**Importing a module**

import math

import sys

import math, sys

import math

print(math.sin(math.pi/2))

import math

def sin(x):

if 2 \* x == pi:

return 0.99999999

else:

return None

pi = 3.14

print(sin(pi/2))

print(math.sin(math.pi/2))

from math import pi

from math import pi

print(math.e)

NameError: name 'math' is not defined

from math import sin, pi

print(sin(pi/2))

from math import sin, pi

print(sin(pi / 2))

pi = 3.14

def sin(x):

if 2 \* x == pi:

return 0.99999999

else:

return None

print(sin(pi / 2))

from module import \*

Treat this as a temporary solution, and try not to use it in regular code.

import module as alias

from module import n as a, m as b, o as c

from math import pi as PI, sin as sine

print(sine(PI/2))

# Key takeaways

import mod1

import mod2, mod3, mod4 is not recommended due to stylistic reasons

import my\_module

result = my\_module.my\_function(my\_module.my\_data)

from module import my\_function, my\_data

result = my\_function(my\_data)

is not recommended because of the danger of causing conflicts with names derived from importing the code's namespace.

from my\_module import \*

result = my\_function(my\_data)

this import's variant is not recommended due to the same reasons as previously

# Selected functions from the math module

* sin(x) → the sine of x;
* cos(x) → the cosine of x;
* tan(x) → the tangent of x.
* asin(x) → the arcsine of x;
* acos(x) → the arccosine of x;
* atan(x) → the arctangent of x.
* pi → a constant with a value that is an approximation of π;
* radians(x) → a function that converts x from degrees to radians;
* degrees(x) → acting in the other direction (from radians to degrees)
* sinh(x) → the hyperbolic sine;
* cosh(x) → the hyperbolic cosine;
* tanh(x) → the hyperbolic tangent;
* asinh(x) → the hyperbolic arcsine;
* acosh(x) → the hyperbolic arccosine;
* atanh(x) → the hyperbolic arctangent.
* e → a constant with a value that is an approximation of Euler's number (e)
* exp(x) → finding the value of ex;
* log(x) → the natural logarithm of x
* log(x, b) → the logarithm of x to base b
* log10(x) → the decimal logarithm of x (more precise than log(x, 10))
* log2(x) → the binary logarithm of x (more precise than log(x, 2))
* pow(x, y) → finding the value of xy (mind the domains). This is a built-in function, and doesn't have to be imported.
* ceil(x) → the ceiling of x (the smallest integer greater than or equal to x)
* floor(x) → the floor of x (the largest integer less than or equal to x)
* trunc(x) → the value of x truncated to an integer (be careful - it's not an equivalent either of ceil or floor)
* factorial(x) → returns x! (x has to be an integral and not a negative)
* hypot(x, y) → returns the length of the hypotenuse of a right-angle triangle with the leg lengths equal to x and y (the same as sqrt(pow(x, 2) + pow(y, 2)) but more precise)

# random module

The most general function named random() (not to be confused with the module's name) **produces a float number**x**coming from the range**(0.0, 1.0) - in other words: (0.0 <= x < 1.0).

from random import random

for i in range(5):

print(random())

* seed() - sets the seed with the current time;
* seed(int\_value) - sets the seed with the integer value int\_value.

**Selected functions from the random module: continued**

**The randrange and randint functions**

If you want integer random values, one of the following functions would fit better:

* randrange(end) => [0, end)
* randrange(beg, end) => [beg, end)
* randrange(beg, end, step) => [beg, end) with **step**
* randint(left, right) => [beg, end]

**The choice and sample functions**

As you can see, this is not a good tool for generating numbers in a lottery. Fortunately, there is a better solution than writing your own code to check the uniqueness of the "drawn" numbers.

It's a function named in a very suggestive way - choice:

* choice(sequence)
* sample(sequence, elements\_to\_choose)

The first variant chooses a "random" element from the input sequence and returns it.

The second one builds a list (a sample) consisting of the elements\_to\_choose element "drawn" from the input sequence.

In other words, the function chooses some of the input elements, returning a list with the choice. The elements in the sample are placed in random order. Note: the elements\_to\_choose must not be greater than the length of the input sequence.

from random import choice, sample

my\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(choice(my\_list))

print(sample(my\_list, 5))

print(sample(my\_list, 10))

3

[7, 1, 2, 3, 6]

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

**Selected functions from the platform module**

**The platform function**

The platform module lets you access the underlying platform's data, i.e., hardware, operating system, and interpreter version information.

There is a function that can show you all the underlying layers in one glance, named platform, too. It just returns a string describing the environment; thus, its output is rather addressed to humans than to automated processing (you'll see it soon).

This is how you can invoke it:

platform(aliased = False, terse = False)

And now:

* aliased → when set to True (or any non-zero value) it may cause the function to present the alternative underlying layer names instead of the common ones;
* terse → when set to True (or any non-zero value) it may convince the function to present a briefer form of the result (if possible)

We ran our sample program using three different platforms - this is what we got:

**Intel x86 + Windows ® Vista (32 bit)**:

Windows-Vista-6.0.6002-SP2

Windows-Vista-6.0.6002-SP2

Windows-Vista

**The machine function**

Sometimes, you may just want to know the generic name of the processor which runs your OS together with Python and your code - a function named machine() will tell you that. As previously, the function returns a string.

**Intel x86 + Windows ® Vista (32 bit)**:

x86

**The processor function**

The processor() function returns a string filled with the real processor name (if possible).

Once again, we ran the sample program on three different platforms:

**Intel x86 + Windows ® Vista (32 bit)**:

x86

**The system function**

A function named system() returns the generic OS name as a string.

Our example platforms presented themselves like this:

**Intel x86 + Windows ® Vista (32 bit)**:

Windows

**The version function**

The OS version is provided as a string by the version() function.

Run the code and check its output. This is what we got:

**Intel x86 + Windows ® Vista (32 bit)**:

6.0.6002

**The python\_implementation and the python\_version\_tuple functions**

If you need to know what version of Python is running your code, you can check it using a number of dedicated functions - here are two of them:

* python\_implementation() → returns a string denoting the Python implementation (expect CPython here, unless you decide to use any non-canonical Python branch)

* python\_version\_tuple() → returns a three-element tuple filled with:
  + the **major** part of Python's version;
  + the **minor** part;
  + the **patch** level number.
* CPython
* 3 7 7

# What is a package?

#!/usr/bin/env python3

""" module.py - an example of a Python module """

\_\_counter = 0

def suml(the\_list):

global \_\_counter

\_\_counter += 1

the\_sum = 0

for element in the\_list:

the\_sum += element

return the\_sum

def prodl(the\_list):

global \_\_counter

\_\_counter += 1

prod = 1

for element in the\_list:

prod \*= element

return prod

if \_\_name\_\_ == "\_\_main\_\_":

print("I prefer to be a module, but I can do some tests for you.")

my\_list = [i+1 for i in range(5)]

print(suml(my\_list) == 15)

print(prodl(my\_list) == 120)

* the line starting with #! has many names - it may be called *shabang*, *shebang*, *hashbang*, *poundbang* or even *hashpling* (don't ask us why). The name itself means nothing here - its role is more important. From Python's point of view, it's just a **comment** as it starts with #. For **Unix and Unix-like OSs (including MacOS)** such a line **instructs the OS how to execute the contents of the file** (in other words, what program needs to be launched to interpret the text). In some environments (especially those connected with web servers) the absence of that line will cause trouble;
* a string (maybe a multiline) placed before any module instructions (including imports) is called the **doc-string**, and should briefly explain the purpose and contents of the module;
* the functions defined inside the module (suml() and prodl()) are available for import;
* we've used the \_\_name\_\_ variable to detect when the file is run stand-alone, and seized this opportunity to perform some simple tests.

Python browses these folders in the order in which they are listed in the list - if the module cannot be found in any of these directories, the import fails.

Otherwise, the first folder containing a module with the desired name will be taken into consideration (if any of the remaining folders contains a module of that name, it will be ignored).

The variable is named path, and it's accessible through the module named sys. This is how you can check its regular value:

import sys

for p in sys.path:

print(p)

The names *shabang*, *shebang*, *hasbang*, *poundbang*, and *hashpling* describe the digraph written as #!, used to instruct Unix-like OSs how the Python source file should be launched. This convention has no effect under MS Windows.

If you want convince Python that it should take into account a non-standard package's directory, its name needs to be inserted/appended into/to the import directory list stored in the path variable contained in the sys module.

import sys

# note the double backslashes!

sys.path.append("D:\\Python\\Project\\Modules")

**Python Package Installer (PIP)**

The repository (or *repo* for short) we mentioned before is named **PyPI** (it's short for Python Package Index) and it's maintained by a workgroup named the Packaging Working Group, a part of the Python Software Foundation, whose main task is to support Python developers in efficient code dissemination.

# The PyPI repo: the Cheese Shop

*pip* means *“****pip installs packages***

According to Bicking himself, the name is a [recursive acronym](https://en.wikipedia.org/wiki/Recursive_acronym) for "**Pip Installs Packages**".

# *pip* on MS Windows

pip –version

# *pip* on Linux

pip --version

pip3 --version

sudo apt install python3-pip

# How to use *pip*

pip help

pip help *operation*

pip help install

pip list

pip show *package\_name*

pip show pip

# Dependencies

**dependency is a phenomenon that appears every time you're going to use a piece of software that relies on other software**.

the process of arduously fulfilling all the subsequent requirements has its own name, and it's called ***dependency hell*.**

# How to use *pip*

pip help

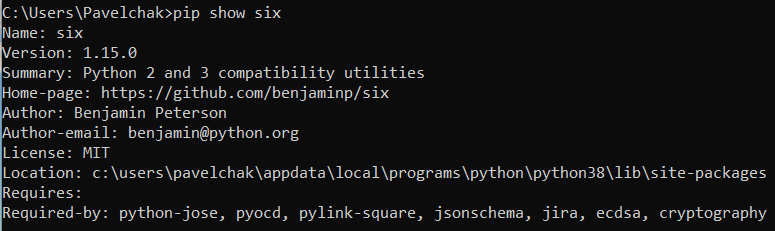
pip help ***operation***

pip help install

pip list

pip show ***package\_name***

pip show pip



Look at the two lines at the bottom of the output. They show:

* which packages are needed to successfully utilize the package (Requires:)
* which packages need the package to be successfully utilized (Required-by:)

pip search anystring

pip search pip

It is deprecated!!!

pip install pygame

If you're not an admin:

pip install --user pygame

to **update** a locally installed package

pip install -U package\_name

to **install a user-selected version**

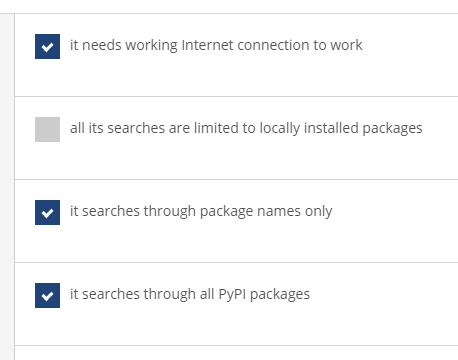
pip install pygame==1.9.2

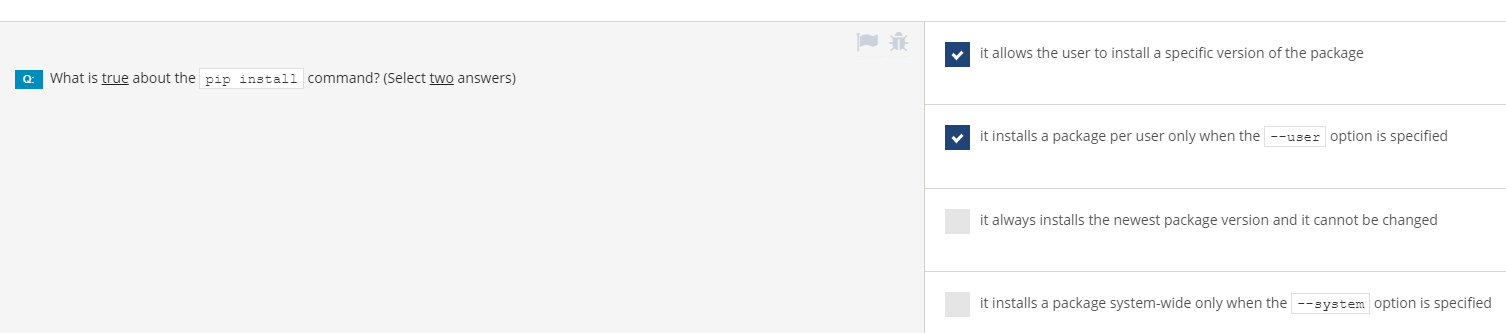
pip uninstall pygame

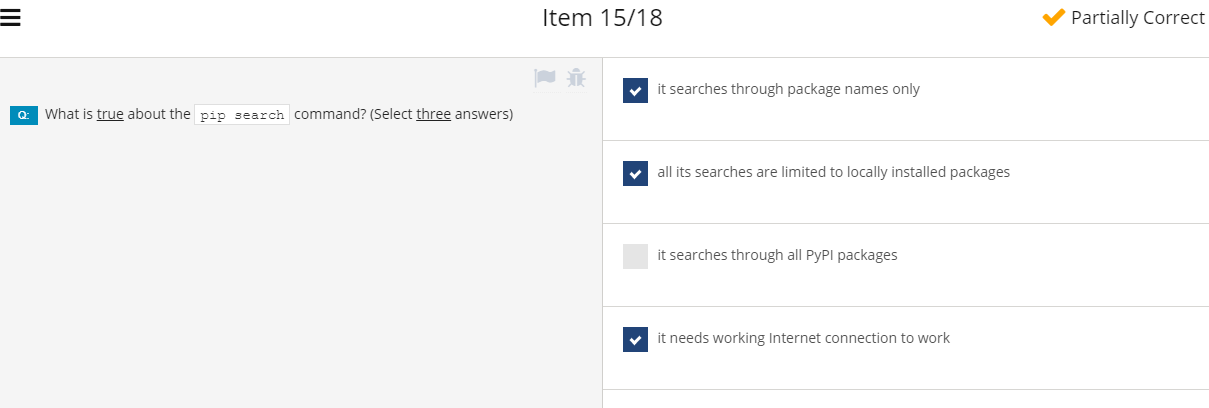
List of main **pip** activities looks as follows:

* pip --version
* pip3 --version
* pip help *operation* - shows brief pip's description;
* pip list - shows list of currently installed packages;
* pip show *package\_name* - shows *package\_name* info including package's dependencies;
* pip search *anystring* - searches through PyPI directories in order to find packages which name contains *anystring*;
* pip install *name* - installs *name* system-wide (expect problems when you don't have administrative rights);
* pip install --user *name* - install *name* for you only; no other your platform's user will be able to use it;
* pip install -U *name* - updates previously installed package;
* pip uninstall *name* - uninstalls previously installed package;









# Python Essentials 2: Module 2 **Strings, String and List Methods, Exceptions**

**ASCII** (short for **American Standard Code for Information Interchange**)

## Code points and code pages

A code point is **a number which makes a character**. For example, *32* is a code point which makes a *space* in ASCII encoding. We can say that standard ASCII code consists of 128 code points.

A code page is a **standard for using the upper 128 code points to store specific national characters**.

## UCS-4

**Universal Character Set**. **UCS-4 uses 32 bits (four bytes) to store each character**

 A file containing UCS-4 encoded text may start with a BOM (byte order mark), an unprintable combination of bits announcing the nature of the file's contents.

## UTF-8

**Unicode Transformation Format**

A number corresponding to a particular character is called a **codepoint**.

If you want **to know a specific character's ASCII/UNICODE code point value**, you can use a function named ord() (as in *ordinal*).

char\_1 = 'a'

char\_2 = ' ' # space

print(ord(char\_1))

print(ord(char\_2))

If you know the code point (number) and want to get the corresponding character, you can use a function named chr().

print(chr(97))

print(chr(945))

* chr(ord(x)) == x
* ord(chr(x)) == x

# Operations on strings: min()

The function **finds the minimum element of the sequence passed as an argument**. There is one condition - the sequence (string, list, it doesn't matter) **cannot be empty**, or else you'll get a ValueError exception.

**Operations on strings: max()**

a function named max() **finds the maximum element of the sequence**.

**Operations on strings: the index() method**

The index() method (it's a method, not a function) **searches the sequence from the beginning, in order to find the first element of the value specified in its argument**.

Note: the element searched for must occur in the sequence - **its absence will cause a ValueError exception**.

print("aAbByYzZaA".index("b"))  
print("aAbByYzZaA".index("Z"))  
print("aAbByYzZaA".index("A"))

2

7

1

**The find() method**

The find() method is similar to index(), which you already know - **it looks for a substring and returns the index of first occurrence of this substring**, but:

* it's safer - it **doesn't generate an error for an argument containing a non-existent substring** (it returns -1 then)
* it **works with strings only** - don't try to apply it to any other sequence.

print("Eeetata".find("ta"))  
print("Eta".find("mma"))

3

-1

print('kappa'.find('a', 2))

4

the\_text = """A variation of the ordinary lorem ipsum

text has been used in typesetting since the 1960s

or earlier, when it was popularized by advertisements

for Letraset transfer sheets. It was introduced to

the Information Age in the mid-1980s by the Aldus Corporation,

which employed it in graphics and word-processing templates

for its desktop publishing program PageMaker (from Wikipedia)"""

fnd = the\_text.find('the')

while fnd != -1:

print(fnd)

fnd = the\_text.find('the', fnd + 1)

15

80

198

221

238

print('kappa'.find('a', 1, 4))

print('kappa'.find('a', 2, 4))

1

-1

**Operations on strings: the list() function**

The list() function **takes its argument (a string) and creates a new list containing all the string's characters, one per list element**.

print(list("abcabc"))

['a', 'b', 'c', 'a', 'b', 'c']

## **Operations on strings: the count() method**

he count() method **counts all occurrences of the element inside the sequence**. The absence of such elements doesn't cause any problems.

print("abcabc".count("bc"))

print('abcabc'.count("d"))

2

0

**The capitalize() method**

print('aBcD DDD ddd. sdfdsf dsfs'.capitalize())  
Abcd ddd ddd. sdfdsf dsfs

# The center() method

The one-parameter variant of the center() method makes a copy of the original string, trying to center it inside a field of a specified width.

print('[' + 'alpha'.center(10) + ']')

Output: [ alpha ]

# The endswith() method

t = "zeta"

print(t.endswith("a"))

print(t.endswith("A"))

print(t.endswith("et"))

print(t.endswith("eta"))

**The isalnum() method**

The parameterless method named isalnum() **checks if the string contains only digits or alphabetical characters (letters), and returns**True**or**False according to the result.

|  |  |  |
| --- | --- | --- |
| t = 'Six lambdas' print(t.isalnum()) False | t = 'ΑβΓδ' print(t.isalnum()) True | t = '20E1' print(t.isalnum()) True |

**The isalpha() method**

The isalpha() method is more specialized - it's interested in **letters only**.

## The isdigit() method

In turn, the isdigit() method looks at **digits only** - anything else produces False as the result.

**The islower() method**

The islower() method is a fussy variant of isalpha() - it accepts **lower-case letters only**.

## The isspace() method

The isspace() method **identifies whitespaces only** - it disregards any other character (the result is False then).

## The isupper() method

The isupper() method is the upper-case version of islower() - it concentrates on **upper-case letters only**.

**The join() method**

**The lower() method**

The lower() method **makes a copy of a source string, replaces all upper-case letters with their lower-case counterparts**, and returns the string as the result. Again, the source string remains untouched.

**The lstrip() method**

The parameterless lstrip() method **returns a newly created string formed from the original one by removing all leading whitespaces**.

print("[" + " tau ".lstrip() + "]")  
[tau ]

The **one-parameter** lstrip() method does the same as its parameterless version, but **removes all characters enlisted in its argument** (a string), not just whitespaces:  
print("www.cisco.com".lstrip("w."))  
cisco.com

**The replace() method**

The **two-parameter** replace() method **returns a copy of the original string in which all occurrences of the first argument have been replaced by the second argument**.

The **three-parameter** replace() variant uses the third argument (a number) to **limit the number of replacements**.

print("This is it!".replace("is", "are", 1))

print("This is it!".replace("is", "are", 2))

**The rfind() method**

The one-, two-, and three-parameter methods named rfind() do nearly the same things as their counterparts (the ones devoid of the *r* prefix), but **start their searches from the end of the string**, not the beginning (hence the prefix *r*, for *right*).

print("tau tau tau".rfind("ta"))  
print("tau tau tau".rfind("ta", 9))  
print("tau tau tau".rfind("ta", 3, 9))  
8  
-1  
4

**The rstrip() method**

Two variants of the rstrip() method do nearly the same as lstrips, but **affect the opposite side of the string**.

print("[" + " upsilon ".rstrip() + "]")  
print("cisco.com".rstrip(".com"))  
[ upsilon]  
cis

**The split() method**

The split() method does what it says - it **splits the string and builds a list of all detected substrings**.

The method **assumes that the substrings are delimited by whitespaces** - the spaces don't take part in the operation, and aren't copied into the resulting list.

print("phi chi\npsi".split())

['phi', 'chi', 'psi']

**The startswith() method**

The startswith() method is a mirror reflection of endswith() - it **checks if a given string starts with the specified substring**.

print("omega".startswith("meg"))  
print("omega".startswith("om"))

False

True

## The strip() method

The strip() method combines the effects caused by rstrip() and lstrip() - it **makes a new string lacking all the leading and trailing whitespaces**.

print("[" + " aleph ".strip() + "]")

[aleph]

# The swapcase() method

The swapcase() method **makes a new string by swapping the case of all letters within the source string**: lower-case characters become upper-case, and vice versa.

print("I know that I know nothing.".swapcase())

i KNOW THAT i KNOW NOTHING.

## The title() method

The title() method performs a somewhat similar function - it **changes every word's first letter to upper-case, turning all other ones to lower-case**.

print("I know that I know nothing. Part 1.".title())

I Know That I Know Nothing. Part 1.

## The upper() method

Last but not least, the upper() method **makes a copy of the source string, replaces all lower-case letters with their upper-case counterparts**, and returns the string as the result.

print("I know that I know nothing. Part 2.".upper())

I KNOW THAT I KNOW NOTHING. PART 2.

# Key takeaways

1. Some of the methods offered by strings are:

* capitalize() – changes all string letters to capitals;
* center() – centers the string inside the field of a known length;
* count() – counts the occurrences of a given character;
* join() – joins all items of a tuple/list into one string;
* lower() – converts all the string's letters into lower-case letters;
* lstrip() – removes the white characters from the beginning of the string;
* replace() – replaces a given substring with another;
* rfind() – finds a substring starting from the end of the string;
* rstrip() – removes the trailing white spaces from the end of the string;
* split() – splits the string into a substring using a given delimiter;
* strip() – removes the leading and trailing white spaces;
* swapcase() – swaps the letters' cases (lower to upper and vice versa)
* title() – makes the first letter in each word upper-case;
* upper() – converts all the string's letter into upper-case letters.

2. String content can be determined using the following methods (all of them return Boolean values):

* endswith() – does the string end with a given substring?
* isalnum() – does the string consist only of letters and digits?
* isalpha() – does the string consist only of letters?
* islower() – does the string consists only of lower-case letters?
* isspace() – does the string consists only of white spaces?
* isupper() – does the string consists only of upper-case letters?
* startswith() – does the string begin with a given substring?

**Comparing strings**

* ==
* !=
* >
* >=
* <
* <=

print('alpha' == 'alpha')  
print('alpha' != 'Alpha')  
print('alpha' < 'alphabet')  
print('beta' > 'Beta')

True

True

True

True

**Comparing strings against numbers is generally a bad idea.**

The only comparisons you can perform with impunity are these symbolized by the == and != operators. The former always gives False, while the latter always produces True.

Using any of the remaining comparison operators will raise a TypeError exception.

**Sorting**

The first is implemented as **a function named**sorted().

The function takes one argument (a list) and **returns a new list**, filled with the sorted argument's elements.

The second method affects the list itself - **no new list is created**. Ordering is performed in situ by the method named sort().

first\_greek = ['omega', 'alpha', 'pi', 'gamma']  
first\_greek\_2 = sorted(first\_greek)

second\_greek = ['omega', 'alpha', 'pi', 'gamma']  
second\_greek.sort()

**Strings vs. numbers**

**how to convert a number (an integer or a float) into a string, and vice versa**.

str() int() float()

# Key takeaways

1. Strings can be compared to strings using general comparison operators, but comparing them to numbers gives no reasonable result, because **no string can be equal** to any number. For example:

* string == number is always False;
* string != number is always True;
* string >= number always **raises an exception** TypeError .

2. Sorting lists of strings can be done by:

* a function named sorted(), creating a new, sorted list;
* a method named sort(), which sorts the list *in situ*

3. A number can be converted to a string using the str() function.

4. A string can be converted to a number (although not every string) using either the int() or float() function. The conversion fails if a string doesn't contain a valid number image (an exception is raised then ValueError ).

# Errors, failures, and other plagues

# Exceptions

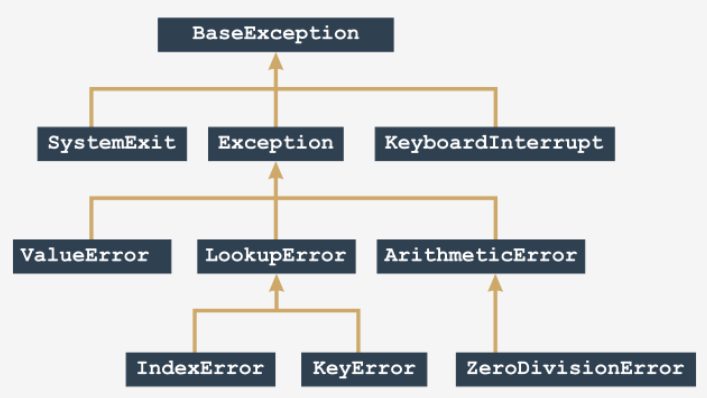
# ZeroDivisionError

# value = 1 value /= 0

# IndexError

my\_list = []  
x = my\_list[0]

# Exceptions



If you want to **handle two or more exceptions** in the same way, you can use the following syntax:

try:

:

except (exc1, exc2):

:

raise

There is one serious restriction: this kind of raise instruction may be used **inside the**except**branch** only; using it in any other context causes an error.

def bad\_fun(n):

try:

return n / 0

except:

print("I did it again!")

**raise**

try:

bad\_fun(0)

except ArithmeticError: # <--- **ZeroDivisionError**

print("I see!")

print("THE END.")

assert expression

**assertions don't supersede exceptions or validate the data**

import math

x = float(input("Enter a number: "))

assert x >= 0.0

x = math.sqrt(x)

print(x)

Enter a number: -1

Traceback (most recent call last):

File "main.py", line 4, in <module>

assert x >= 0.0

AssertionError

**Key takeaways**

1. You cannot add more than one anonymous (unnamed) except branch after the named ones.

:

# The code that always runs smoothly.

:

try:

:

# Risky code.

:

except Except\_1:

# Crisis management takes place here.

except Except\_2:

# We save the world here.

except:

# All other issues fall here.

:

# Back to normal.

:

2. All the predefined Python exceptions form a hierarchy, i.e. some of them are more general (the one named BaseException is the most general one) while others are more or less concrete (e.g. IndexError is more concrete than LookupError).

You shouldn't put more concrete exceptions before the more general ones inside the same except branche sequence. For example, you can do this:

try:

# Risky code.

except IndexError:

# Taking care of mistreated lists

except LookupError:

# Dealing with other erroneous lookups

but don't do that (unless you're absolutely sure that you want some part of your code to be useless)

try:

# Risky code.

except LookupError:

# Dealing with erroneous lookups

except IndexError:

# You'll never get here

3. The Python statement raise ExceptionName can raise an exception on demand. The same statement, but lacking *ExceptionName*, can be used inside the try branch **only**, and raises the same exception which is currently being handled.

4. The Python statement assert expression evaluates the *expression* and raises the AssertError exception when the *expression* is equal to zero, an empty string, or None. You can use it to protect some critical parts of your code from devastating data.

**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**

+-- **Warning**

+-- **DeprecationWarning**

+-- **PendingDeprecationWarning**

+-- **RuntimeWarning**

+-- **SyntaxWarning**

+-- **UserWarning**

+-- **FutureWarning**

+-- **ImportWarning**

+-- **UnicodeWarning**

+-- **BytesWarning**

+-- EncodingWarning

+-- **ResourceWarning**

## **ArithmeticError**

**Location:** BaseException ← Exception ← ArithmeticError

**Description:** an abstract exception including all exceptions caused by arithmetic operations like zero division or an argument's invalid domain

## **AssertionError**

**Location:** BaseException ← Exception ← AssertionError

**Description:** a concrete exception raised by the assert instruction when its argument evaluates to False, None, 0, or an empty string

## **BaseException**

**Location:** BaseException

**Description:** the most general (abstract) of all Python exceptions - all other exceptions are included in this one; it can be said that the following two except branches are equivalent: except: and except BaseException:.

## **IndexError**

**Location:** BaseException ← Exception ← LookupError ← IndexError

**Description:** a concrete exception raised when you try to access a non-existent sequence's element (e.g., a list's element)

## **KeyboardInterrupt**

**Location:** BaseException ← KeyboardInterrupt

**Description:** a concrete exception raised when the user uses a keyboard shortcut designed to terminate a program's execution (*Ctrl-C* in most OSs); if handling this exception doesn't lead to program termination, the program continues its execution.

Note: this exception is not derived from the Exception class. Run the program in IDLE.

## **LookupError**

**Location:** BaseException ← Exception ← LookupError

**Description:** an abstract exception including all exceptions caused by errors resulting from invalid references to different collections (lists, dictionaries, tuples, etc.)

## **MemoryError**

**Location:** BaseException ← Exception ← MemoryError

**Description:** a concrete exception raised when an operation cannot be completed due to a lack of free memory.

## **OverflowError**

**Location:** BaseException ← Exception ← ArithmeticError ← OverflowError

**Description:** a concrete exception raised when an operation produces a number too big to be successfully stored

## **ImportError**

**Location:** BaseException ← Exception ← StandardError ← ImportError

**Description:** a concrete exception raised when an import operation fails

## **ImportError**

**Location:** BaseException ← Exception ← StandardError ← ImportError

**Description:** a concrete exception raised when an import operation fails

## **KeyError**

**Location:** BaseException ← Exception ← LookupError ← KeyError

**Description:** a concrete exception raised when you try to access a collection's non-existent element (e.g., a dictionary's element)

# Key takeaways

1. Some abstract built-in Python exceptions are:

* ArithmeticError,
* BaseException,
* LookupError.

2. Some concrete built-in Python exceptions are:

* AssertionError,
* ImportError,
* IndexError,
* KeyboardInterrupt,
* KeyError,
* MemoryError,
* OverflowError.

**Object-Oriented Programming**

# Key takeaways

1. An **instance variable** is a property whose existence depends on the creation of an object. Every object can have a different set of instance variables.

Moreover, they can be freely added to and removed from objects during their lifetime. All object instance variables are stored inside a dedicated dictionary named \_\_dict\_\_, contained in every object separately.

2. An instance variable can be private when its name starts with \_\_, but don't forget that such a property is still accessible from outside the class using a **mangled name** constructed as \_ClassName\_\_PrivatePropertyName.

3. A **class variable** is a property which exists in exactly one copy, and doesn't need any created object to be accessible. Such variables are not shown as \_\_dict\_\_ content.

All a class's class variables are stored inside a dedicated dictionary named \_\_dict\_\_, contained in every class separately.

4. A function named hasattr() can be used to determine if any object/class contains a specified property.

For example:

class Sample:

gamma = 0 # Class variable.

def \_\_init\_\_(self):

self.alpha = 1 # Instance variable.

self.\_\_delta = 3 # Private instance variable.

obj = Sample()

obj.beta = 2 # Another instance variable (existing only inside the "obj" instance.)

print(obj.\_\_dict\_\_)

The code outputs:

{'alpha': 1, '\_Sample\_\_delta': 3, 'beta': 2}

\_\_name\_\_ attribute is absent from the object - **it exists only inside classes**.

\_\_module\_\_  - it **stores the name of the module which contains the definition of the class**.

\_\_bases\_\_ is a tuple. The **tuple contains classes** (not class names) which are direct superclasses for the class. (**only classes have this attribute** - objects don't.) (**a class without explicit superclasses points to object** (a predefined Python class) as its direct ancestor.)

* **introspection**, which is the ability of a program to examine the type or properties of an object at runtime;
* **reflection**, which goes a step further, and is the ability of a program to manipulate the values, properties and/or functions of an object at runtime.

getattr(), setattr(), isinstance()

# Key takeaways

1. A method is a function embedded inside a class. The first (or only) parameter of each method is usually named self, which is designed to identify the object for which the method is invoked in order to access the object's properties or invoke its methods.

2. If a class contains a **constructor** (a method named \_\_init\_\_) it cannot return any value and cannot be invoked directly.

3. All classes (but not objects) contain a property named \_\_name\_\_, which stores the name of the class. Additionally, a property named \_\_module\_\_ stores the name of the module in which the class has been declared, while the property named \_\_bases\_\_ is a tuple containing a class's superclasses.

For example:

class Sample:

def \_\_init\_\_(self):

self.name = Sample.\_\_name\_\_

def myself(self):

print("My name is " + self.name + " living in a " + Sample.\_\_module\_\_)

obj = Sample()

obj.myself()

The code outputs:

My name is Sample living in a \_\_main\_\_

# Inheritance

issubclass(ClassOne, ClassTwo)

**each class is considered to be a subclass of itself**.

isinstance(objectName, ClassName)

**Being an instance of a class means that the object (the cake) has been prepared using a recipe contained in either the class or one of its superclasses**.

object\_one is object\_two

**The**is**operator checks whether two variables (**object\_one**and**object\_two**here) refer to the same object**.

# How Python finds properties and methods

Super.\_\_init\_\_(self, name)

super().\_\_init\_\_(name)

the super() function, which **accesses the superclass without needing to know its name**:

you can use this mechanism not only to **invoke the superclass constructor, but also to get access to any of the resources available inside the superclass**.

the situation in which **the subclass is able to modify its superclass behavior (just like in the example) is called polymorphism**.

The method, redefined in any of the superclasses, thus changing the behavior of the superclass, is called **virtual**.

# Method Resolution Order (MRO)

class Top:

…

class Middle(Top):

…

class Bottom(Middle):<-ok class Bottom(Middle, Top): <-ok class Bottom(Top, Middle): <-error

…

object = Bottom()

**Key takeaways**

1. A method named \_\_str\_\_() is responsible for **converting an object's contents into a (more or less) readable string**. You can redefine it if you want your object to be able to present itself in a more elegant form. For example:

class Mouse:

def \_\_init\_\_(self, name):

self.my\_name = name

def \_\_str\_\_(self):

return self.my\_name

the\_mouse = Mouse('mickey')

print(the\_mouse) # Prints "mickey".

2. A function named issubclass(Class\_1, Class\_2) is able to determine if Class\_1 is a **subclass** of Class\_2. For example:

class Mouse:

pass

class LabMouse(Mouse):

pass

print(issubclass(Mouse, LabMouse), issubclass(LabMouse, Mouse)) # Prints "False True"

3. A function named isinstance(Object, Class) checks if an object **comes from an indicated class**. For example:

class Mouse:

pass

class LabMouse(Mouse):

pass

mickey = Mouse()

print(isinstance(mickey, Mouse), isinstance(mickey, LabMouse)) # Prints "True False".

4. A operator called is checks if two variables refer to **the same object**. For example:

class Mouse:

pass

mickey = Mouse()

minnie = Mouse()

cloned\_mickey = mickey

print(mickey is minnie, mickey is cloned\_mickey) # Prints "False True".

5. A parameterless function named super() returns a **reference to the nearest superclass of the class**. For example:

class Mouse:

def \_\_str\_\_(self):

return "Mouse"

class LabMouse(Mouse):

def \_\_str\_\_(self):

return "Laboratory " + super().\_\_str\_\_()

doctor\_mouse = LabMouse();

print(doctor\_mouse) # Prints "Laboratory Mouse".

6. Methods as well as instance and class variables defined in a superclass are **automatically inherited** by their subclasses. For example:

class Mouse:

Population = 0

def \_\_init\_\_(self, name):

Mouse.Population += 1

self.name = name

def \_\_str\_\_(self):

return "Hi, my name is " + self.name

class LabMouse(Mouse):

pass

professor\_mouse = LabMouse("Professor Mouser")

print(professor\_mouse, Mouse.Population) # Prints "Hi, my name is Professor Mouser 1"

7. In order to find any object/class property, Python looks for it inside:

* the object itself;
* all classes involved in the object's inheritance line from bottom to top;
* if there is more than one class on a particular inheritance path, Python scans them from left to right;
* if both of the above fail, the AttributeError exception is raised.

8. If any of the subclasses defines a method/class variable/instance variable of the same name as existing in the superclass, the new name **overrides** any of the previous instances of the name. For example:

class Mouse:

def \_\_init\_\_(self, name):

self.name = name

def \_\_str\_\_(self):

return "My name is " + self.name

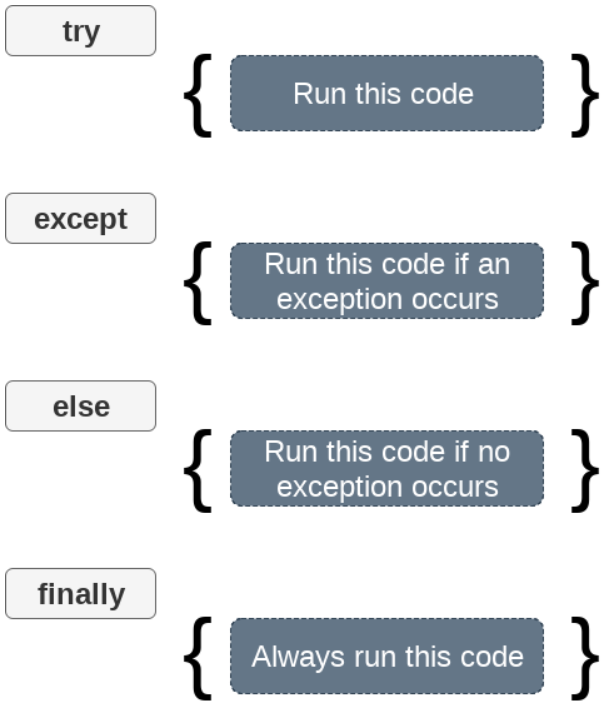
class AncientMouse(Mouse):

def \_\_str\_\_(self):

return "Meum nomen est " + self.name

mus = AncientMouse("Caesar") # Prints "Meum nomen est Caesar"

print(mus)



BaseException

+---Exception

| +---TypeError

| +---StopAsyncIteration

| +---StopIteration

| +---ImportError

| | +---ModuleNotFoundError

| | +---ZipImportError

| +---OSError

| | +---ConnectionError

| | | +---BrokenPipeError

| | | +---ConnectionAbortedError

| | | +---ConnectionRefusedError

| | | +---ConnectionResetError

| | +---BlockingIOError

| | +---ChildProcessError

| | +---FileExistsError

| | +---FileNotFoundError

| | +---IsADirectoryError

| | +---NotADirectoryError

| | +---InterruptedError

| | +---PermissionError

| | +---ProcessLookupError

| | +---TimeoutError

| | +---UnsupportedOperation

| | +---ItimerError

| +---EOFError

| +---RuntimeError

| | +---RecursionError

| | +---NotImplementedError

| | +---\_DeadlockError

| +---NameError

| | +---UnboundLocalError

| +---AttributeError

| +---SyntaxError

| | +---IndentationError

| | | +---TabError

| +---LookupError

| | +---IndexError

| | +---KeyError

| | +---CodecRegistryError

| +---ValueError

| | +---UnicodeError

| | | +---UnicodeEncodeError

| | | +---UnicodeDecodeError

| | | +---UnicodeTranslateError

| | +---UnsupportedOperation

| +---AssertionError

| +---ArithmeticError

| | +---FloatingPointError

| | +---OverflowError

| | +---ZeroDivisionError

| +---SystemError

| | +---CodecRegistryError

| +---ReferenceError

| +---MemoryError

| +---BufferError

| +---Warning

| | +---UserWarning

| | +---DeprecationWarning

| | +---PendingDeprecationWarning

| | +---SyntaxWarning

| | +---RuntimeWarning

| | +---FutureWarning

| | +---ImportWarning

| | +---UnicodeWarning

| | +---BytesWarning

| | +---ResourceWarning

| +---Error

+---GeneratorExit

+---SystemExit

+---KeyboardInterrupt

The BaseException class introduces a property named args. It's a **tuple designed to gather all arguments passed to the class constructor**. It is empty if the construct has been invoked without any arguments, or contains just one element when the constructor gets one argument (we don't count the self argument here), and so on.

# Key takeaways

1. The else: branch of the try statement is executed when there has been no exception during the execution of the try: block.

2. The finally: branch of the try statement is **always** executed.

3. The syntax except *Exception\_Name* as an *exception\_object*: lets you intercept an object carrying information about a pending exception. The object's property named args (a tuple) stores all arguments passed to the object's constructor.

4. The exception classes can be extended to enrich them with new capabilities, or to adopt their traits to newly defined exceptions.

For example:

try:

assert \_\_name\_\_ == "\_\_main\_\_"

except:

print("fail", end=' ')

else:

print("success", end=' ')

finally:

print("done")

The code outputs: success done.

# Python Essentials 2: Module 4

**Miscellaneous**

# Generators

The **iterator protocol is a way in which an object should behave to conform to the rules imposed by the context of the**for**and**in**statements**. An object conforming to the iterator protocol is called an **iterator**.

An iterator must provide two methods:

* \_\_iter\_\_() which should **return the object itself** and which is invoked once (it's needed for Python to successfully start the iteration)
* \_\_next\_\_() which is intended to **return the next value** (first, second, and so on) of the desired series - it will be invoked by the for/in statements in order to pass through the next iteration; if there are no more values to provide, the method should **raise the**StopIteration**exception**.

Generators may also be used within **list comprehensions**

The list() function can transform a series of subsequent generator invocations into **a real list**

Moreover, the context created by the in operator allows you to use a generator, too.

# The lambda function

A lambda function is a function without a name (you can also call it **an anonymous function**).

lambda parameters: expression

Such a clause **returns the value of the expression when taking into account the current value of the current**lambda**argument**.

two = lambda: 2

sqr = lambda x: x \* x

pwr = lambda x, y: x \*\* y

for a in range(-2, 3):

print(sqr(a), end=" ")

print(pwr(a, two()))

lambda x: 2 \* x\*\*2 - 4 \* x + 2

map(function, list)

* the second map() argument may be any entity that can be iterated (e.g., a tuple, or just a generator)
* map() can accept more than two arguments.

filter()

 it **filters its second argument while being guided by directions flowing from the function specified as the first argument** (the function is invoked for each list element, just like in map()).

he elements which return True from the function **pass the filter** - the others are rejected.

# closures

def func1():

a = 1

b = 'line'

c = [1, 2, 3]

def func2():

c.append(4)

# a += 1 <- generate: **UnboundLocalError:** local variable 'a' referenced before assignment

return a, b, c

return func2

a, b, c = call\_func = func1()()

print(a)

print(b)

print(c)

def func1():

a = 1

b = 'line'

c = [1, 2, 3]

def func2():

c.append(4)

**nonlocal** a

a += 1

return a, b, c

return func2

### **When and why to use Closures:**

Closures can avoid the use of global values and provides some form of data hiding. It can also provide an object oriented solution to the problem.

1. As closures are used as callback functions, they provide some sort of data hiding. This helps us to reduce the use of global variables.

2.  When we have few functions in our code, closures prove to be an efficient way. But if we need to have many functions, then go for class (OOP).

def func\_as\_object(a,b):

def add():

return a+b

def sub():

return a-b

def mul():

return a\*b

def replace():

pass

replace.add = add

replace.sub = sub

replace.mul = mul

return replace

obj1 = func\_as\_object(5,2)

print(obj1.add())

obj2 = func\_as\_object(3,6)

print(obj2.add())

# Key takeaways

1. An **iterator** is an object of a class providing at least **two** methods (not counting the constructor!):

* \_\_iter\_\_() is invoked once when the iterator is created and returns the iterator's object **itself**;
* \_\_next\_\_() is invoked to provide the **next iteration's value** and raises the StopIteration exception when the iteration **comes to and end**.

2. The yield statement can be used only inside functions. The yield statement suspends function execution and causes the function to return the yield's argument as a result. Such a function cannot be invoked in a regular way – its only purpose is to be used as a **generator** (i.e. in a context that requires a series of values, like a for loop.)

3. A **conditional expression** is an expression built using the if-else operator. For example:

print(True if 0 >=0 else False)

outputs True.

4. A **list comprehension** becomes a **generator** when used inside **parentheses** (used inside brackets, it produces a regular list). For example:

for x in (el \* 2 for el in range(5)):

print(x)

outputs 02468.

4. A **lambda function** is a tool for creating **anonymous functions**. For example:

def foo(x,f):

return f(x)

print(foo(9, lambda x: x \*\* 0.5))

outputs 3.0.

5. The map(fun, list) function creates a **copy** of a list argument, and applies the fun function to all of its elements, returning a **generator** that provides the new list content element by element. For example:

short\_list = ['mython', 'python', 'fell', 'on', 'the', 'floor']

new\_list = list(map(lambda s: s.title(), short\_list))

print(new\_list)

outputs ['Mython', 'Python', 'Fell', 'On', 'The', 'Floor'].

6. The filter(fun, list) function creates a **copy** of those list elements, which cause the fun function to return True. The function's result is a **generator** providing the new list content element by element. For example:

short\_list = [1, "Python", -1, "Monty"]

new\_list = list(filter(lambda s: isinstance(s, str), short\_list))

print(new\_list)

outputs ['Python', 'Monty'].

7. A closure is a technique which allows the **storing of values** in spite of the fact that the **context** in which they have been created **does not exist anymore**. For example:

def tag(tg):

tg2 = tg

tg2 = tg[0] + '/' + tg[1:]

def inner(str):

return tg + str + tg2

return inner

b\_tag = tag('<b>')

print(b\_tag('Monty Python'))

outputs <b>Monty Python</b>

[PEP 8](https://www.python.org/dev/peps/pep-0008/#programming-recommendations), the Style Guide for Python Code, recommends that **lambdas should not be assigned to variables, but rather they should be defined as functions**.

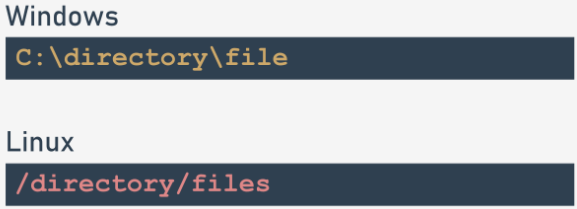
# Recommended:

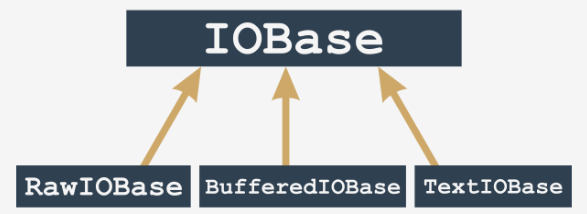
def f(x): return 3\*x

# Not recommended:

f = lambda x: 3\*x

**FILES**





stream = open(file, mode = 'r', encoding = None)

* the name of the function (open) speaks for itself; if the opening is successful, the function returns a stream object; otherwise, an exception is raised (e.g., FileNotFoundError **if the file you're going to read doesn't exist**);
* the first parameter of the function (file) specifies the name of the file to be associated with the stream;
* the second parameter (mode) specifies the open mode used for the stream; it's a string filled with a sequence of characters, and each of them has its own special meaning (more details soon);
* the third parameter (encoding) specifies the encoding type (e.g., UTF-8 when working with text files)
* the opening must be the very first operation performed on the stream.

**Opening the streams: modes**

r open mode: read

* the stream will be opened in **read mode**;
* the file associated with the stream **must exist** and has to be readable, otherwise the open() function raises an exception.

w open mode: write

* the stream will be opened in **write mode**;
* the file associated with the stream **doesn't need to exist**; if it doesn't exist it will be created; if it exists, it will be truncated to the length of zero (erased); if the creation isn't possible (e.g., due to system permissions) the open() function raises an exception.

a open mode: append

* the stream will be opened in **append mode**;
* the file associated with the stream **doesn't need to exist**; if it doesn't exist, it will be created; if it exists the virtual recording head will be set at the end of the file (the previous content of the file remains untouched.)

r+ open mode: read and update

* the stream will be opened in **read and update mode**;
* the file associated with the stream **must exist and has to be writeable**, otherwise the open() function raises an exception;
* both read and write operations are allowed for the stream.

w+ open mode: write and update

* the stream will be opened in **write and update** mode;
* the file associated with the stream **doesn't need to exist**; if it doesn't exist, it will be created; the previous content of the file remains untouched;
* both read and write operations are allowed for the stream.

**Selecting text and binary modes**

If there is a letter b at the end of the mode string it means that the stream is to be opened in the **binary mode**.

If the mode string ends with a letter t the stream is opened in the **text mode**.

Text mode is the default behaviour assumed when no binary/text mode specifier is used.

Finally, the successful opening of the file will set the current file position (the virtual reading/writing head) before the first byte of the file **if the mode is not**a and after the last byte of file **if the mode is set to**a.

| **Text mode** | **Binary mode** | **Description** |
| --- | --- | --- |
| rt | rb | read |
|  |  |  |
| wt | wb | write |
| at | ab | append |
| r+t | r+b | read and update |
| w+t | w+b | write and update |

EXTRA

You can also open a file for its exclusive creation. You can do this using the x open mode. If the file already exists, the open() function will raise an exception.

* sys.stdin
  + stdin (as *standard input*)
  + the stdin stream is normally associated with the keyboard, pre-open for reading and regarded as the primary data source for the running programs;
  + the well-known input() function reads data from stdin by default.

* sys.stdout
  + stdout (as *standard output*)
  + the stdout stream is normally associated with the screen, pre-open for writing, regarded as the primary target for outputting data by the running program;
  + the well-known print() function outputs the data to the stdout stream.

* sys.stderr
  + stderr (as *standard error output*)
  + the stderr stream is normally associated with the screen, pre-open for writing, regarded as the primary place where the running program should send information on the errors encountered during its work;
  + we haven't presented any method to send the data to this stream (we will do it soon, we promise)
  + the separation of stdout (useful results produced by the program) from the stderr (error messages, undeniably useful but does not provide results) gives the possibility of redirecting these two types of information to the different targets. More extensive discussion of this issue is beyond the scope of our course. The operation system handbook will provide more information on these issues.

**exc.errno:**

Let's take a look at some selected **constants useful for detecting stream errors**:

* errno.EACCES → Permission denied  
    
  The error occurs when you try, for example, to open a file with the *read only* attribute for writing.

* errno.EBADF → Bad file number  
    
  The error occurs when you try, for example, to operate with an unopened stream.

* errno.EEXIST → File exists  
    
  The error occurs when you try, for example, to rename a file with its previous name.

* errno.EFBIG → File too large  
    
  The error occurs when you try to create a file that is larger than the maximum allowed by the operating system.

* errno.EISDIR → Is a directory  
    
  The error occurs when you try to treat a directory name as the name of an ordinary file.

* errno.EMFILE → Too many open files  
    
  The error occurs when you try to simultaneously open more streams than acceptable for your operating system.

* errno.ENOENT → No such file or directory  
    
  The error occurs when you try to access a non-existent file/directory.

* errno.ENOSPC → No space left on device  
    
  The error occurs when there is no free space on the media.

# Key takeaways

1. A file needs to be **open** before it can be processed by a program, and it should be **closed** when the processing is finished.

Opening the file associates it with the **stream**, which is an abstract representation of the physical data stored on the media. The way in which the stream is processed is called **open mode**. **Three** open modes exist:

* **read mode** – only read operations are allowed;
* **write mode** – only write operations are allowed;
* **update mode** – both writes and reads are allowed.

2. Depending on the physical file content, different Python classes can be used to process files. In general, the BufferedIOBase is able to process any file, while TextIOBase is a specialized class dedicated to processing text files (i.e. files containing human-visible texts divided into lines using new-line markers). Thus, the streams can be divided into **binary** and **text** ones.

3. The following open() function syntax is used to open a file:  
  
open(file\_name, mode=open\_mode, encoding=text\_encoding)  
The invocation creates a stream object and associates it with the file named file\_name, using the specified open\_mode and setting the specified text\_encoding, or it **raises an exception in the case of an error**.

4. Three **predefined** streams are already open when the program starts:

* sys.stdin – standard input;
* sys.stdout – standard output;
* sys.stderr – standard error output.

5. The IOError exception object, created when any file operations fails (including open operations), contains a property named errno, which contains the completion code of the failed action. Use this value to diagnose the problem.

# read()

# readline()

# readlines()

for line in open('text.txt', 'rt'):

The **iteration protocol defined for the file object** is very simple - its \_\_next\_\_ method just **returns the next line read in from the file**.

Moreover, you can expect that the object automatically invokes close() when any of the file reads reaches the end of the file.

# write()

## **bytearray**

Python has more than one such container - one of them is **a specialized class name bytearray** - as the name suggests, it's **an array containing (amorphous) bytes**.

data = bytearray(10)

Note: such a constructor **fills the whole array with zeros**.

Bytearrays resemble lists in many respects. For example, they are **mutable**, they're a subject of the len() function, and you can access any of their elements using conventional indexing.

There is one important limitation - **you mustn't set any byte array elements with a value which is not an integer** (violating this rule will cause a TypeError exception) and you're **not allowed to assign a value that doesn't come from the range 0 to 255 inclusive** (unless you want to provoke a ValueError exception).

The write() method returns a number of successfully written bytes.

Reading from a binary file requires use of a specialized method name readinto(), as the method doesn't create a new byte array object, but fills a previously created one with the values taken from the binary file.

from os import strerror

data = bytearray(10)

for i in range(len(data)):

data[i] = 10 + i

try:

bf = open('file.bin', 'wb')

bf.write(data)

bf.close()

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

try:

bf = open('file.bin', 'rb')

bf.readinto(data)

bf.close()

for b in data:

print(hex(b), end=' ')

except IOError as e:

print("I/O error occurred:", strerror(e.errno))

bf = open('file.bin', 'rb')

data = bytearray(bf.read())

bf.close()

If the read() method is invoked with an argument, it **specifies the maximum number of bytes to be read**.

bf = open('file.bin', 'rb')

data = bytearray(bf.read(5))

bf.close()

**Key takeaways**

1. To read a file’s contents, the following stream methods can be used:

* read(number) – reads the number characters/bytes from the file and returns them as a string; is able to read the whole file at once;
* readline() – reads a single line from the text file;
* readlines(number) – reads the number lines from the text file; is able to read all lines at once;
* readinto(bytearray) – reads the bytes from the file and fills the bytearray with them;

2. To write new content into a file, the following stream methods can be used:

* write(string) – writes a string to a text file;
* write(bytearray) – writes all the bytes of bytearray to a file;

3. The open() method returns an iterable object which can be used to iterate through all the file's lines inside a for loop. For example:

for line in open("file", "rt"):

print(line, end='')

The code copies the file's contents to the console, line by line. **Note**: the stream closes itself **automatically** when it reaches the end of the file.

# os module

os.**name**

* **posix** — you'll get this name if you use Unix;
* **nt** — you'll get this name if you use Windows;
* **java** — you'll get this name if your code is written in Jython.

os.uname() only for Unix

* **systemname** — stores the name of the operating system;
* **nodename** — stores the machine name on the network;
* **release** — stores the operating system release;
* **version** — stores the operating system version;
* **machine** — stores the hardware identifier, e.g., x86\_64.

platform.uname() for all system

# Creating directories in Python

The *mkdir* function requires a path that can be relative or absolute.

* **my\_first\_directory** — this is a relative path which will create the *my\_first\_directory* directory in the current working directory;
* **./my\_first\_directory** — this is a relative path that explicitly points to the current working directory. It has the same effect as the path above;
* **../my\_first\_directory** — this is a relative path that will create the *my\_first\_directory* directory in the parent directory of the current working directory;
* **/python/my\_first\_directory** — this is the absolute path that will create the *my\_first\_directory* directory, which in turn is in the *python* directory in the root directory.

The ***mkdir*** function creates a directory in the specified path. Note that running the program twice will raise a *FileExistsError*.

To change the directory permissions, we recommend the ***chmod*** function

os.listdir()

The ***listdir*** function returns a list containing the names of the files and directories that are in the path passed as an argument. If no argument is passed to it, the current working directory will be used

The ***makedirs*** function enables recursive directory creation, which means that all directories in the path will be created.

os.makedirs("my\_first\_directory/my\_second\_directory")

To move between directories, you can use a function called ***chdir***, which changes the current working directory to the specified path.

os.getcwd ()

returns information about the current working directory.

To delete a single directory, you can use a function called **rmdir**, which takes the path as its argument.

To remove a directory and its subdirectories, you can use the **removedirs** function, which requires you to specify a path containing all directories that should be removed. As with the *rmdir* function, if one of the directories doesn't exist or isn't empty, an exception will be raised.

# The system() function

executes a command passed to it as a string.

returned\_value = os.system("mkdir my\_first\_directory")

print(returned\_value)

# Key takeaways

1. The uname function returns an object that contains information about the current operating system. The object has the following attributes:

* *systemname* (stores the name of the operating system)
* *nodename* (stores the machine name on the network)
* *release* (stores the operating system release)
* *version* (stores the operating system version)
* *machine* (stores the hardware identifier, e.g. x86\_64.)

2. The *name* attribute available in the os module allows you to distinguish the operating system. It returns one of the following three values:

* *posix* (you'll get this name if you use Unix)
* *nt* (you'll get this name if you use Windows)
* *java* (you'll get this name if your code is written in something like Jython)

3. The mkdir function creates a directory in the path passed as its argument. The path can be either relative or absolute, e.g:

import os

os.mkdir("hello") # the relative path

os.mkdir("/home/python/hello") # the absolute path

**Note**: If the directory exists, a FileExistsError exception will be thrown. In addition to the mkdir function, the os module provides the makedirs function, which allows you to recursively create all directories in a path.

4. The result of the listdir() function is a list containing the names of the files and directories that are in the path passed as its argument.

It's important to remember that the listdir function omits the entries '.' and '..', which are displayed, for example, when using the ls -a command on Unix systems. If the path isn't passed, the result will be returned for the current working directory.

5. To move between directories, you can use a function called chdir(), which changes the current working directory to the specified path. As its argument, it takes any relative or absolute path.

If you want to find out what the current working directory is, you can use the getcwd() function, which returns the path to it.

6. To remove a directory, you can use the rmdir() function, but to remove a directory and its subdirectories, use the removedirs() function.

7. On both Unix and Windows, you can use the system function, which executes a command passed to it as a string, e.g.:

import os

returned\_value = os.system("mkdir hello")

The system function on Windows returns the value returned by shell after running the command given, while on Unix it returns the exit status of the process.

# datetime module

from datetime import date

my\_date = date(2019, 11, 4)

print(my\_date)

|  |  |
| --- | --- |
| **Parameter** | **Restrictions** |
| year | The *year* parameter must be greater than or equal to 1 (MINYEAR constant) and less than or equal to 9999 (MAXYEAR constant). |
| month | The *month* parameter must be greater than or equal to 1 and less than or equal to 12. |
| day | The *day* parameter must be greater than or equal to 1 and less than or equal to the last day of the given month and year. |

from datetime import date

today = date.today()

print("Today:", today)

print("Year:", today.year)

print("Month:", today.month)

print("Day:", today.day)

Today: 2022-09-03

Year: 2022

Month: 9

Day: 3

# Creating a date object from a timestamp

from datetime import date

import time

timestamp = time.time()

print("Timestamp:", timestamp)

d = date.fromtimestamp(timestamp)

print("Date:", d)

Timestamp: 1662237544.962187

Date: 2022-09-03

# Creating a date object using the ISO format

from datetime import date

d = date.fromisoformat('2019-11-04')

print(d) # 2019-11-04

# The replace() method

The replace method returns a changed *date* object, so you must remember to assign it to some variable.

from datetime import date

d = date(1991, 2, 5)

print(d)

d = d.replace(year=1992, month=1, day=16)

print(d)

1991-02-05

1992-01-16

# What day of the week is it?

d = date(2019, 11, 4)

print(d.weekday())

0

# Creating time objects

|  |  |
| --- | --- |
| **Parameter** | **Restrictions** |
| hour | The *hour* parameter must be greater than or equal to 0 and less than 23. |
| minute | The *minute* parameter must be greater than or equal to 0 and less than 59. |
| second | The *second* parameter must be greater than or equal to 0 and less than 59. |
| microsecond | The *microsecond* parameter must be greater than or equal to 0 and less than 1000000. |
| tzinfo | The *tzinfo* parameter must be a tzinfo subclass object or None (default). |
| fold | The *fold* parameter must be 0 or 1 (default 0). |

from datetime import time

t = time(14, 53, 20, 1)

print("Time:", t)

print("Hour:", t.hour)

print("Minute:", t.minute)

print("Second:", t.second)

print("Microsecond:", t.microsecond)

Time: 14:53:20.000001

Hour: 14

Minute: 53

Second: 20

Microsecond: 1

# The time module

import time

class Student:

def take\_nap(self, seconds):

print("I'm very tired. I have to take a nap. See you later.")

time.**sleep**(seconds)

print("I slept well! I feel great!")

student = Student()

student.take\_nap(5)

# The ctime() function

**converts the time in seconds since January 1, 1970 (Unix epoch) to a string**.

import time

timestamp = 1572879180

print(time.ctime(timestamp))

Mon Nov 4 14:53:00 2019

# The gmtime() and localtime() functions

time.struct\_time:

tm\_year # specifies the year

tm\_mon # specifies the month (value from 1 to 12)

tm\_mday # specifies the day of the month (value from 1 to 31)

tm\_hour # specifies the hour (value from 0 to 23)

tm\_min # specifies the minute (value from 0 to 59)

tm\_sec # specifies the second (value from 0 to 61 )

tm\_wday # specifies the weekday (value from 0 to 6)

tm\_yday # specifies the year day (value from 1 to 366)

tm\_isdst # specifies whether daylight saving time applies (1 – yes, 0 – no, -1 – it isn't known)

tm\_zone # specifies the timezone name (value in an abbreviated form)

tm\_gmtoff # specifies the offset east of UTC (value in seconds)

The difference between them is that the gmtime function returns the *struct\_time* object in UTC, while the localtime function returns local time. For the gmtime function, the *tm\_isdst* attribute is always 0.

import time

timestamp = 1572879180

print(time.gmtime(timestamp))

print(time.localtime(timestamp))

time.struct\_time(tm\_year=2019, tm\_mon=11, tm\_mday=4, tm\_hour=14, tm\_min=53, tm\_sec=0, tm\_wday=0, tm\_yday=308, tm\_isdst=0)

time.struct\_time(tm\_year=2019, tm\_mon=11, tm\_mday=4, tm\_hour=14, tm\_min=53, tm\_sec=0, tm\_wday=0, tm\_yday=308, tm\_isdst=0)

# The asctime() and mktime() functions

asctime, converts a *struct\_time* object or a tuple to a string. Note that the familiar gmtime function is used to get the *struct\_time* object. If you don't provide an argument to the asctime function, the time returned by the localtime function will be used.

The second function called mktime converts a *struct\_time* object or a tuple that expresses the local time to the number of seconds since the Unix epoch.

import time

timestamp = 1572879180

st = time.gmtime(timestamp)

print(time.asctime(st))

print(time.mktime((2019, 11, 4, 14, 53, 0, 0, 308, 0)))

Mon Nov 4 14:53:00 2019

1572879180.0

# Creating datetime objects

|  |  |
| --- | --- |
| **Parameter** | **Restrictions** |
| year | The *year* parameter must be greater than or equal to 1 (MINYEAR constant) and less than or equal to 9999 (MAXYEAR constant). |
| month | The *month* parameter must be greater than or equal to 1 and less than or equal to 12. |
| day | The *day* parameter must be greater than or equal to 1 and less than or equal to the last day of the given month and year. |
| hour | The *hour* parameter must be greater than or equal to 0 and less than 23. |
| minute | The *minute* parameter must be greater than or equal to 0 and less than 59. |
| second | The *second* parameter must be greater than or equal to 0 and less than 59. |
| microsecond | The *microsecond* parameter must be greater than or equal to 0 and less than 1000000. |
| tzinfo | The *tzinfo* parameter must be a tzinfo subclass object or None (default). |
| fold | The *fold* parameter must be 0 or 1 (default 0). |

from datetime import datetime

dt = datetime(2019, 11, 4, 14, 53)

print("Datetime:", dt)

print("Date:", dt.date())

print("Time:", dt.time())

Datetime: 2019-11-04 14:53:00

Date: 2019-11-04

Time: 14:53:00

# Methods that return the current date and time

* today() — returns the current local date and time with the *tzinfo* attribute set to *None*;
* now() — returns the current local date and time the same as the *today* method, unless we pass the optional argument *tz* to it. The argument of this method must be an object of the *tzinfo* subclass;
* utcnow() — returns the current UTC date and time with the *tzinfo* attribute set to *None*.

from datetime import datetime

print("today:", datetime.today())

print("now:", datetime.now())

print("utcnow:", datetime.utcnow())

today: 2022-09-04 23:22:27.934853

now: 2022-09-04 23:22:27.935883

utcnow: 2022-09-04 20:22:27.935883

# Getting a timestamp

The timestamp method returns a float value expressing the number of seconds elapsed between the date and time indicated by the *datetime* object and January 1, 1970, 00:00:00 (UTC).

from datetime import datetime

dt = datetime(2020, 10, 4, 14, 55)

print("Timestamp:", dt.timestamp())

Timestamp: 1601823300.0

# Date and time formatting (part 1)

strftime returns the date and time in the format we specify.

* %Y – returns the year with the century as a decimal number. In our example, this is 2020.
* %m – returns the month as a zero-padded decimal number. In our example, it's 01.
* %d – returns the day as a zero-padded decimal number. In our example, it's 04.

from datetime import date

d = date(2020, 1, 4)

print(d.strftime('%Y/%m/%d'))

2020/01/04

# Date and time formatting (part 2)

%H returns the hour as a zero-padded decimal number, %M returns the minute as a zero-padded decimal number, while %S returns the second as a zero-padded decimal number.

The directive %y returns the year without a century as a zero-padded decimal number (in our example it's 20). The %B directive returns the month as the locale’s full name (in our example, it's November).

from datetime import time

from datetime import datetime

t = time(14, 53)

print(t.strftime("%H:%M:%S"))

dt = datetime(2020, 11, 4, 14, 53)

print(dt.strftime("%y/%B/%d %H:%M:%S"))

14:53:00

20/November/04 14:53:00

# The strftime() function in the time module

In the first function call, we format the *struct\_time* object, while in the second call (without the optional argument), we format the local time.

import time

timestamp = 1572879180

st = time.gmtime(timestamp)

print(time.strftime("%Y/%m/%d %H:%M:%S", st))

print(time.strftime("%Y/%m/%d %H:%M:%S"))

2019/11/04 14:53:00

2022/09/04 20:55:27

# The strptime() method

 it creates a datetime object from a string representing a date and time.

from datetime import datetime

print(datetime.strptime("2019/11/04 14:53:00", "%Y/%m/%d %H:%M:%S"))

2019-11-04 14:53:00

# Date and time operations

Sooner or later you'll have to perform some calculations on the date and time. Fortunately, there's a class called timedelta in the datetime module that was created for just such a purpose.

To create a timedelta object, just do subtraction on the date or datetime objects, just like we did in the example in the editor.

from datetime import date

from datetime import datetime

d1 = date(2020, 11, 4)

d2 = date(2019, 11, 4)

print(d1 - d2)

dt1 = datetime(2020, 11, 4, 0, 0, 0)

dt2 = datetime(2019, 11, 4, 14, 53, 0)

print(dt1 - dt2)

366 days, 0:00:00

365 days, 9:07:00

# Creating timedelta objects

from datetime import timedelta

delta = timedelta(weeks=2, days=2, hours=3)

print(delta)

16 days, 3:00:00

# Creating timedelta objects: continued

The timedelta object can be multiplied by an integer. In our example, we multiply the object representing 16 days and 2 hours by 2. As a result, we receive a timedelta object representing 32 days and 4 hours.

from datetime import timedelta

from datetime import date

from datetime import datetime

delta = timedelta(weeks=2, days=2, hours=2)

print(delta)

delta2 = delta \* 2

print(delta2)

d = date(2019, 10, 4) + delta2

print(d)

dt = datetime(2019, 10, 4, 14, 53) + delta2

print(dt)

16 days, 2:00:00

32 days, 4:00:00

2019-11-05

2019-11-05 18:53:00

# Key takeaways

1. To create a date object, you must pass the year, month, and day arguments as follows:

from datetime import date

my\_date = date(2020, 9, 29)

print("Year:", my\_date.year) # Year: 2020

print("Month:", my\_date.month) # Month: 9

print("Day:", my\_date.day) # Day: 29

The date object has three (read-only) attributes: year, month, and day.

2. The today method returns a date object representing the current local date:

from datetime import date

print("Today:", date.today()) # Displays: Today: 2020-09-29

3. In Unix, the timestamp expresses the number of seconds since January 1, 1970, 00:00:00 (UTC). This date is called the "Unix epoch", because it began the counting of time on Unix systems. The timestamp is actually the difference between a particular date (including time) and January 1, 1970, 00:00:00 (UTC), expressed in seconds. To create a date object from a timestamp, we must pass a Unix timestamp to the fromtimestamp method:

from datetime import date

import time

timestamp = time.time()

d = date.fromtimestamp(timestamp)

Note: The time function returns the number of seconds from January 1, 1970 to the current moment in the form of a float number.

4. The constructor of the time class accepts six arguments (*hour*, *minute*, *second*, *microsecond*, *tzinfo*, and *fold*). Each of these arguments is optional.

from datetime import time

t = time(13, 22, 20)

print("Hour:", t.hour) # Hour: 13

print("Minute:", t.minute) # Minute: 22

print("Second:", t.second) # Second: 20

5. The time module contains the sleep function, which suspends program execution for a given number of seconds, e.g.:

import time

time.sleep(10)

print("Hello world!") # This text will be displayed after 10 seconds.

6. In the datetime module, date and time can be represented either as separate objects, or as one object. The class that combines date and time is called *datetime*. All arguments passed to the constructor go to read-only class attributes. They are *year*, *month*, *day*, *hour*, *minute*, *second*, *microsecond*, *tzinfo*, and *fold*:

from datetime import datetime

dt = datetime(2020, 9, 29, 13, 51)

print("Datetime:", dt) # Displays: Datetime: 2020-09-29 13:51:00

7. The strftime method takes only one argument in the form of a string specifying a format that can consist of directives. A directive is a string consisting of the character % (percent) and a lower-case or upper-case letter. Below are some useful directives:

* %Y – returns the year with the century as a decimal number;
* %m – returns the month as a zero-padded decimal number;
* %d – returns the day as a zero-padded decimal number;
* %H – returns the hour as a zero-padded decimal number;
* %M – returns the minute as a zero-padded decimal number;
* %S – returns the second as a zero-padded decimal number.

Example:

from datetime import date

d = date(2020, 9, 29)

print(d.strftime('%Y/%m/%d')) # Displays: 2020/09/29

8. It's possible to perform calculations on date and datetime objects, e.g.:

from datetime import date

d1 = date(2020, 11, 4)

d2 = date(2019, 11, 4)

d = d1 - d2

print(d) # Displays: 366 days, 0:00:00.

print(d \* 2) # Displays: 732 days, 0:00:00.

The result of the subtraction is returned as a timedelta object that expresses the difference in days between the two dates in the example above.

Note that the difference in hours, minutes, and seconds is also displayed. The timedelta object can be used for further calculations (e.g. you can multiply it by 2).

# calendar module

import calendar

print(calendar.calendar(2020))

2020

January February March

Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su

1 2 3 4 5 1 2 1

6 7 8 9 10 11 12 3 4 5 6 7 8 9 2 3 4 5 6 7 8

13 14 15 16 17 18 19 10 11 12 13 14 15 16 9 10 11 12 13 14 15

20 21 22 23 24 25 26 17 18 19 20 21 22 23 16 17 18 19 20 21 22

27 28 29 30 31 24 25 26 27 28 29 23 24 25 26 27 28 29

30 31

April May June

Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su

1 2 3 4 5 1 2 3 1 2 3 4 5 6 7

6 7 8 9 10 11 12 4 5 6 7 8 9 10 8 9 10 11 12 13 14

13 14 15 16 17 18 19 11 12 13 14 15 16 17 15 16 17 18 19 20 21

20 21 22 23 24 25 26 18 19 20 21 22 23 24 22 23 24 25 26 27 28

27 28 29 30 25 26 27 28 29 30 31 29 30

July August September

Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su

1 2 3 4 5 1 2 1 2 3 4 5 6

6 7 8 9 10 11 12 3 4 5 6 7 8 9 7 8 9 10 11 12 13

13 14 15 16 17 18 19 10 11 12 13 14 15 16 14 15 16 17 18 19 20

20 21 22 23 24 25 26 17 18 19 20 21 22 23 21 22 23 24 25 26 27

27 28 29 30 31 24 25 26 27 28 29 30 28 29 30

31

October November December

Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su

1 2 3 4 1 1 2 3 4 5 6

5 6 7 8 9 10 11 2 3 4 5 6 7 8 7 8 9 10 11 12 13

12 13 14 15 16 17 18 9 10 11 12 13 14 15 14 15 16 17 18 19 20

19 20 21 22 23 24 25 16 17 18 19 20 21 22 21 22 23 24 25 26 27

26 27 28 29 30 31 23 24 25 26 27 28 29 28 29 30 31

30

print(calendar.calendar(2021, w=2, l=1, c=3, m=4))

* w – date column width (default 2)
* l – number of lines per week (default 1)
* c – number of spaces between month columns (default 6)
* m – number of columns (default 3)

calendar.prcal(2020)

print(calendar.month(2020, 11))

allows you to display a calendar for a specific month.

* w – date column width (default 2)
* l – number of lines per week (default 1)

calendar.prmonth(2020, 5)

# The setfirstweekday() function

calendar.setfirstweekday(calendar.SUNDAY)

calendar.prmonth(2020, 12)

December 2020

Su Mo Tu We Th Fr Sa

1 2 3 4 5

6 7 8 9 10 11 12

13 14 15 16 17 18 19

20 21 22 23 24 25 26

27 28 29 30 31

# The weekday() function

returns the day of the week as an integer value for the given year, month, and day.

# The weekheader() function

import calendar

print(calendar.weekheader(2))

Mo Tu We Th Fr Sa Su

# How do we check if a year is a leap year?

print(calendar.isleap(2020))

print(calendar.leapdays(2010, 2021)) # Up to but not including 2021.

True

3

# Classes for creating calendars

* calendar.Calendar – provides methods to prepare calendar data for formatting;
* calendar.TextCalendar – is used to create regular text calendars;
* calendar.HTMLCalendar – is used to create HTML calendars;
* calendar.LocalTextCalendar – is a subclass of the calendar.TextCalendar class. The constructor of this class takes the *locale* parameter, which is used to return the appropriate months and weekday names.
* calendar.LocalHTMLCalendar – is a subclass of the calendar.HTMLCalendar class. The constructor of this class takes the locale parameter, which is used to return the appropriate months and weekday names.

The Calendar class constructor takes one optional parameter named firstweekday, by default equal to 0 (Monday).

c = calendar.Calendar(calendar.SUNDAY)

# The itermonthdates() method

for date in c.itermonthdates(2019, 11):

print(date, end=" ")

2019-10-31 2019-11-01 2019-11-02 ...

for xxx in c.itermonthdays3(2022, 12):  
 print(xxx, end=" ")

(2022, 11, 30) (2022, 12, 1) (2022, 12, 2) …

* itermonthdates2 – returns days in the form of tuples consisting of a day of the month number and a week day number;
* itermonthdates3 – returns days in the form of tuples consisting of a year, a month, and a day of the month numbers. This method has been available since version 3.7;
* itermonthdates4 – returns days in the form of tuples consisting of a year, a month, a day of the month, and a day of the week numbers. This method has been available since Python version 3.7.

# The monthdays2calendar() method

for data in c.monthdays2calendar(2022, 12):  
 print(data)

[(0, 0), (0, 1), (0, 2), (1, 3), (2, 4), (3, 5), (4, 6)]

[(5, 0), (6, 1), (7, 2), (8, 3), (9, 4), (10, 5), (11, 6)]

[(12, 0), (13, 1), (14, 2), (15, 3), (16, 4), (17, 5), (18, 6)]

[(19, 0), (20, 1), (21, 2), (22, 3), (23, 4), (24, 5), (25, 6)]

[(26, 0), (27, 1), (28, 2), (29, 3), (30, 4), (31, 5), (0, 6)]

# Key takeaways

1. In the calendar module, the days of the week are displayed from Monday to Sunday. Each day of the week has its representation in the form of an integer, where the first day of the week (Monday) is represented by the value 0, while the last day of the week (Sunday) is represented by the value 6.

2. To display a calendar for any year, call the calendar function with the year passed as its argument, e.g.:

import calendar

print(calendar.calendar(2020))

Note: A good alternative to the above function is the function called prcal, which also takes the same parameters as the calendar function, but doesn't require the use of the print function to display the calendar.

3. To display a calendar for any month of the year, call the month function, passing year and month to it. For example:

import calendar

print(calendar.month(2020, 9))

Note: You can also use the prmonth function, which has the same parameters as the month function, but doesn't require the use of the print function to display the calendar.

4. The setfirstweekday function allows you to change the first day of the week. It takes a value from 0 to 6, where 0 is Sunday and 6 is Saturday.

5. The result of the weekday function is a day of the week as an integer value for a given year, month, and day:

import calendar

print(calendar.weekday(2020, 9, 29)) # This displays 1, which means Tuesday.

6. The weekheader function returns the weekday names in a shortened form. The weekheader method requires you to specify the width in characters for one day of the week. If the width you provide is greater than 3, you'll still get the abbreviated weekday names consisting of only three characters. For example:

import calendar

print(calendar.weekheader(2)) # This display: Mo Tu We Th Fr Sa Su

7. A very useful function available in the calendar module is the function called isleap, which, as the name suggests, allows you to check whether the year is a leap year or not:

import calendar

print(calendar.isleap(2020)) # This displays: True

8. You can create a calendar object yourself using the Calendar class, which, when creating its object, allows you to change the first day of the week with the optional firstweekday parameter, e.g.:

import calendar

c = calendar.Calendar(2)

for weekday in c.iterweekdays():

print(weekday, end=" ")

# Result: 2 3 4 5 6 0 1

The iterweekdays returns an iterator for weekday numbers. The first value returned is always equal to the value of the firstweekday property.