Windows error code, and the *errno* argument is ignored. On other platforms, the *winerror* argument is ignored, and the *winerror* attribute does not exist.

strerror

The corresponding error message, as provided by the operating system. It is formatted by the C functions *perror()* under POSIX, and *FormatMessage()* under Windows.

filename**filename2**

For exceptions that involve a file system path (such as *open()* or *os.unlink()*), *filename* is the file name passed to the function. For functions that involve two file system paths (such as *os.rename()*), *filename2* corresponds to the second file name passed to the function.

Changed in version 3.3: *EnvironmentError*, *IOError*, *WindowsError*, *socket.error*, *select.error* and *mmap.error* have been merged into *OSError*, and the constructor may return a subclass.

Changed in version 3.4: The *filename* attribute is now the original file name passed to the function, instead of the name encoded to or decoded from the filesystem encoding. Also, the *filename2* constructor argument and attribute was added.

exception OverflowError

Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for integers (which would rather raise *MemoryError* than give up). However, for historical reasons, *OverflowError* is sometimes raised for integers that are outside a required range. Because of the lack of standardization of floating point exception handling in C, most floating point operations are not checked.

exception RecursionError

This exception is derived from *RuntimeError*. It is raised when the interpreter detects that the maximum recursion depth (see *sys.getrecursionlimit()*) is exceeded.

New in version 3.5: Previously, a plain *RuntimeError* was raised.

exception ReferenceError

This exception is raised when a weak reference proxy, created by the *weakref.proxy()* function, is used to access an attribute of the referent after it has been garbage collected. For more information on weak references, see the *weakref* module.

exception RuntimeError

Raised when an error is detected that doesn’t fall in any of the other categories. The associated value is a string indicating what precisely went wrong.

exception StopIteration

Raised by built-in function `next()` and an *iterator*'s `__next__()` method to signal that there are no further items produced by the iterator.

The exception object has a single attribute `value`, which is given as an argument when constructing the exception, and defaults to `None`.

When a *generator* or *coroutine* function returns, a new `StopIteration` instance is raised, and the value returned by the function is used as the `value` parameter to the constructor of the exception.

If a generator code directly or indirectly raises `StopIteration`, it is converted into a `RuntimeError` (retaining the `StopIteration` as the new exception's cause).

Changed in version 3.3: Added `value` attribute and the ability for generator functions to use it to return a value.

Changed in version 3.5: Introduced the `RuntimeError` transformation via `from __future__ import generator_stop`, see [PEP 479](#).

Changed in version 3.7: Enable [PEP 479](#) for all code by default: a `StopIteration` error raised in a generator is transformed into a `RuntimeError`.

exception StopAsyncIteration

Must be raised by `__anext__()` method of an *asynchronous iterator* object to stop the iteration.

New in version 3.5.

exception SyntaxError

Raised when the parser encounters a syntax error. This may occur in an `import` statement, in a call to the built-in functions `exec()` or `eval()`, or when reading the initial script or standard input (also interactively).

Instances of this class have attributes `filename`, `lineno`, `offset` and `text` for easier access to the details. `str()` of the exception instance returns only the message.

exception IndentationError

Base class for syntax errors related to incorrect indentation. This is a subclass of `SyntaxError`.

exception TabError

Raised when indentation contains an inconsistent use of tabs and spaces. This is a subclass of `IndentationError`.

exception SystemError

Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms).

You should report this to the author or maintainer of your Python interpreter. Be sure to report the version of the Python interpreter (`sys.version`; it is also printed at the start of an interactive Python session), the exact error message (the exception's associated value) and if possible the source of the program that triggered the error.

exception SystemExit

This exception is raised by the `sys.exit()` function. It inherits from `BaseException` instead of `Exception` so that it is not accidentally caught by code that catches `Exception`. This allows the exception to properly propagate up and cause the interpreter to exit. When it is not handled, the Python interpreter exits; no stack traceback is printed. The constructor accepts the same optional argument passed to `sys.exit()`. If the value is an integer, it specifies the system exit status (passed to C's `exit()` function); if it is `None`, the exit status is zero; if it has another type (such as a string), the object's value is printed and the exit status is one.

A call to `sys.exit()` is translated into an exception so that clean-up handlers (`finally` clauses of `try` statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The `os._exit()` function can be used if it is absolutely positively necessary to exit immediately (for example, in the child process after a call to `os.fork()`).

code

The exit status or error message that is passed to the constructor. (Defaults to None.)

exception TypeError

Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

This exception may be raised by user code to indicate that an attempted operation on an object is not supported, and is not meant to be. If an object is meant to support a given operation but has not yet provided an implementation, *NotImplementedError* is the proper exception to raise.

Passing arguments of the wrong type (e.g. passing a *list* when an *int* is expected) should result in a *TypeError*, but passing arguments with the wrong value (e.g. a number outside expected boundaries) should result in a *ValueError*.

exception UnboundLocalError

Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of *NameError*.

exception UnicodeError

Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of *ValueError*.

UnicodeError has attributes that describe the encoding or decoding error. For example, `err.object[err.start:err.end]` gives the particular invalid input that the codec failed on.

encoding

The name of the encoding that raised the error.

reason

A string describing the specific codec error.

object

The object the codec was attempting to encode or decode.

start

The first index of invalid data in *object*.

end

The index after the last invalid data in *object*.

exception UnicodeEncodeError

Raised when a Unicode-related error occurs during encoding. It is a subclass of *UnicodeError*.

exception UnicodeDecodeError

Raised when a Unicode-related error occurs during decoding. It is a subclass of *UnicodeError*.

exception UnicodeTranslateError

Raised when a Unicode-related error occurs during translating. It is a subclass of *UnicodeError*.

exception ValueError

Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as *IndexError*.

exception ZeroDivisionError

Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

The following exceptions are kept for compatibility with previous versions; starting from Python 3.3, they are aliases of *OSError*.

exception EnvironmentError

exception IOError

exception WindowsError

Only available on Windows.

5.2.1 OS exceptions

The following exceptions are subclasses of [OSError](#), they get raised depending on the system error code.

exception BlockingIOError

Raised when an operation would block on an object (e.g. socket) set for non-blocking operation. Corresponds to `errno EAGAIN`, `EALREADY`, `EWOULD_BLOCK` and `EINPROGRESS`.

In addition to those of [OSError](#), [BlockingIOError](#) can have one more attribute:

characters_written

An integer containing the number of characters written to the stream before it blocked. This attribute is available when using the buffered I/O classes from the [io](#) module.

exception ChildProcessError

Raised when an operation on a child process failed. Corresponds to `errno ECHILD`.

exception ConnectionError

A base class for connection-related issues.

Subclasses are [BrokenPipeError](#), [ConnectionAbortedError](#), [ConnectionRefusedError](#) and [ConnectionResetError](#).

exception BrokenPipeError

A subclass of [ConnectionError](#), raised when trying to write on a pipe while the other end has been closed, or trying to write on a socket which has been shutdown for writing. Corresponds to `errno EPIPE` and `ESHUTDOWN`.

exception ConnectionAbortedError

A subclass of [ConnectionError](#), raised when a connection attempt is aborted by the peer. Corresponds to `errno ECONNABORTED`.

exception ConnectionRefusedError

A subclass of [ConnectionError](#), raised when a connection attempt is refused by the peer. Corresponds to `errno ECONNREFUSED`.

exception ConnectionResetError

A subclass of [ConnectionError](#), raised when a connection is reset by the peer. Corresponds to `errno ECONNRESET`.

exception FileExistsError

Raised when trying to create a file or directory which already exists. Corresponds to `errno EEXIST`.

exception FileNotFoundError

Raised when a file or directory is requested but doesn't exist. Corresponds to `errno ENOENT`.

exception InterruptedError

Raised when a system call is interrupted by an incoming signal. Corresponds to `errno EINTR`.

Changed in version 3.5: Python now retries system calls when a syscall is interrupted by a signal, except if the signal handler raises an exception (see [PEP 475](#) for the rationale), instead of raising [InterruptedError](#).

exception IsADirectoryError

Raised when a file operation (such as `os.remove()`) is requested on a directory. Corresponds to `errno EISDIR`.

exception NotADirectoryError

Raised when a directory operation (such as `os.listdir()`) is requested on something which is not a directory. Corresponds to `errno ENOTDIR`.

exception `PermissionError`

Raised when trying to run an operation without the adequate access rights - for example filesystem permissions.
Corresponds to `errno EACCES` and `EPERM`.

exception `ProcessLookupError`

Raised when a given process doesn't exist. Corresponds to `errno ESRCH`.

exception `TimeoutError`

Raised when a system function timed out at the system level. Corresponds to `errno ETIMEDOUT`.

New in version 3.3: All the above `OSError` subclasses were added.

See also:

[PEP 3151](#) - Reworking the OS and IO exception hierarchy

5.3 Warnings

The following exceptions are used as warning categories; see the [Warning Categories](#) documentation for more details.

exception `Warning`

Base class for warning categories.

exception `UserWarning`

Base class for warnings generated by user code.

exception `DeprecationWarning`

Base class for warnings about deprecated features when those warnings are intended for other Python developers.

exception `PendingDeprecationWarning`

Base class for warnings about features which are obsolete and expected to be deprecated in the future, but are not deprecated at the moment.

This class is rarely used as emitting a warning about a possible upcoming deprecation is unusual, and `DeprecationWarning` is preferred for already active deprecations.

exception `SyntaxWarning`

Base class for warnings about dubious syntax.

exception `RuntimeWarning`

Base class for warnings about dubious runtime behavior.

exception `FutureWarning`

Base class for warnings about deprecated features when those warnings are intended for end users of applications that are written in Python.

exception `ImportWarning`

Base class for warnings about probable mistakes in module imports.

exception `UnicodeWarning`

Base class for warnings related to Unicode.

exception `BytesWarning`

Base class for warnings related to `bytes` and `bytearray`.

exception `ResourceWarning`

Base class for warnings related to resource usage. Ignored by the default warning filters.

New in version 3.2.

5.4 Exception hierarchy

The class hierarchy for built-in exceptions is:

```

BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        |    +-- FloatingPointError
        |    +-- OverflowError
        |    +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
        |    +-- ModuleNotFoundError
    +-- LookupError
        |    +-- IndexError
        |    +-- KeyError
    +-- MemoryError
    +-- NameError
        |    +-- UnboundLocalError
    +-- OSError
        |    +-- BlockingIOError
        |    +-- ChildProcessError
        |    +-- ConnectionError
            |    +-- BrokenPipeError
            |    +-- ConnectionAbortedError
            |    +-- ConnectionRefusedError
            |    +-- ConnectionResetError
        |    +-- FileExistsError
        |    +-- FileNotFoundError
        |    +-- InterruptedError
        |    +-- IsADirectoryError
        |    +-- NotADirectoryError
        |    +-- PermissionError
        |    +-- ProcessLookupError
        |    +-- TimeoutError
    +-- ReferenceError
    +-- RuntimeError
        |    +-- NotImplementedError
        |    +-- RecursionError
    +-- SyntaxError
        |    +-- IndentationError
            |        +-- TabError
    +-- SystemError
    +-- TypeError
    +-- ValueError
        |    +-- UnicodeError
            |        +-- UnicodeDecodeError
            |        +-- UnicodeEncodeError
            |        +-- UnicodeTranslateError

```

(continues on next page)

(continued from previous page)

```
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning
```



Astronomy ESFRI & Research Infrastructure Cluster
ASTERICS - 653477



2nd ASTERICS-OBELICS International School

4-8 June 2018, Annecy, France.



H2020-Astronomy ESFRI and Research Infrastructure Cluster (Grant Agreement number: 653477).

PYTHON LIBRARIES

Tamás Gál
tamas.gal@fau.de



<https://github.com/tamasgal>



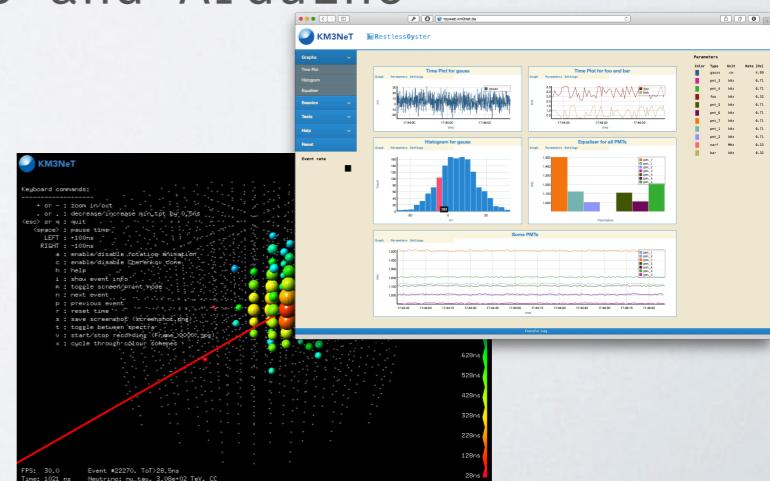
@tamasgal

OVERVIEW

- Who is this clown?
 - Python Introduction
 - Basic Python Internals
 - Libraries and Tools for Scientific Computing
 - NumPy
 - Numba
 - NumExpr
 - SciPy
 - AstroPy
 - Pandas
 - SymPy
 - Matplotlib
 - Jupyter
 - IPython
-
- { Make it faster!
- { Tools for scientists!

WHO IS THIS CLOWN?

- Tamás Gál, born 1985 in Debrecen (Hungary)
- PhD candidate in astro particle physics at Erlangen Centre for Astroparticle Physics (ECAP) working on the KM3NeT project
- Programming background:
 - Coding enthusiast since ~1993
 - First real application written in Amiga Basic (toilet manager, tons of GOTOs ;)
 - Python, Julia, JavaScript and C/C++/Obj-C for work
 - Haskell for fun
 - Earlier also Java, Perl, PHP, Delphi, MATLAB, whatsoever...
 - I also like playing around with integrated circuits and Arduino
- Some related projects:
 - KM3Pipe (analysis framework in the KM3NeT experiment),
 - RainbowAlga (interactive 3D neutrino event display),
 - ROyWeb (interactive realtime visualisation/graphing)

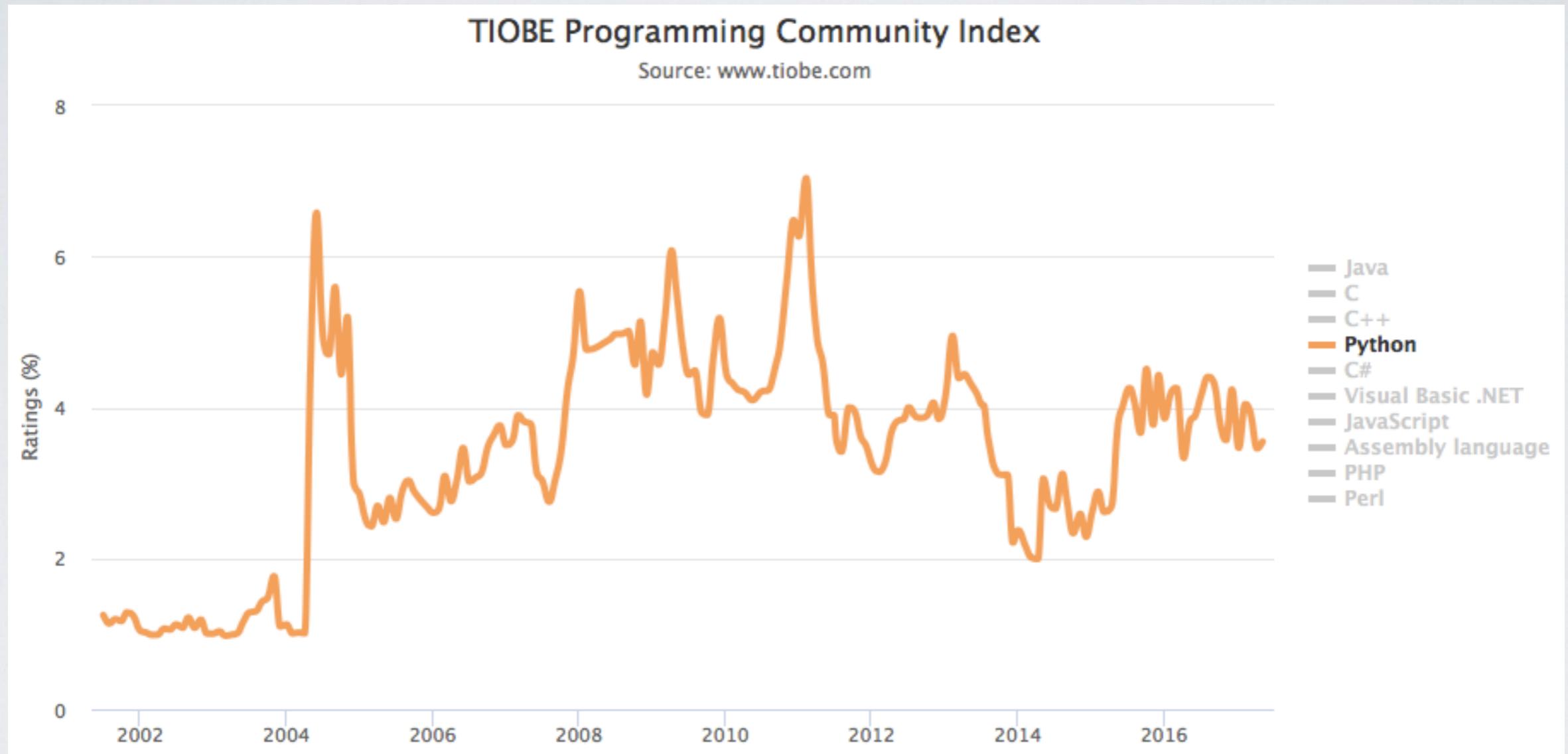


PYTHON

BRIEF HISTORY OF PYTHON

- Rough idea in the late 1980s
- Meant to descend the ABC language
- First line of code in December 1989 by Guido van Rossum
- Python 2.0 in October 2000
- Python 3.0 in December 2008

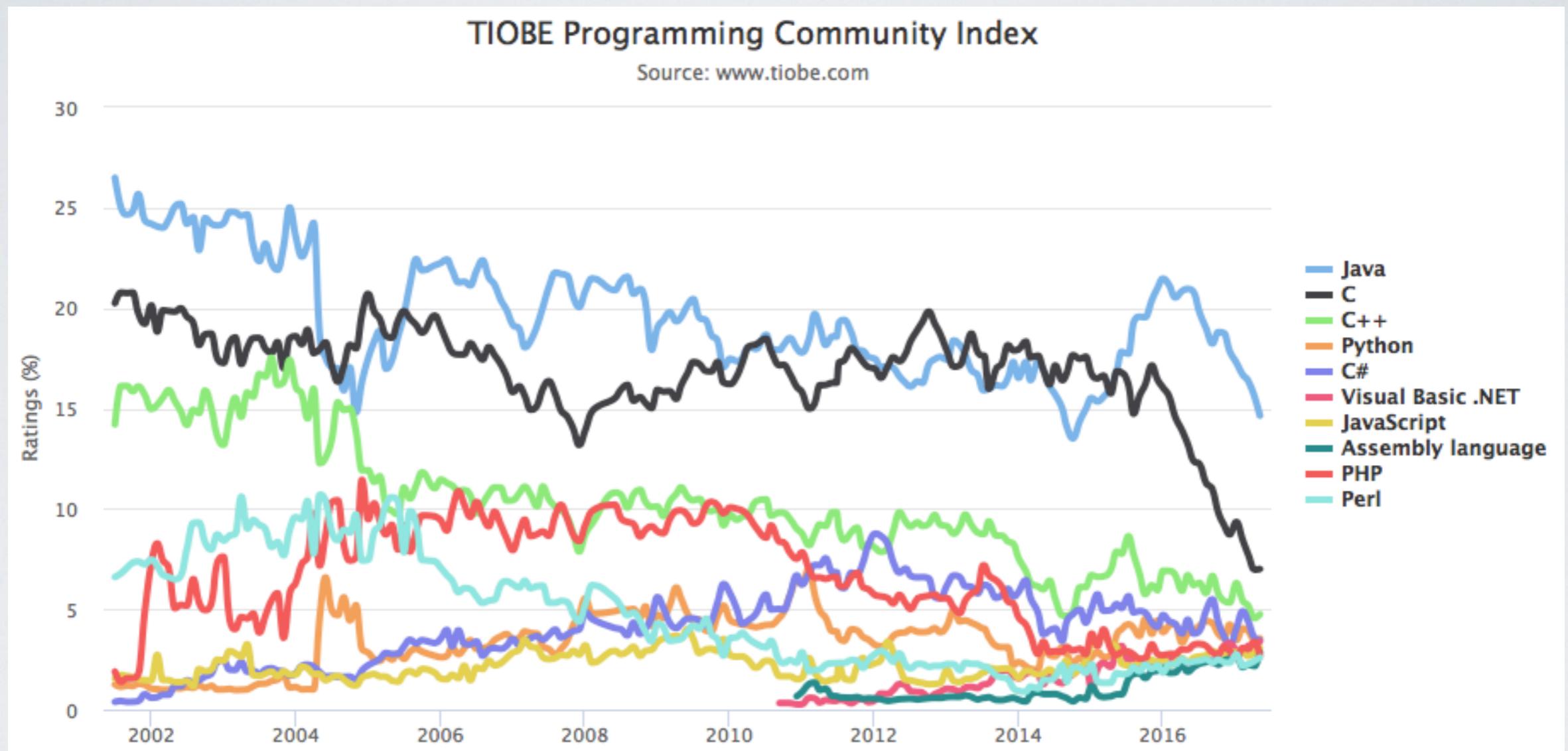
PYTHON'S POPULARITY



“Programming language of the year” in 2007 and 2010.

POPULAR LANGUAGES

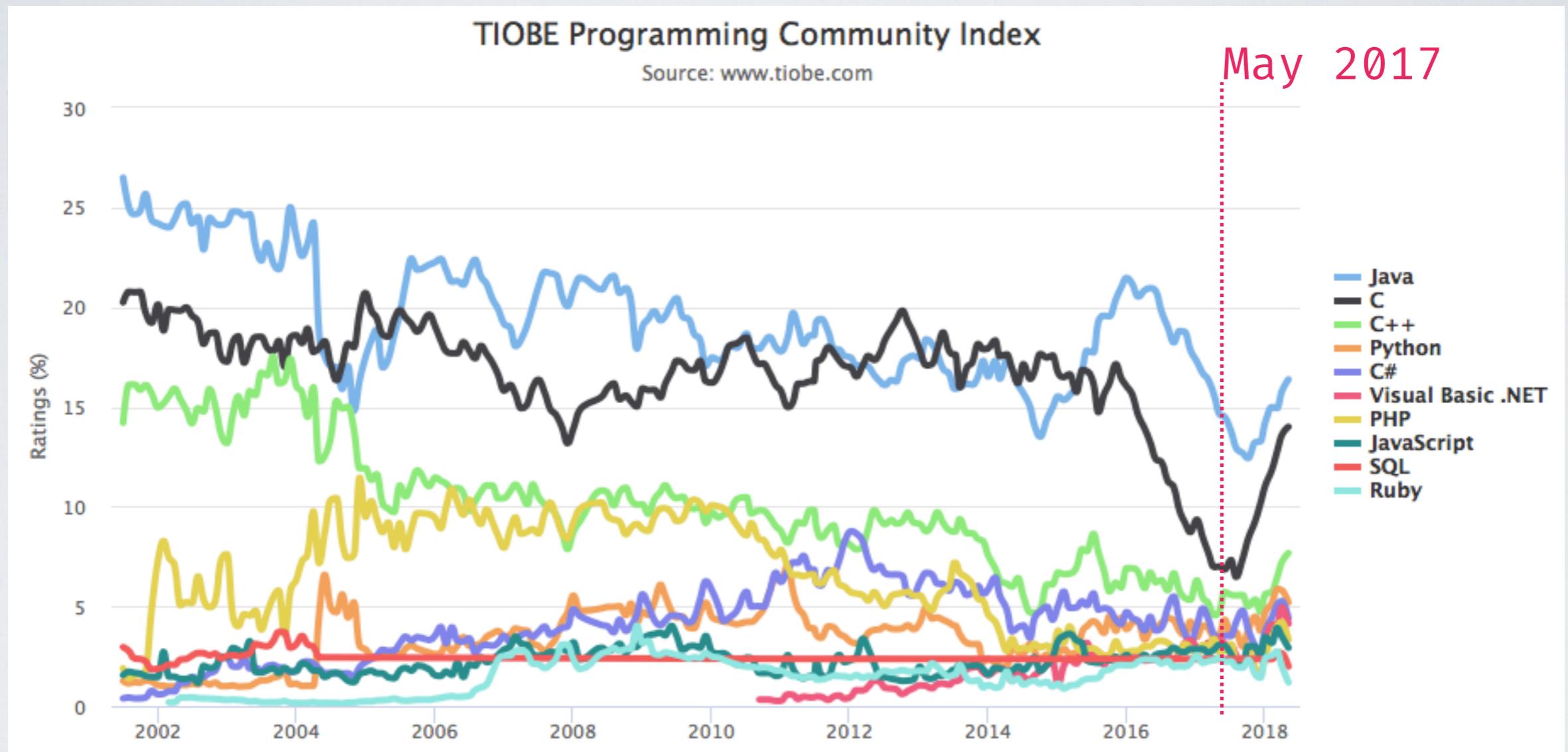
(MAY 2017)



Python is the fourth most popular language and rocks the top 10 since 2003.

POPULAR LANGUAGES

(MAY 2018)



Python is still the fourth most popular language and rocks the top 10 since 2003.

YOUR JOURNEY THROUGH PYTHON?

(JUST A VERY ROUGH GUESS, NOT A MEAN GAME)

Raise your hand and keep it up until you answer a question with “no”.

- Have you ever launched the Python interpreter?
- Wrote for/while-loops or if/else statements?
- ...your own functions?
- ...classes?
- ...list/dict/set comprehensions?
- Do you know what a generator is?
- Have you ever implemented a decorator?
- ...a metaclass?
- ...a C-extension?
- Do you know and can you explain the output of the following line?

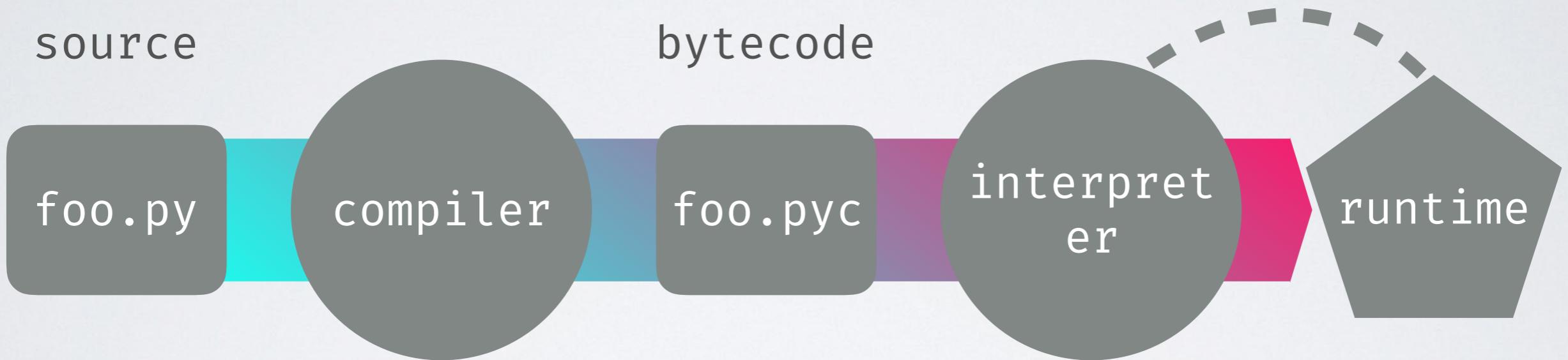
```
print(5 is 7 - 2, 300 is 302 - 2)
```



BASIC PYTHON INTERNALS

to understand the performance issues

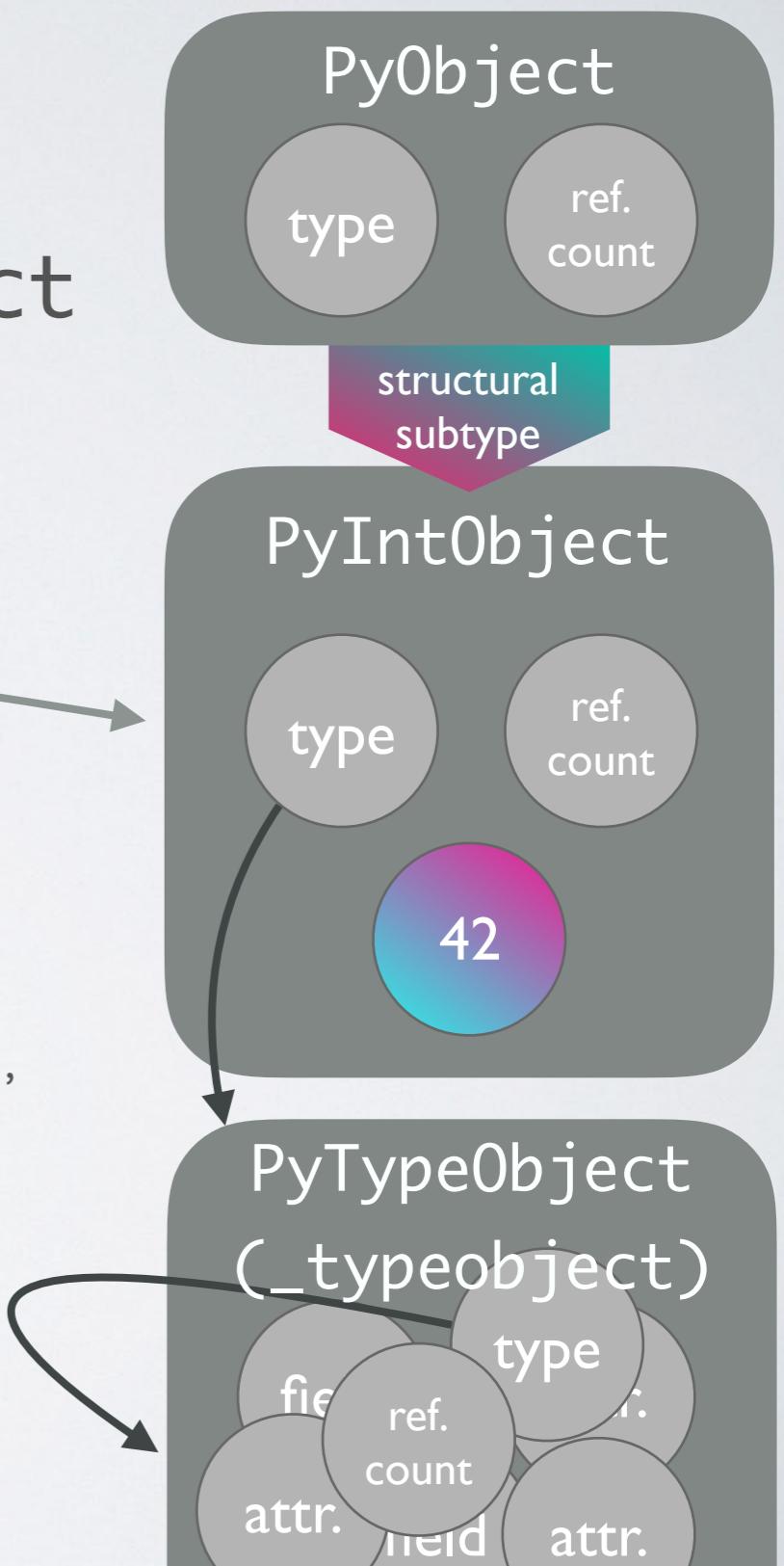
FROM SOURCE TO RUNTIME



DATA IN PYTHON

- Every piece of data is a PyObject

```
>>> dir(42)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__',
'__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__',
'__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length',
'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real',
'to_bytes']
```



THE TYPE OF A PyObject

“An object has a ‘type’ that determines what it represents and what kind of data it contains.

An object’s type is fixed when it is created. Types themselves are represented as objects. The type itself has a type pointer pointing to the object representing the type ‘type’, which contains a pointer to itself!”

– object.h

YOUR BEST FRIEND AND WORST ENEMY: GIL - Global Interpreter Lock

- The GIL prevents parallel execution of (Python) bytecode
- Even though Python has real threads, they never execute code at the same time
- Context switching between threads creates overhead (the user cannot control thread-priority)
- Threads perform pretty bad on CPU bound tasks
- They do a great job speeding up I/O heavy tasks

THREADS AND CPU BOUND TASKS

single thread:

```
N = 100000000

def count(n):
    while n != 0: n -=1

%time count(N)

CPU times: user 5.59 s, sys: 32.5 ms, total: 5.62 s
Wall time: 7.71 s
```

two threads:

```
from threading import Thread

def count_threaded(n):
    t1 = Thread(target=count, args=(N/2,))
    t2 = Thread(target=count, args=(N/2,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()

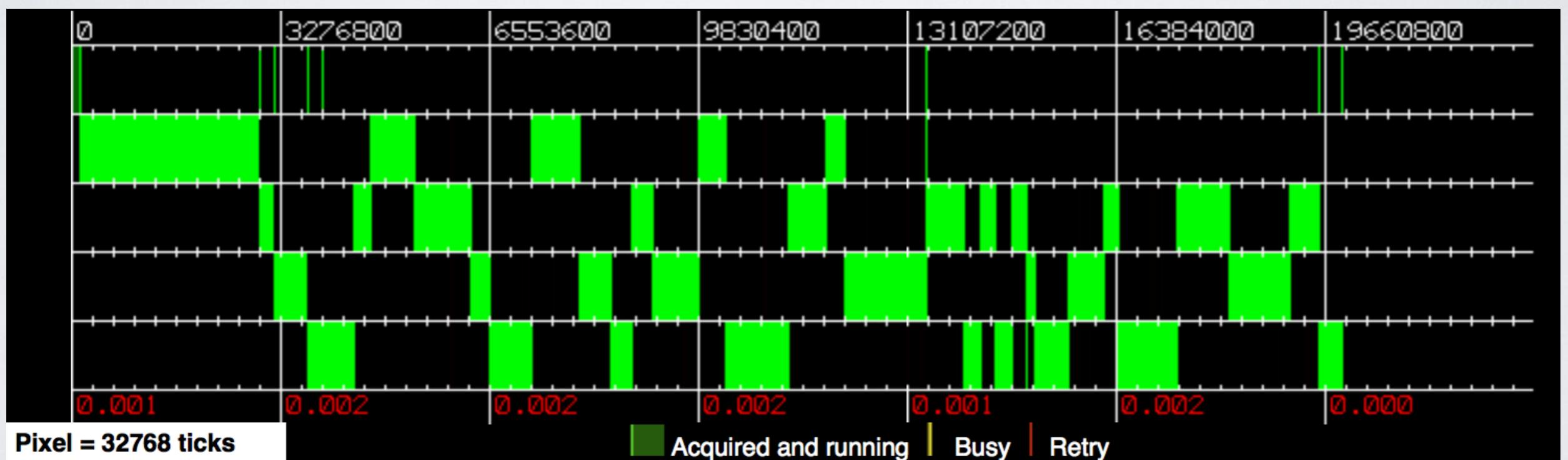
%time count_threaded(N)

CPU times: user 7.18 s, sys: 31 ms, total: 7.21 s
Wall time: 9.01 s
```

This is probably not really what you expected...

THREADS FIGHTING FOR THE GIL

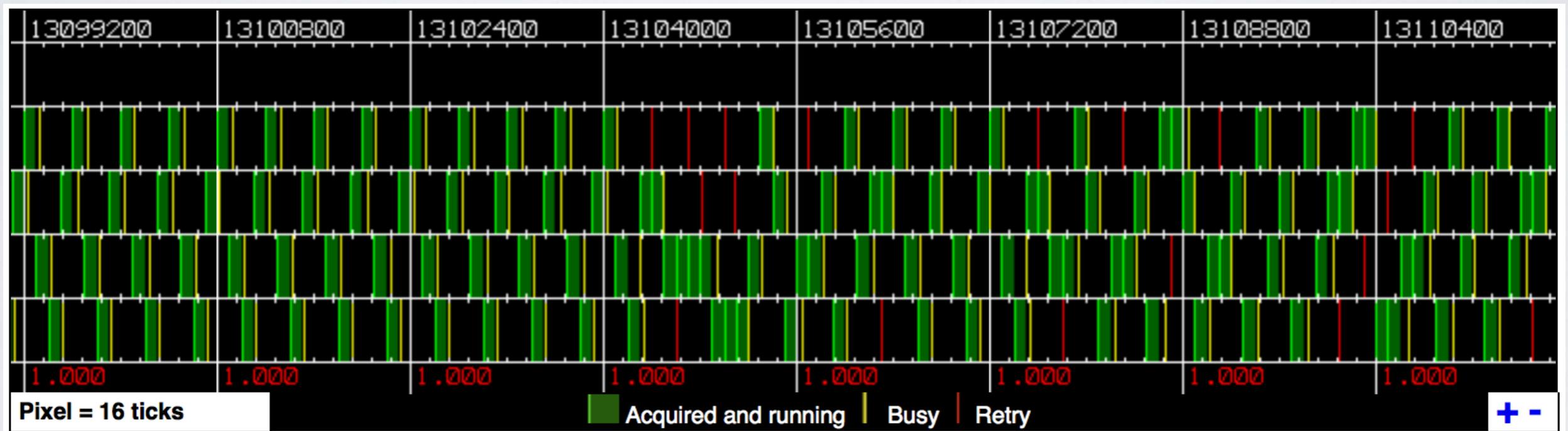
OS X: 4 threads on 1 CPU (Python 2.6)



By David M Beazley: <http://dabeaz.com/GIL/gilvis>

THREADS FIGHTING FOR THE GIL

OS X: 4 threads on 4 CPUs (Python 2.6)



By David M Beazley: <http://dabeaz.com/GIL/gilvis>

OK, huge overhead for every single object,
no real parallel execution of code ...

**How should Python ever compete with all
those super fast C/Fortran libraries?**

C-extensions and interfacing C/Fortran!

Those can release the GIL and do the heavy stuff in the background.

A DUMB SPEED COMPARISON

CALCULATING THE MEAN OF 1000000 RANDOM NUMBERS

pure Python:

```
def mean(numbers):
    return sum(numbers)/len(numbers)

numbers = list(range(1000000))
%timeit mean(numbers)

8.59 ms ± 234 µs per loop
```

NumPy (~13x faster):

```
numbers = np.random.random(1000000)
%timeit np.mean(numbers)

638 µs ± 38.3 µs per loop
```

Numba (~8x faster):

```
@nb.jit
def numba_mean(numbers):
    s = 0
    N = len(numbers)
    for i in range(N):
        s += numbers[i]
    return s/N

numbers = np.random.random(1000000)
%timeit numba_mean(numbers)

1.1 ms ± 6.64 µs per loop
```

Julia (~16x faster):

```
numbers = rand(1000000)
@benchmark mean(numbers)

BenchmarkTools.Trial:
  memory estimate: 16 bytes
  allocs estimate: 1
  -----
  minimum time:      464.824 µs (0.00% GC)
  median time:       524.386 µs (0.00% GC)
  mean time:         544.573 µs (0.00% GC)
  maximum time:     2.095 ms (0.00% GC)
  -----
  samples:           8603
  evals/sample:      1
```

CRAZY LLVM COMPILER OPTIMISATIONS

SUMMING UP NUMBERS FROM 0 TO N=100,000,000

pure Python:

```
def simple_sum(N):
    s = 0
    for i in range(N):
        s += i
    return s

%time simple_sum(N)

CPU times: user 7.13 s, sys: 103 ms, total: 7.23 s
Wall time: 7.43 s

4999999950000000
```

NumPy (~80x faster):

```
np_numbers = np.array(range(N))

%time np.sum(np_numbers)

CPU times: user 84 ms, sys: 2.65 ms, total: 86.6 ms
Wall time: 91.1 ms

4999999950000000
```

Numba (~300000x faster):

```
@nb.jit
def simple_sum(N):
    s = 0
    for i in range(N):
        s += i
    return s

%time numba_sum(N)

CPU times: user 11 µs, sys: 3 µs, total: 14 µs
Wall time: 21.9 µs

4999999950000000
```

Julia (~700000x faster):

```
function simple_sum(N)
    s = 0
    for i ∈ 1:N
        s += i
    end
    return s
end

simple_sum (generic function)

@time simple_sum(N)

0.000002 seconds (5

4999999950000000

Source line: 3
testq %rdi, %rdi
jle L32
leaq -1(%rdi), %rax
leaq -2(%rdi), %rcx
mulq %rcx
shldq $63, %rax, %rdx
leaq -1(%rdx,%rdi,2), %rax

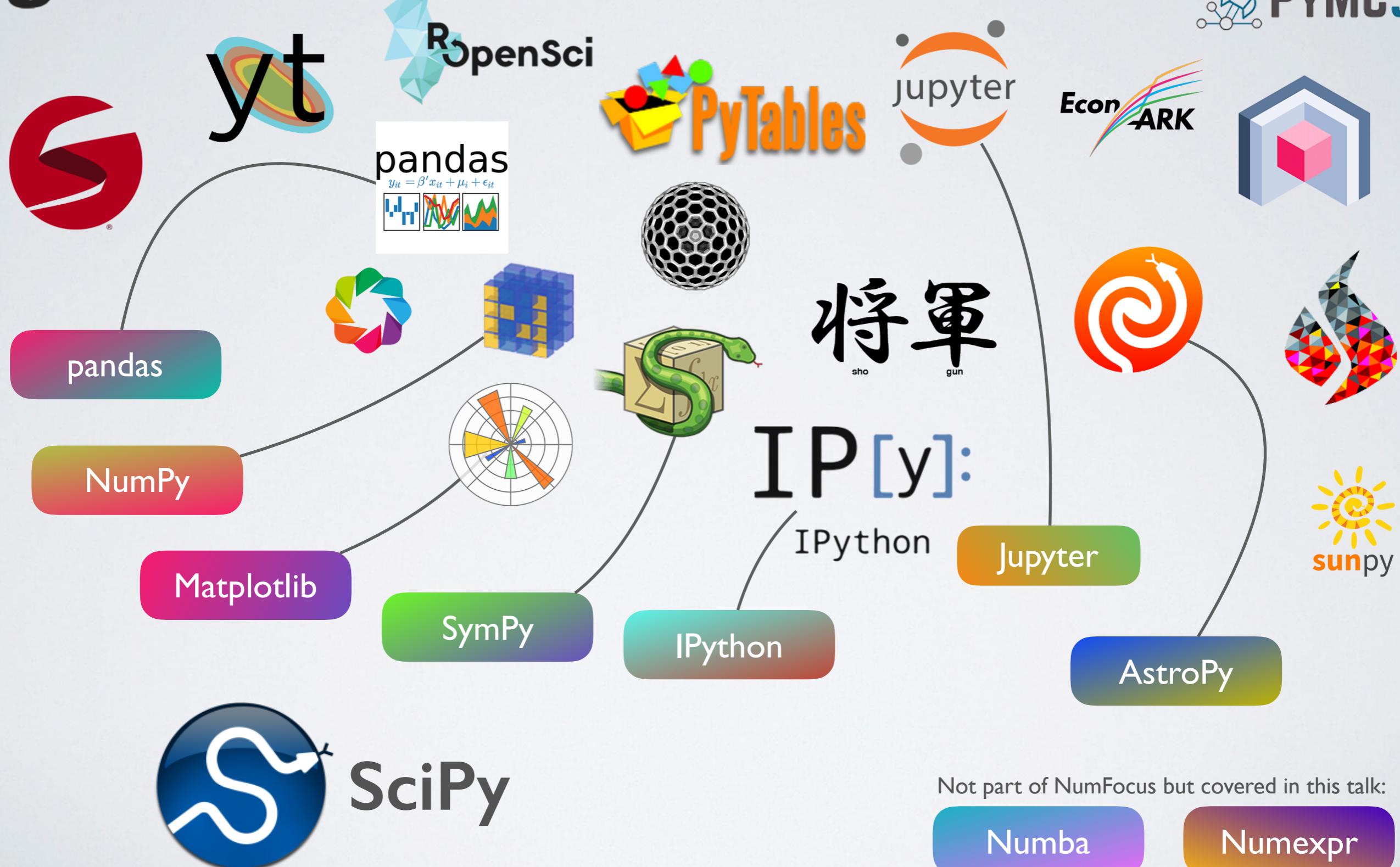
Source line: 6
L32:
popq %rbp
retq
nopw %cs:(%rax,%rax)
```

PYTHON LIBRARIES

for scientific computing



NUMFOCUS
OPEN CODE = BETTER SCIENCE





SCIPY

Scientific Computing Tools for Python

THE SCIPY STACK

- Core packages
 - SciPy Library: numerical algorithms, signal processing, optimisation, statistics etc.
 - NumPy
 - Matplotlib: 2D/3D plotting library
 - pandas: high performance, easy to use data structures
 - SymPy: symbolic mathematics and computer algebra
 - IPython: a rich interactive interface to process data and test ideas
 - Jupyter: notebooks to document and code at the same time
 - nose: testing framework for Python code
- Other packages:
 - Chaco, Mayavi, Cython, Scikits (scikit-learn, scikit-image), h5py, PyTables and much more

<https://www.scipy.org>

SCIPY CORE LIBRARY

- Clustering package (`scipy.cluster`)
- Constants (`scipy.constants`)
- Discrete Fourier transforms (`scipy.fftpack`)
- Integration and ODEs (`scipy.integrate`)
- Interpolation (`scipy.interpolate`)
- Input and output (`scipy.io`)
- Linear algebra (`scipy.linalg`)
- Miscellaneous routines (`scipy.misc`)
- Multi-dimensional image processing (`scipy.ndimage`)
- Orthogonal distance regression (`scipy.odr`)
- Optimization and root finding (`scipy.optimize`)
- Signal processing (`scipy.signal`)
- Sparse matrices (`scipy.sparse`)
- Sparse linear algebra (`scipy.sparse.linalg`)
- Compressed Sparse Graph Routines
(`scipy.sparse.csgraph`)
- Spatial algorithms and data structures (`scipy.spatial`)
- Special functions (`scipy.special`)
- Statistical functions (`scipy.stats`)
- Statistical functions for masked arrays (`scipy.stats.mstats`)

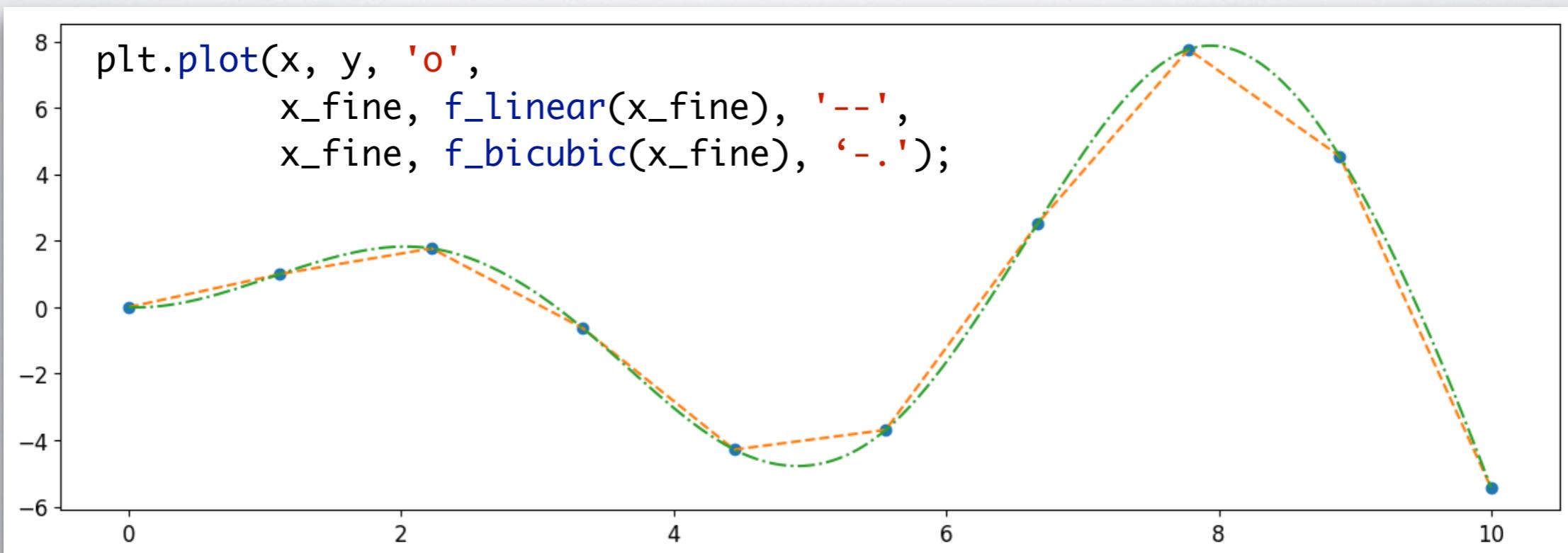
SCIPY INTERPOLATE

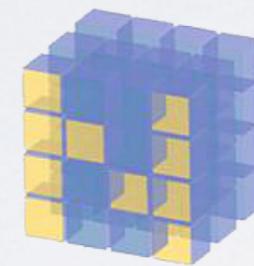
```
from scipy import interpolate

x = np.linspace(0, 10, 10)
y = np.sin(x)

x_fine = np.linspace(0, 10, 500)

f_linear = interpolate.interp1d(x, y, kind='linear')
f_bicubic = interpolate.interp1d(x, y, kind='cubic')
```





NUMPY
Numerical Python

NUMPY

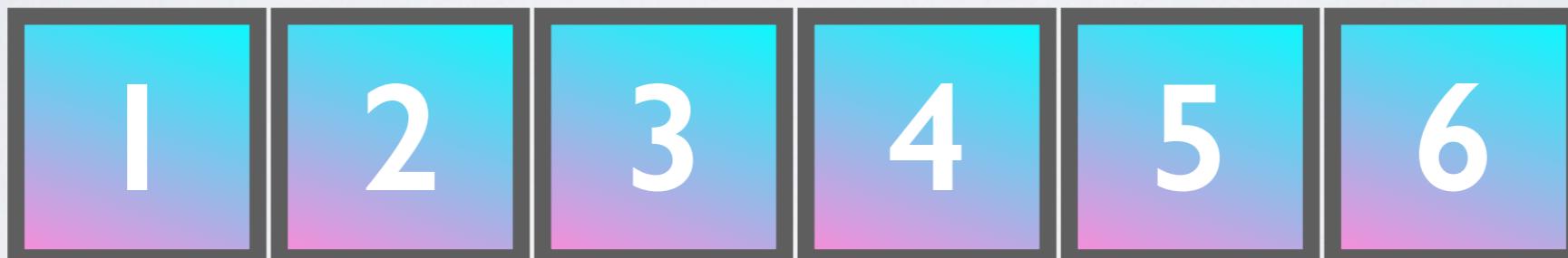
NumPy is the fundamental package for scientific computing with Python.

- gives us a powerful N-dimensional array object: `ndarray`
- broadcasting functions
- tools for integrating C/C++ and Fortran
- linear algebra, Fourier transform and random number capabilities
- most of the scientific libraries build upon NumPy

NUMPY: ndarray

```
a = np.arange(6)  
a  
  
array([0, 1, 2, 3, 4, 5])
```

ndim: 1
shape: (6,)



Continuous array in memory with a fixed type,
no pointer madness!

C/Fortran compatible memory layout,
so they can be passed to those
without any further efforts.

NUMPY: ARRAY OPERATIONS AND ufuncs

```
a * 23
```

```
array([ 0, 23, 46, 69, 92, 115])
```

```
a**a
```

```
array([ 1, 1, 4, 27, 256, 3125])
```

easy and intuitive
element-wise
operations

a ufunc, which can operate both on scalars and arrays (element-wise)

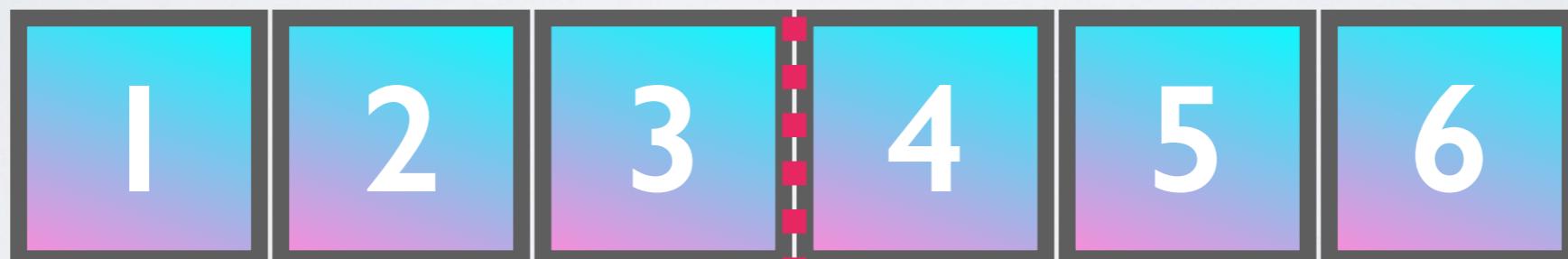
```
np.exp(a)
```

```
array([ 1. , 2.71828183, 7.3890561 , 20.08553692,
       54.59815003, 148.4131591 ])
```

RESHAPING ARRAYS

```
a = np.arange(6)  
a  
  
array([0, 1, 2, 3, 4, 5])
```

ndim: 1
shape: (6,)



a[0]

a[1]

```
a.reshape(2, 3)  
  
array([[0, 1, 2],  
       [3, 4, 5]])
```

No rearrangement of the elements
but setting the iterator limits internally!

RESHAPING ARRAYS IS CHEAP

```
a = np.arange(10000000)  
%timeit b = a.reshape(100, 5000, 20)  
563 ns ± 8.18 ns per loop (mean ± std.
```

Don't worry, we will discover NumPy in the hands-on workshop!



The logo for matplotlib consists of the word "matplotlib" in a large, bold, blue sans-serif font. The letter "o" is replaced by a circular pie chart with several colored segments (orange, yellow, green, blue) and radial grid lines.

MATPLOTLIB

A Python plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments.

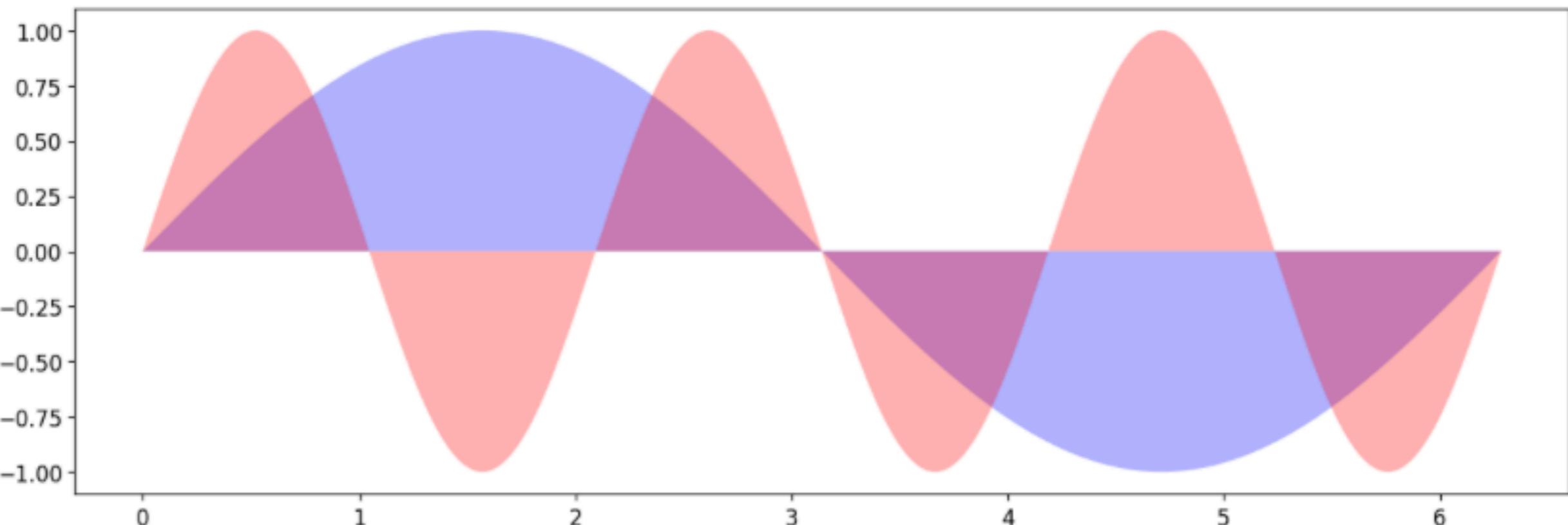
- Integrates well with IPython and Jupyter
- Plots, histograms, power spectra, bar charts, error chars, scatterplots, etc. with an easy to use API
- Full control of line styles, font properties, axes properties etc.
- The easiest way to get started is browsing its wonderful gallery full of thumbnails and copy&paste examples:
<http://matplotlib.org/gallery.html>

MATPLOTLIB EXAMPLE

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi, 500)
y1 = np.sin(x)
y2 = np.sin(3 * x)

fig, ax = plt.subplots()
ax.fill(x, y1, 'b', x, y2, 'r', alpha=0.3)
plt.show()
```

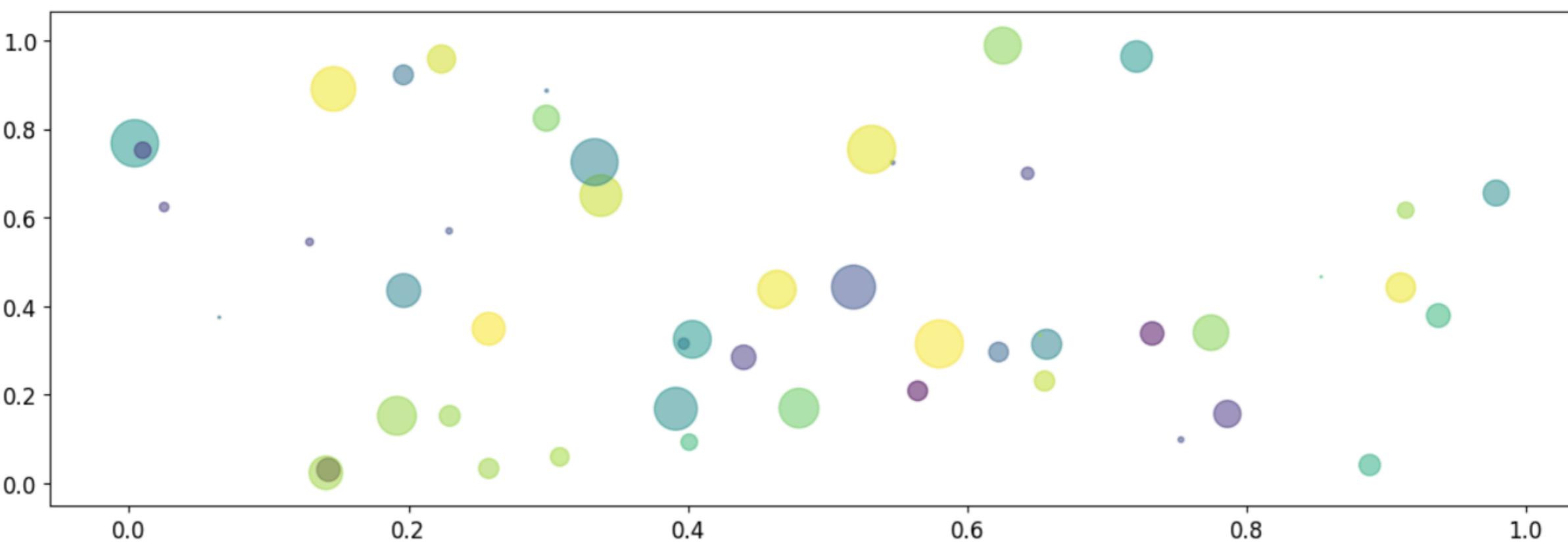


MATPLOTLIB EXAMPLE

```
import numpy as np
import matplotlib.pyplot as plt

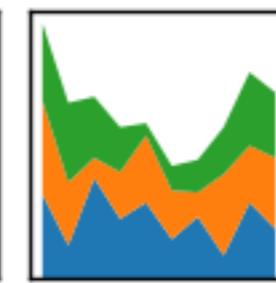
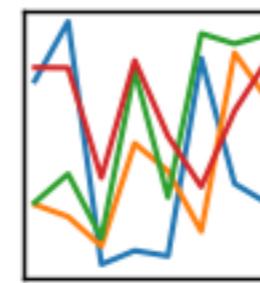
N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2

plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



PANDAS

A Python Data Analysis Library inspired by data frames in R:

- gives us a powerful data structure: `DataFrame`
- database-like handling of data
- integrates well with NumPy
- wraps the Matplotlib API (which can also cause troubles ;)
- has a huge number of I/O related functions to parse data:
`CSV`, `HDF5`, `SQL`, `Feather`, `JSON`, `HTML`, `Excel`, and more...

THE DataFrame

A table-like structure, where you can access elements by row and column.

```
hits = pd.read_hdf("event_file.h5", "events/23")
hits.head(3)
```

	channel_id	dom_id	event_id	id	pmt_id	time	tot	triggered
0	25	808430036		0	0	30652287	21	0
1	18	808430036		0	0	30656200	16	0
2	15	808430449		0	0	30648451	26	0

THE DataFrame

Lots of functions to allow filtering, manipulating and aggregating the data to fit your needs.

```
▼ active_doms = hits.pivot_table(index='event_id',
                                   values='dom_id',
                                   aggfunc=lambda x: set(x))
```

Don't worry, we will discover Pandas in the hands-on workshop!

sponsored by
CONTINUUM®
ANALYTICS



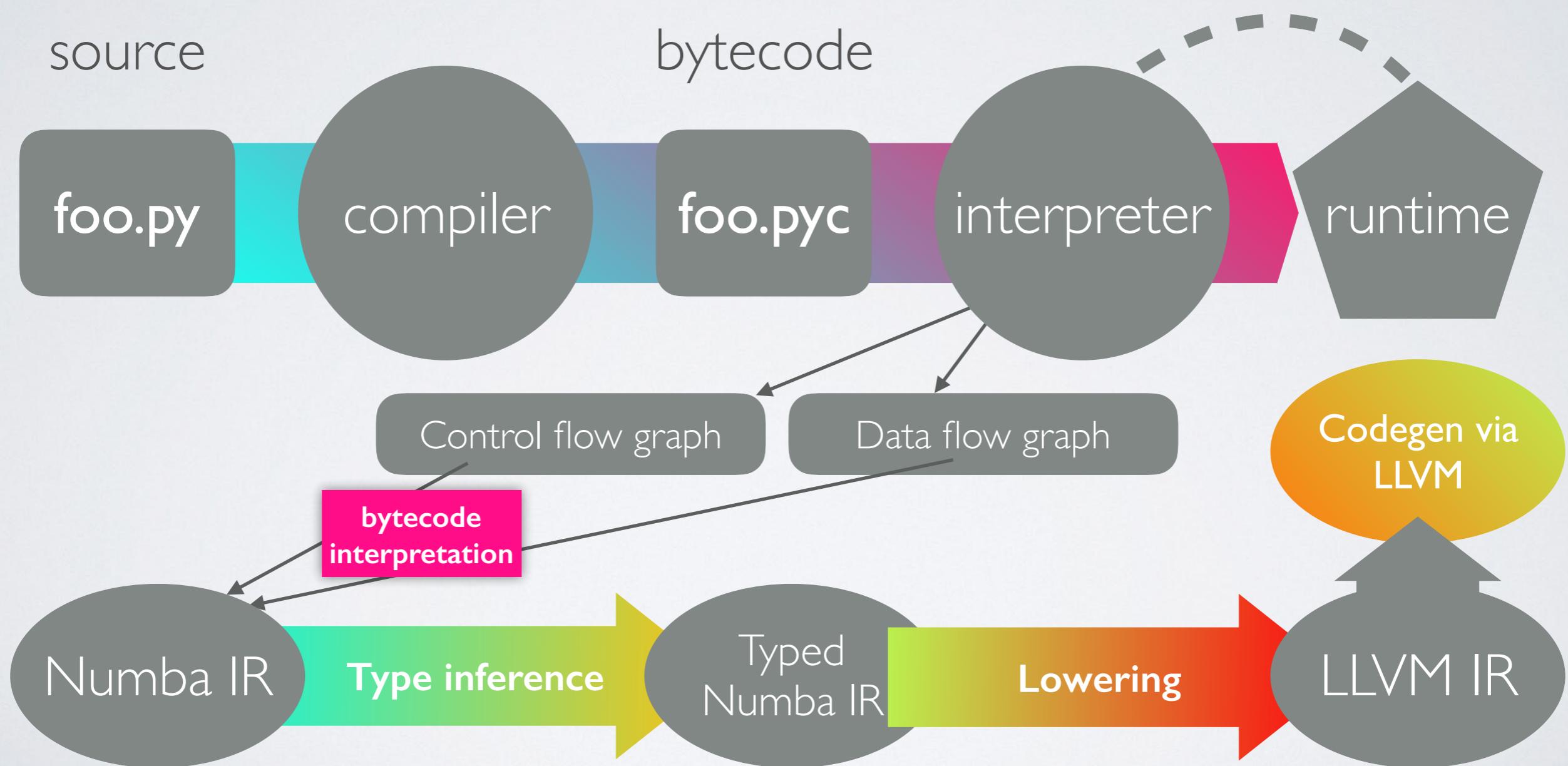
NUMBA
JIT (LLVM) compiler for Python

NUMBA

Numba is a compiler for Python array and numerical functions that gives you the power to speed up code written directly in Python.

- uses LLVM to boil down pure Python code to JIT optimised machine code
- only accelerates selected functions decorated by yourself
- native code generation for CPU (default) and GPU
- integration with the Python scientific software stack (thanks to NumPy)
- runs side by side with regular Python code or third-party C extensions and libraries
- great CUDA support
- N-core scalability by releasing the GIL (beware: no protection from race conditions!)
- create NumPy ufuncs with the `@[gu]vectorize` decorator(s)

FROM SOURCE TO RUNTIME



NUMBA JIT-EXAMPLE

```
numbers = np.arange(1000000).reshape(2500, 400)
```

```
def sum2d(arr):  
    M, N = arr.shape  
    result = 0.0  
    for i in range(M):  
        for j in range(N):  
            result += arr[i,j]  
    return result
```

```
@nb.jit  
def sum2d_jit(arr):  
    M, N = arr.shape  
    result = 0.0  
    for i in range(M):  
        for j in range(N):  
            result += arr[i,j]  
    return result
```

289 ms ± 3.02 ms per loop

2.13 ms ± 42.6 µs per loop

~135x faster, with a single line of code

NUMBA VECTORIZE-EXAMPLE

```
a = np.arange(1000000, dtype='f8')
b = np.arange(1000000, dtype='f8') + 23
```

NumPy:

```
np.abs(a - b) / (np.abs(a) + np.abs(b))           23 ms ± 845 µs per loop
```

Numba @vectorize:

```
@nb.vectorize
def nb_rel_diff(a, b):
    return abs(a - b) / (abs(a) + abs(b))
```

```
rel_diff(a, b)           3.56 ms ± 43.2 µs per loop
```

~6x faster

NUMEXPR

initially written by David Cooke

Routines for the fast evaluation of array expressions element-wise
by using a vector-based virtual machine.

NUMEXPR USAGE EXAMPLE

```
import numpy as np
import numexpr as ne

a = np.arange(5)
b = np.linspace(0, 2, 5)

ne.evaluate("a**2 + 3*b")

array([ 0. ,  2.5,  7. , 13.5, 22. ])
```

NUMEXPR SPEED-UP

```
a = np.random.random(1000000)
```

NumPy:

```
2 * a**3 - 4 * a**5 + 6 * np.log(a)
```

82.4 ms ± 1.88 ms per loop

Numexpr with 4 threads:

```
ne.set_num_threads(4)
```

```
ne.evaluate("2 * a**3 - 4 * a**5 + 6 * log(a)")
```

7.85 ms ± 103 µs per loop

~10x faster

NUMEXPR – SUPPORTED OPERATORS

- Logical operators: `&`, `|`, `~`
- Comparison operators:
`<`, `<=`, `==`, `!=`, `>=`, `>`
- Unary arithmetic operators: `-`
- Binary arithmetic operators:
`+`, `-`, `*`, `/`, `**`, `%`, `<<`, `>>`

NUMEXPR – SUPPORTED FUNCTIONS

- `where(bool, number1, number2)`: number -- number1 if the bool condition is true, number2 otherwise.
- `{sin,cos,tan}(float|complex)`: float|complex -- trigonometric sine, cosine or tangent.
- `{arcsin,arccos,arctan}(float|complex)`: float|complex -- trigonometric inverse sine, cosine or tangent.
- `arctan2(float1, float2)`: float -- trigonometric inverse tangent of float1/float2.
- `{sinh,cosh,tanh}(float|complex)`: float|complex -- hyperbolic sine, cosine or tangent.
- `{arsinh,arccosh,arctanh}(float|complex)`: float|complex -- hyperbolic inverse sine, cosine or tangent.
- `{log,log10,log1p}(float|complex)`: float|complex -- natural, base-10 and log(1+x) logarithms.
- `{exp,expm1}(float|complex)`: float|complex -- exponential and exponential minus one.
- `sqrt(float|complex)`: float|complex -- square root.
- `abs(float|complex)`: float|complex -- absolute value.
- `conj(complex)`: complex -- conjugate value.
- `{real,imag}(complex)`: float -- real or imaginary part of complex.
- `complex(float, float)`: complex -- complex from real and imaginary parts.
- `contains(str, str)`: bool -- returns True for every string in `op1` that contains `op2`.
- `sum(number, axis=None)`: Sum of array elements over a given axis. Negative axis are not supported.
- `prod(number, axis=None)`: Product of array elements over a given axis. Negative axis are not supported.

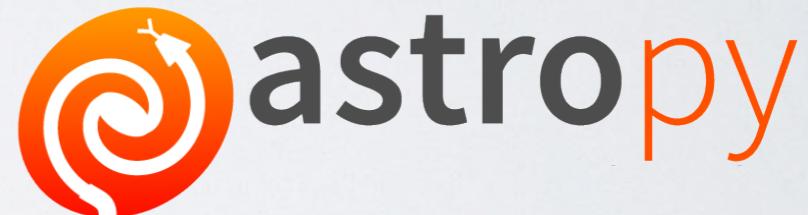


THE HISTORY OF ASTROPY

(standard situation back in 2011)

- Example Problem: convert from EQ J2000 RA/Dec to Galactic coordinates
- Solution in Python
 - pyast
 - Astrolib
 - Astrophysics
 - PyEphem
 - PyAstro
 - Kapteyn
 - ???

huge discussion
started in June 2011
series of votes



First public version (v0.2) presented and described in the following paper:
<http://adsabs.harvard.edu/abs/2013A&A...558A..33A>

ASTROPY CORE PACKAGE

A community-driven package intended to contain much of the core functionality and some common tools needed for performing astronomy and astrophysics with Python.

- **Data structures and transformations**

- constants, units and quantities, N-dimensional datasets, data tables, times and dates, astronomical coordinate system, models and fitting, analytic functions

- **Files and I/O**

- unified read/write interface
- FITS, ASCII tables, VOTable (XML), Virtual Observatory access, HDF5, YAML, ...

- **Astronomy computations and utilities**

- cosmological calculations, convolution and filtering, data visualisations, astrostatistics tools

ASTROPY AFFILIATED PACKAGES

- Tons of astronomy related packages
- which are not part of the core package,
- but has requested to be included as part of the Astropy project's community

ASTROPY EXAMPLE

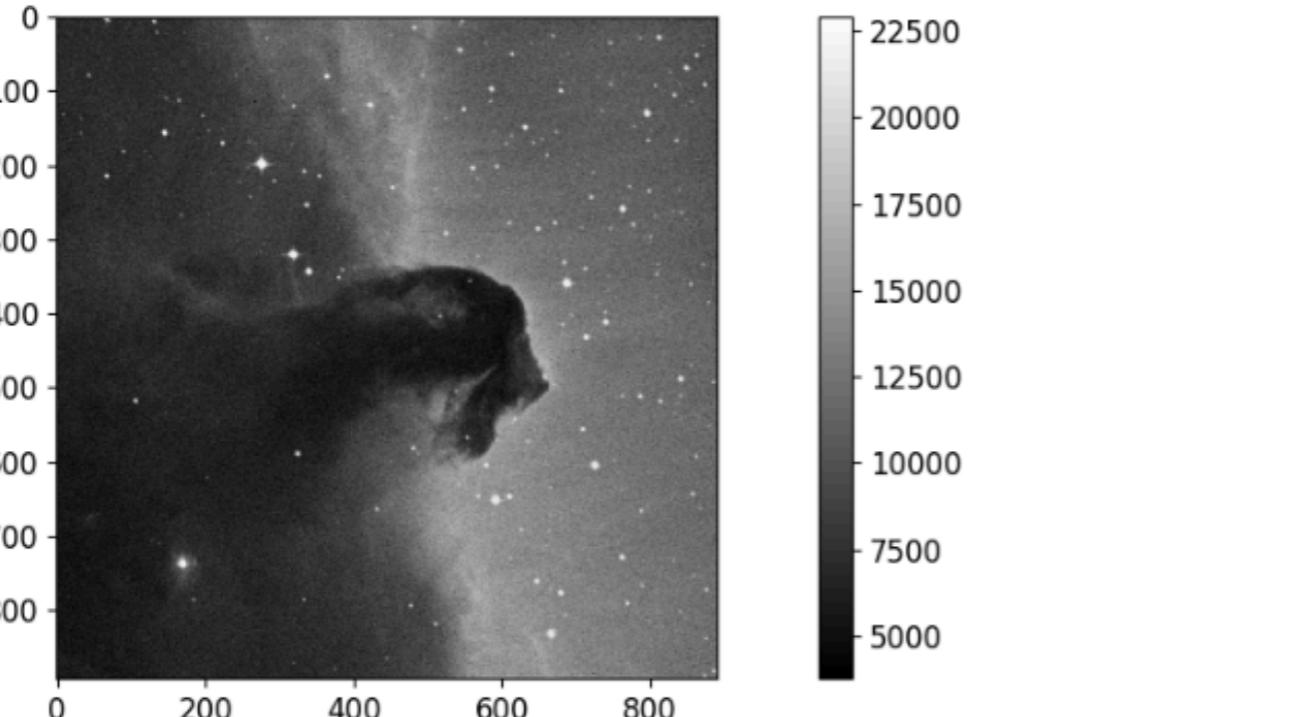
```
from astropy.utils.data import download_file
from astropy.io import fits

image_file = download_file('http://data.astropy.org/tutorials/FITS-images/HorseHead.fits')
Downloading http://data.astropy.org/tutorials/FITS-images/HorseHead.fits [Done]

fits.info(image_file)
Filename: /Users/tamasgal/.astropy/cache/download/py3/2c9202ae878ecfcb60878ceb63837f5f
No.    Name        Type      Cards   Dimensions   Format
 0  PRIMARY    PrimaryHDU     161   (891, 893)   int16
 1  er.mask    TableHDU      25   1600R x 4C   [F6.2, F6.2, F6.2, F6.2]

image_data = fits.getdata(image_file, ext=0)

plt.figure()
plt.imshow(image_data, cmap='gray')
plt.colorbar();
```



- ← downloading via HTTP
- ← checking some FITS meta
- ← extracting image data
- ← plotting via Matplotlib

ASTROPY EXAMPLE

```
from astropy.coordinates import SkyCoord
import astropy.units as u

m13 = SkyCoord.from_name('m13')
m13

<SkyCoord (ICRS): (ra, dec) in deg
  ( 250.4234583, 36.4613056)>

m13.ra, m13.ra.to(u.hourangle)

(<Longitude 250.4234583 deg>, <Longitude 16.69489722 hourangle>)
```

Don't worry, we will discover AstroPy in the hands-on workshop!



A Python library for symbolic mathematics.

SIMPY

- It aims to become a full-featured computer algebra system (CAS)
- while keeping the code as simple as possible
- in order to be comprehensible and easily extensible.
- SymPy is written entirely in Python.
- It only depends on mpmath, a pure Python library for arbitrary floating point arithmetic

SIMPY

- solving equations
- solving differential equations
- simplifications: trigonometry, polynomials
- substitutions
- factorisation, partial fraction decomposition
- limits, differentiation, integration, Taylor series
- combinatorics, statistics, ...
- much much more

SIMPY EXAMPLE

Base Python

```
In [1]: import math  
  
In [2]: math.sqrt(8)  
Out[2]: 2.8284271247461903  
  
In [3]: math.sqrt(8)**2  
Out[3]: 8.000000000000002
```

SymPy

```
In [4]: import sympy  
  
In [5]: sympy.sqrt(8)  
Out[5]: 2*sqrt(2)  
  
In [6]: sympy.sqrt(8)**2  
Out[6]: 8
```

SIMPY EXAMPLE

```
In [15]: x, y = sympy.symbols('x y')
```

```
In [16]: expr = x + 2*y
```

```
In [17]: expr
```

```
Out[17]: x + 2*y
```

```
In [18]: expr + 1
```

```
Out[18]: x + 2*y + 1
```

```
In [19]: expr * x
```

```
Out[19]: x*(x + 2*y)
```

```
In [20]: sympy.expand(expr * x)
```

```
Out[20]: x**2 + 2*x*y
```

SIMPY EXAMPLE

```
In [1]: import sympy
```

```
In [2]: from sympy import init_printing, integrate, diff, exp, cos, sin, oo
```

```
In [3]: init_printing(use_unicode=True)
```

```
In [4]: x = sympy.symbols('x')
```

```
In [5]: diff(sin(x)*exp(x), x)
```

```
Out[5]:
```

$$e^x \cdot \sin(x) + e^x \cdot \cos(x)$$

```
In [6]: integrate(exp(x)*sin(x) + exp(x)*cos(x), x)
```

```
Out[6]:
```

$$\frac{e^x \cdot \sin(x)}{2}$$

```
In [7]: integrate(sin(x**2), (x, -oo, oo))
```

```
Out[7]:
```

$$\frac{\sqrt{2} \cdot \sqrt{\pi}}{2}$$



I P [y]:

IPython

IPYTHON

- The interactive Python shell!
- Object introspection
- Input history, persistent across sessions
- Extensible tab completion
- “Magic” commands (basically macros)
- Easily embeddable in other Python programs and GUIs
- Integrated access to the pdb debugger and the Python profiler
- Syntax highlighting
- real multi-line editing
- Provides a kernel for Jupyter
- ...and such more!



Project Jupyter is an open source project that offers a set of tools for interactive and exploratory computing.

JUPYTER

- Born out of the IPython project in 2014
- Jupyter provides a console and a notebook server for all kinds of languages
(the name Jupyter comes from **Julia**, **Python** and **R**)
- An easy way to explore and prototype
- Notebooks support Markdown and LaTeX-like input and rendering
 - Allows sharing code and analysis results
 - Extensible (slideshow plugins, JupyterLab, VIM binding, ...)

JUPYTER CONSOLE

A terminal frontend for kernels which use the Jupyter protocol.

The screenshot displays three terminal windows illustrating the Jupyter console:

- Top Terminal:** Shows the command `jupyter kernelspec list` being run, displaying available kernels: haskell, julia-0.5, julia-0.6, km3net, and python3.
- Middle Left Terminal:** Shows the command `jupyter console` being run, resulting in the Python 3.6 Jupyter console starting up.
- Middle Right Terminal:** Shows the command `jupyter console --kernel=julia-0.5` being run, resulting in the Julia Jupyter console starting up.
- Bottom Terminal:** Shows the Python 3.6 Jupyter console interface, including the IPython banner and a code cell input field labeled "In [1]:".

JUPYTER NOTEBOOK

- A Web-based application suitable for capturing the whole computation process:
 - developing
 - documenting
 - and executing code
 - as well as communicating the results.
- Two main components:
 - a web application: a browser-based tool for interactive authoring of documents which combine explanatory text, mathematics, computations and their rich media output.
 - notebook documents: a representation of all content visible in the web application, including inputs and outputs of the computations, explanatory text, mathematics, images, and rich media representations of objects.

JUPYTER NOTEBOOK

The screenshot shows a Jupyter Notebook interface running on a Mac OS X system. The top navigation bar includes standard OS X icons for window control and a tab for 'localhost:8888/notebooks/Research/DU-2'. The main window has two side-by-side panes.

Left Pane (Code Input): This pane contains two code cells. The first cell's input is:

```
In [51]: df[df["hit_time"] < 40]["tot"].hist(bins=255)
```

Its output is:

```
Out[51]: <matplotlib.axes._subplots.AxesSubplot at 0x11d73d6a0>
```

Below this is another code cell:

```
In [72]: df[df['time_length'] > 100]
```

Its output is:

channel_id	dom_id	time	tot	triggered	event_id	hit_time	time_length
0	23	808953148	241952	64	0	0	0
1	25	808953148	241953	30	0	0	1
2	27	808953148	241957	34	0	0	5
3	30	808953148	241978	25	0	0	26
4	0	808953148	241955	37	0	0	3
5	0	808953148	242041	37	0	0	89
6	1	808953148	242041	46	0	0	89

Right Pane (Rendered Output): This pane shows a 3D visualization of event data and a snippet of Python code. The code is:(useOffset=False, axis='y')
formatter(xfmt)
(bbox_to_anchor=(1.005, 1), loc=2, borderaxespad=0.)
.lines():
'

Below the code is a 3D plot showing event data over time. A legend on the right lists floors from 0 to 18. The x-axis shows dates from December 2015 to February 2016. The y-axis ranges from 0 to 1400. The z-axis represents different floors. The plot shows several colored layers of data points, with a prominent yellow layer at the bottom.

Annotations:

- cells for code/markup input** (pink text) points to the code cells in the left pane.
- rendered output** (pink text) points to the 3D plot in the right pane.
- for text/images/tables etc.** (pink text) points to the table output in the left pane.

JUPYTERLAB

- The next level of interacting with notebooks
- Extensible: terminal, text editor, image viewer, etc.
- Supports editing multiple notebooks at once
- Drag and drop support to arrange panes

JUPYTERLAB

Screenshot of JupyterLab interface showing multiple notebooks and a file browser.

File Browser:

- Path: Research > Playground
- Files:
 - Julia (a month ago)
 - scipy_2015_sklearn_t... (6 months ago)
 - System Monitoring (a year ago)
 - 1.2_Tools_numpy_p... (a year ago)
 - 3D Line Fit.ipynb (a year ago)
 - An introduction to Ma... (6 months ago)
 - Aussie Rules Football.... (a year ago)
 - Bad Colour Maps.ipynb (a year ago)
 - Coin Flip - Waiting for ... (a year ago)
 - Configparser.ipynb (a year ago)
 - Cython.ipynb (a year ago)
 - Distances of points in ... (a year ago)
 - Distributions.ipynb (a year ago)
 - Draw Picture Pixel by ... (a year ago)
 - DU Plot.ipynb (10 months ago)
 - Fun with the Pipeline.i... (a year ago)
 - HDF5 Basics.ipynb (a month ago)
 - HDF5 Formats.ipynb (a year ago)
 - HDF5 Performance.ip... (8 months ago)
 - Hit vs CHit Performan... (a year ago)
 - HitSeries.ipynb (a year ago)
 - Interact.ipynb (a year ago)
 - Känguruh.ipynb (a year ago)
 - Leap Seconds.ipynb (a year ago)
 - Linear Equations Syst... (a year ago)
 - Machine Learning.ipynb (6 months ago)
 - Matplotlib Subplots.ip... (a year ago)
 - Mensch Ärgere Dich ... (4 months ago)
 - Neural Networks.ipynb (7 months ago)
 - Numba.ipynb (4 months ago)
 - Numexpr.ipynb (8 months ago)
 - Numpy - Named Tipl... (a year ago)
 - Pandas Appendix.ipynb (a year ago)

Notebooks:

- DU2-DOM9 Lo X** (No Kernel):
 - In [21]:

```
fig, ax = plt.subplots()
du2dom9 = db.doms.via_omkey((2, 9), "D_ARCA003")
du2dom3 = db.doms.via_omkey((2, 3), "D_ARCA003")
temp[temp.SOURCE_NAME == du2dom9.clb_upi].plot('DATETIME',
'VALUE', ax=ax, label=du2dom9)
temp[temp.SOURCE_NAME == du2dom3.clb_upi].plot('DATETIME',
'VALUE', ax=ax, label=du2dom3)
plt.xlabel("Time on 2016-11-04 [UTC]")
plt.ylabel("Temperature [°C]")
```

 - Out[21]:
- K40.ipynb** (KM3NeT):
 - In [11]:

```
times, channel_ids = [np.array(i) for i in
zip(*foo)]
print(len(times))
#print(channel_ids)

diffs = np.diff(times)
#print(diffs)
idx = np.where(np.diff(times) < 20)[0]
#print(idx)
break
narf(times)
#print(channel_ids[idx])
```

 - Out[11]:

channel_id	dom_id	id	pmt_id	time	tot	triggered	event_id	
0	28	808430449	0	0	20292053	28	False	0
1	12	808430571	1	0	20290049	26	False	0
2	8	808447091	2	0	20288472	27	False	0

 - In [104]:

```
tmax = 20
def mongincidence(times, tdcs):
    coincidences = []
    cur_t = 0
    las_t = 0
    for t_idx, t in enumerate(times):
        cur_t = t
        diff = cur_t - las_t
        if diff < tmax and t_idx > 0:
            coincidences.append(((tdcs[t_idx - 1],
tdcs[t_idx]), diff))
            las_t = cur_t
    return coincidences
```

 - In [105]:

```
mongincidence((1, 20, 21), (10, 11, 12))
```

 - Out[105]:

```
[((10, 11), 19), ((11, 12), 1)]
```
- IPython: Users X**:
 - In [5]:

```
...: del shorterr
...:
```

 - In [6]:

```
import numpy as np
```

 - In [6]:

```
np.add[...]
```

add_newdoc_ufunc()
add_docstring()
add_newdocs
add_newdoc()

JUPYTERHUB

- JupyterHub creates a multi-user Hub which spawns, manages, and proxies multiple instances of the single-user Jupyter notebook server
- A nice environment for teaching
- Great tool for collaborations
(ask your IT admin ;)

SOME OTHER USEFUL LIBRARIES

SEABORN

statistical data visualisation
uses matplotlib as backend

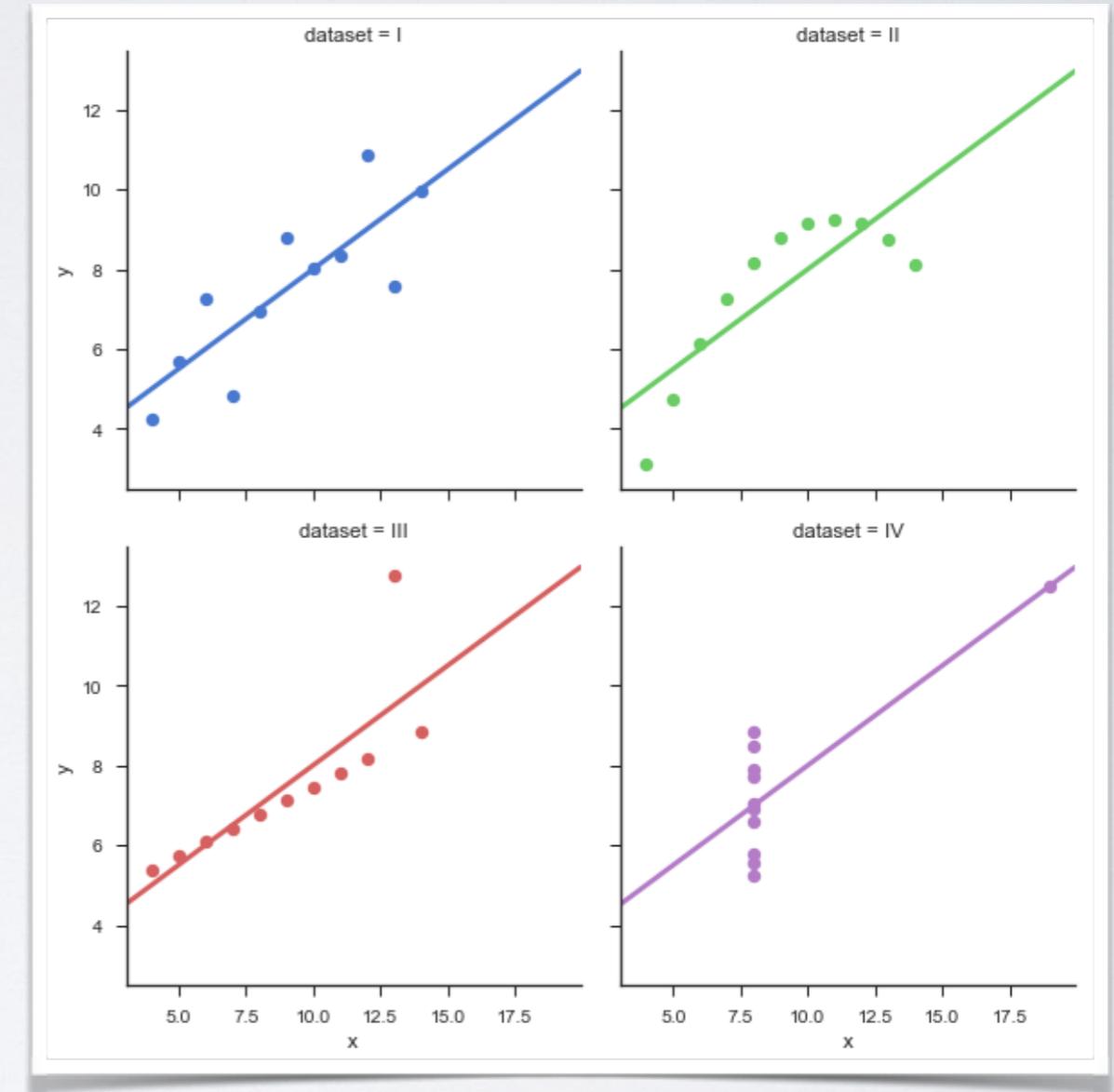
<https://seaborn.pydata.org>

CONVENIENT WRAPPER FUNCTIONS FOR MATPLOTLIB

```
import seaborn as sns
sns.set(style="ticks")

df = sns.load_dataset("anscombe")

# Show the results of a linear regression
# within each dataset
sns.lmplot(x="x", y="y", col="dataset",
            hue="dataset", data=df,
            col_wrap=2, ci=None,
            palette="muted", size=4,
            scatter_kws={"s": 50, "alpha": 1})
```

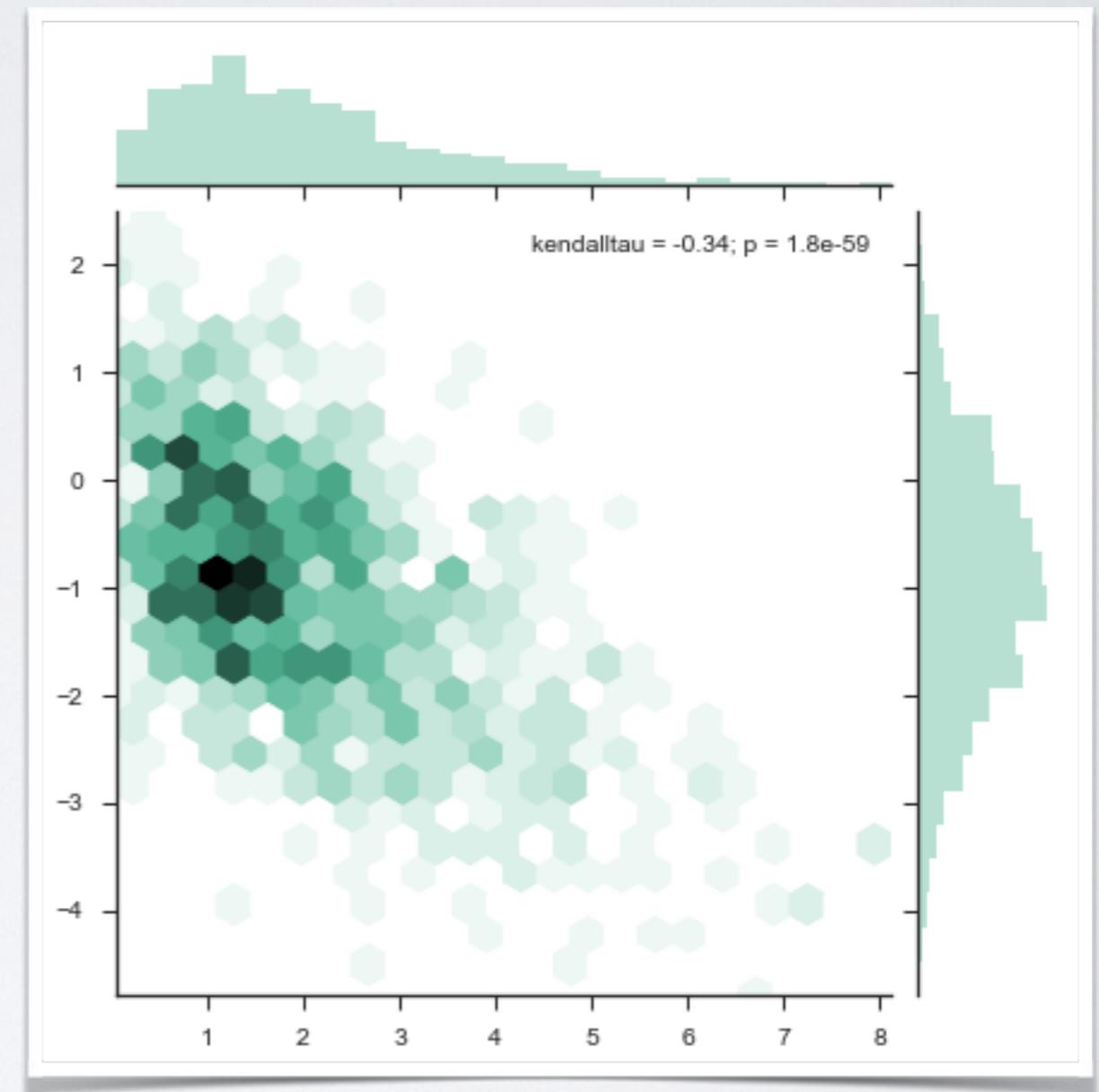


CONVENIENT WRAPPER FUNCTIONS FOR MATPLOTLIB

```
import numpy as np
from scipy.stats import kendalltau
import seaborn as sns
sns.set(style="ticks")

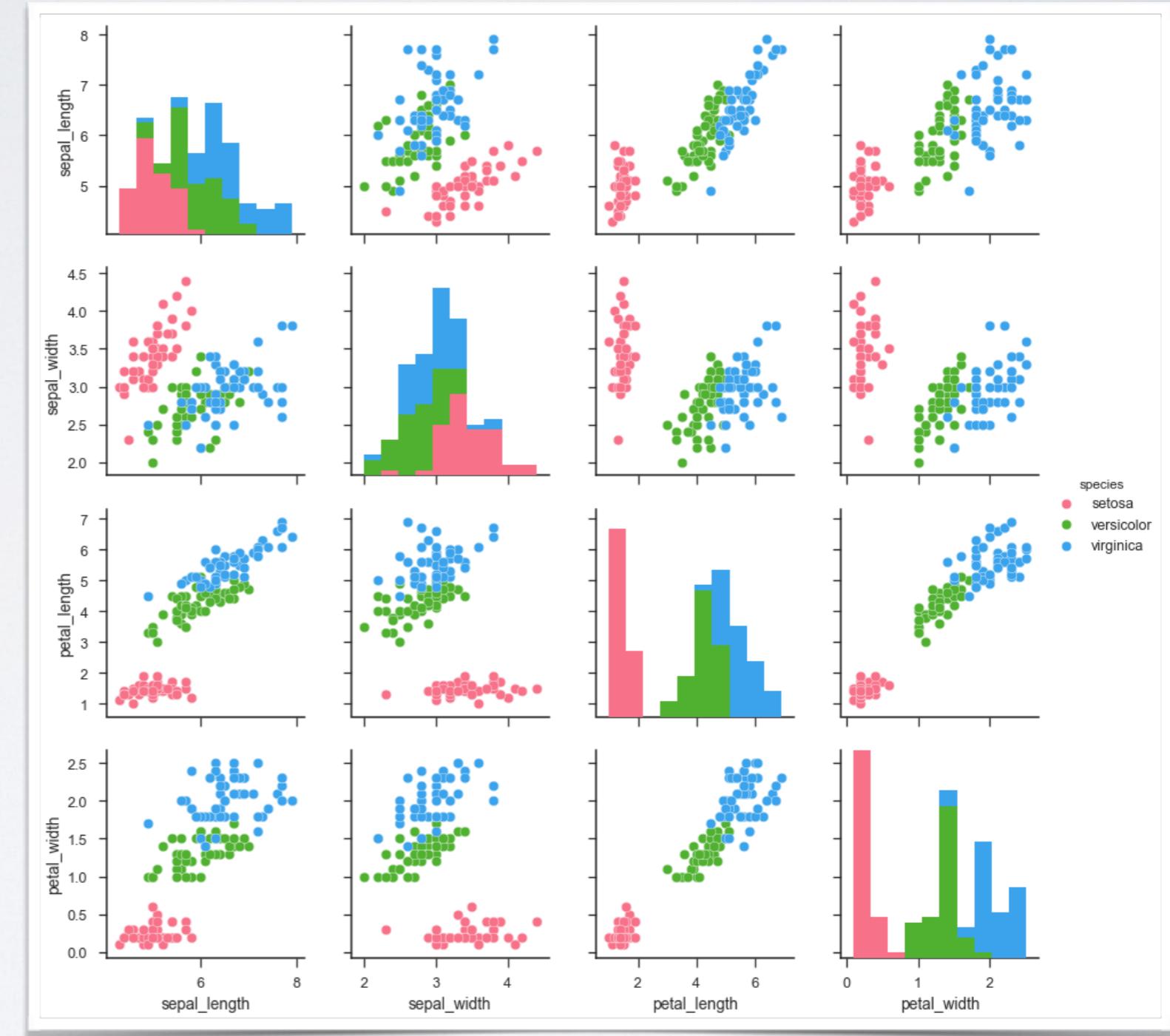
rs = np.random.RandomState(11)
x = rs.gamma(2, size=1000)
y = -.5 * x + rs.normal(size=1000)

sns.jointplot(x, y, kind="hex",
                stat_func=kendalltau,
                color="#4CB391")
```



CONVENIENT WRAPPER FUNCTIONS FOR MATPLOTLIB

```
import seaborn as sns  
sns.set(style="ticks",  
       color_codes=True)  
  
iris = sns.load_dataset("iris")  
sns.pairplot(iris,  
                      hue="species",  
                      palette="husl")
```



You will learn more about
seaborn from **David Kirkby!**

DOCOPT

creates beautiful command-line interfaces

by Vladimir Keleshev

<https://github.com/docopt/docopt>

ARGPARSE/OPTPARSE

Many classes and functions,
default values,
extensive documentation,
very hard to memorise
a basic setup.

The screenshot shows a Python Software Foundation IDE window with a scrollable content area. The content displays the argparse module's documentation. At the top, there are two tabs: one showing a command-line interface with usage information and another showing code examples. Below these are several sections of text, each preceded by a '16.4.' identifier. The sections include:

- 16.4.4.3. Arguments containing -
- 16.4.4.4. Argument abbreviations (prefix matching)

Each section contains explanatory text and code examples. The code examples show how to use the `ArgumentParser` class to handle command-line arguments, including positional arguments, optional arguments, and subparsers. The explanatory text provides details on how argparse handles various argument types and conflicts.

DOCOPT

```
#!/usr/bin/env python
```

```
"""
```

Naval Fate.

Usage:

```
naval_fate ship new <name> ...
naval_fate ship <name> move <x> <y> [--speed=<kn>]
naval_fate ship shoot <x> <y>
naval_fate mine (set|remove) <x> <y> [--moored|--drifting]
naval_fate -h | --help
naval_fate --version
```

Options:

-h --help	Show this screen.
--version	Show version.
--speed=<kn>	Speed in knots [default: 10].
--moored	Moored (anchored) mine.
--drifting	Drifting mine.

```
"""
```

```
from docopt import docopt
```

```
arguments = docopt(__doc__, version='Naval Fate 2.0')
```

DOCOPT

```
naval_fate ship Guardian move 10 50 --speed=20
```



```
arguments =  
{  
    "--drifting": false,  
    "--help": false,  
    "--moored": false,  
    "--speed": "20",  
    "--version": false,  
    "<name>": [  
        "Guardian"  
    ],  
    "<x>": "10",  
    "<y>": "50",  
    "mine": false,  
    "move": true,  
    "new": false,  
    "remove": false,  
    "set": false,  
    "ship": true,  
    "shoot": false  
}
```

CLICK

a mature command line utility interface package

<http://click.pocoo.org>

CLICK

- Much more advanced compared to docopt
- The no.1 choice if you want to go crazy with command line utilities

```
import click

@click.command()
@click.option('--count', default=1, help='Number of greetings.')
@click.option('--name', prompt='Your name',
              help='The person to greet.')
def hello(count, name):
    """Simple program that greets NAME for a total of COUNT times."""
    for x in range(count):
        click.echo('Hello %s!' % name)

if __name__ == '__main__':
    hello()
```

SO, WHAT NOW?

FINAL PERSONAL THOUGHTS

I spent a lot of time optimising Python code in the past years, here is a short summary of my personal experience.

- There were several attempts to make Python itself faster w.r.t. low level programming, none of them are satisfying (PyPy may have a future, but still doesn't fully support Python 3), many of them were abandoned
- Think twice (or more) before you bake Cython or any other static compilation into your project. The two language problem is real and it's hard to get it right. The performance gain is often disillusioning compared to the work, workarounds and "mess" one needs to deal with.
- Me and my lovely dev-team made the best experiences with numba
 - no clutter or double bookkeeping, no (static) compilation
 - minimal dependencies (basically only LLVMlite)
 - often orders of magnitudes faster than comparable low level algorithms utilising custom Cython class instances or ctypes
 - downside: super slow without numba ...
- When it comes to high performance code using Python, you have to think in numpy arrays and cannot model your own datatypes like e.g. in C or C++ (structs, classes ...)

MY RECEIPT FOR PERFORMANT PYTHON CODE

- **Avoid massive amounts of Python class instances**

(e.g. don't create a class for a Point and then a list of 10 million points!)

- **Use numpy arrays for large homogenous data**

(w.r.t. the "points" example above, create a 3xN numpy recarray instead, so you can access points.x, points.y and point.z. Subclass the array if you need some special functionality)

- **Vectorisation is a good idea (most of the time).**

For basic operations, you most likely find a dedicated function in numpy or scipy.

- **Try to reuse already allocated memory** (allocations are expensive!)

- **Always profile first, before you do heavy optimisations!**

"[...] premature optimization is the root of all evil." –D. Knuth

Keep in mind, this doesn't mean that you sit down and hack together code, whatever works, this is not what Donald meant! Take care of the basic principles of performant code from the very beginning, otherwise you will have a hard time to refactor.

- **Do not reinvent the wheel.**

You mostly find a lib which does what you need, better, faster and for no cost.

Ohne more thing ...

AN EXAMPLE WHY IT'S SO HARD TO MAKE PYTHON FAST?

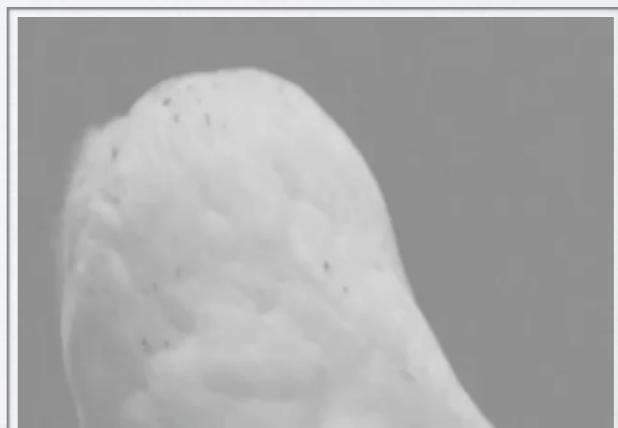
JUST A SIMPLE, BUT CRUCIAL ASPECT ...

- Python lets you do anything.
- Here is a "pure" function, written in Python:

```
def square(x):  
    return float(x)**2
```

- Every decent compiler should now be able to optimise code using this function (repeated calls, tail recursion elimination, inlining, thread safety guarantees, etc.)

```
import builtins  
builtins.float = int
```



THANK YOU!

...also many thanks to Vincent and Jayesh,
and the whole organising committee!

Acknowledgement

- H2020-Astronomy ESFRI and Research Infrastructure Cluster (Grant Agreement number: 653477).

Useful Python Libraries

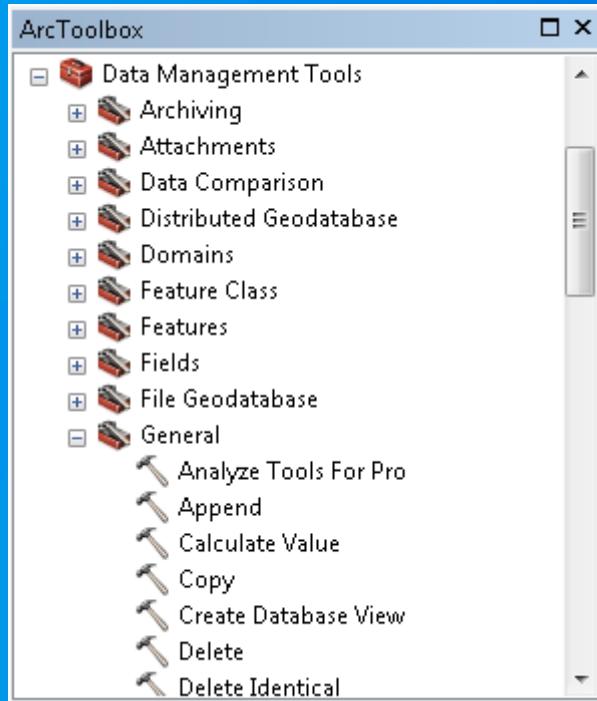
Ghislain Prince

This presentation

- Standard libraries
 - “batteries included”
- 3rd party libs

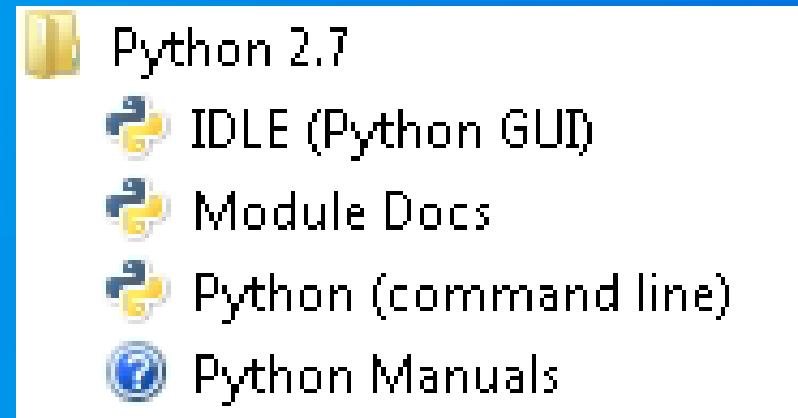
Regarding python 3

- docs.python.org/2/library/2to3.html
- “Analyze tools for Pro” tool
- Future : pick you python (Conda)



Standard Library is Large, powerful and Well-Documented

- docs.python.org
- Installed with ArcGIS



3rd Party Library

- Not part of standard library
- Pypi – the Python Package Index
 - 60,000+ packages
 - <https://pypi.python.org/pypi>
 - pip
- C:\>c:\Python27\ArcGIS10.3\Scripts\pip.exe install requests
- Downloading requests-2.7.0-py2.py3-none-any.whl (470kB)
- 100% |#####| 471kB 718kB/s
- Installing collected packages: requests
- Successfully installed requests-2.7.0

Data Formats

- XML
- JSON
- CSV
- Excel
- PDF
- Numpy/Pandas*
- NetCDF

3 XML options within the standard library

- SAX
 - Can handle huge datasets
 - Hard to use, verbose code
- DOM
 - Familiar: Javascript uses the DOM
 - Easy (ish)
- Etree
 - Feels like python
 - Extremely easy and concise for common tasks

JSON

- json module

```
>>> import json  
>>> d = json.load(open("states.json", 'r'))  
>>> print(type(d))  
<class 'dict'>
```

CSV & Excel

- **csv module**

```
>>> import csv  
>>> with open('eggs.csv', 'rb') as csvfile:  
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')  
...     for row in spamreader:  
...         print ', '.join(row)  
Spam, Spam, Spam, Spam, Spam, Baked Beans  
Spam, Lovely Spam, Wonderful Spam
```

- **xlrd & xlwt (third-party, included in ArcGIS)**

PDF

- **arcpy.mapping.PDFDocument**
- **reportlab (third-party)**
 - *allows rapid creation of rich PDF documents, and also creation of charts in a variety of bitmap and vector formats.*

Networking

- Calls to HTTP servers
 - `urllib2`
 - `requests` (third-party, pip-install)
 - `asyncio`

Computing

- Numpy
- Pandas *
- Scipy *
- Sympy *
- R bridge (coming soon: github.com/R-ArcGIS ... wait this isn't python at all)

- * Pro 1.0 & planned for Desktop/Server 10.4

Resources

arcpy.wordpress.com

twitter.com/arcpy

desktop.arcgis.com/en/desktop/



Understanding our world.