Python Institute: Python Essentials 2

Networking Academy

This certificate is awarded to

David Stacey

for successfully completing

Python Essentials 2

offered by Per Scholas Inc.
through the Cisco Networking Academy program.

Aikerim Daniiar Kyzy
Instructor
Per Scholas Inc.

24 Dec 2025
Completion Date

If you're looking for information on random number generators, be sure to check this out. It's long..
https://en.wikipedia.org/wiki/Mersenne_Twister

1.0 - Welcome to Python Essentials 2

If you cannot answer **all three** instantly, you **don't understand** the line yet:

1. *When does this execute?*
2. *What state does it mutate or depend on?*
3. *What breaks if I remove it?*

Syllabus

In this course you will learn:

- how to adopt general coding techniques and best practices in your projects;
- how to process strings;
- how to use object-oriented programming in Python;
- how to import and use Python modules, including the math, random, platform, os, time, datetime, and calendar modules;
- how to create and use your own Python modules and packages;
- how to use the exception mechanism in Python;
- how to use generators, iterators, and closures in Python;
- how to process files.

The course is divided into four modules:

1. Module 1
   Modules, Packages and PIP;
2. Module 2
   Strings, string and list methods, and exceptions;
3. Module 3
   Object-Oriented Programming;
4. Module 4
   Miscellaneous (generators, iterators, closures, file streams, processing text and binary files, the os, time, datetime, and calendar module).

Module 1: Modules, Packages and PIP

1.1 - Introduction to modules in Python

What is a module?

A file containing Python definitions and statements, which can be later imported and used when necessary.

The handling of modules consists of two different issues:

- The first (probably the most common) happens when you want to use an already existing module, written by someone else, or created by yourself during your work on some complex project - in this case you are the module's user;
- The second occurs when you want to create a brand new module, either for your own use, or to make other programmers' lives easier - you are the module's supplier.

Importing a module

Let's assume that you want to use two entities provided by the math module:

- A symbol (constant) representing a precise value of π
  - although using a Greek letter to name a variable is fully possible in Python, the symbol is named pi - it's a more convenient solution
- A function named sin() (the computer equivalent of the mathematical sine function)

```
import math
import sys
```

OR

```
import math, sys
```

- The instruction may be located anywhere in your code, but it must be placed before the first use of any of the **module's entities**.

It's simple, you put:

- the name of the **module** (e.g., math)
- a **dot** (i.e., .)
- the name of the **entity** (e.g., pi)

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

Such a form clearly indicates the namespace in which the name exists.

This qualification is compulsory if a module has been imported by the import module instruction.

Namespace

Inside a certain namespace, each name must remain unique.

Importing Continued

The method to import a module while **avoiding conflict with namespace**:

```
from math import pi
```

(**keyword** module **keyword** entity/entities)

The instruction has this effect:

- The **listed entities** (**only those ones**) are imported from the indicated module
- The names of the imported entities are accessible without qualification.

A method that doesn't work:

```
print(math.e)
```

Importing a module: *

```
from module import *
```

the import's syntax is a more aggressive form of the previously presented one

- This imports all of the module's entities.

Aliasing

A module's name can be given an **alias** to invoke it with:

```
import module as alias
```

Similarly, to change an **entity's** name, and can be **repeated using commas**:

```
from module import name as alias, n as a, m as b, o as c
```

As an example:

```
from math import pi as PI, sin as sine
```

1.2 - Py Modules: Math, Random, Platform

Standard modules

dir(module)
Used to invoke a list of all the known parameters of a particular module.

```
import math

for name in dir(math):
  print(name, end="        ")
```

Functions from math

The math module couples **more than 50 symbols** (functions and constants) that perform mathematical operations (like sine(), pow(), factorial()) or provide important values (like π and the Euler symbol e).

The first group of the math's functions are connected with **trigonometry**:

- **All these functions take one argument** and **return the appropriate result**
  - **sin(x)** → the sine of x;
  - **cos(x)** → the cosine of x;
  - **tan(x)** → the tangent of x.
- **(be careful with tan() - not all arguments are accepted).**

(an angle measurement expressed in radians)

Of course, there are also their **inverted versions**:

- **These functions take one argument** and return a **measure of an angle in radians**
- **asin(x)** → the arcsine of x;
- **acos(x)** → the arccosine of x;
- **atan(x)** → the arctangent of x.

(mind the domains):

Domains (valid inputs)
- **asin(x)** → **x** ∈ **[-1, 1]**
- **acos(x)** → **x** ∈ **[-1, 1]**
- **atan(x)** → **x** ∈ **(-∞, +∞)** (any real number)

Why?
- **Sine** and **cosine never exceed -1 or +1**, so **their inverses can't accept anything outside that range**.
- **Tangent** can produce **any real value**, so **arctangent accepts anything** you throw at it (within floating-point sanity).

Ranges (what they return, in radians)
- **asin(x)** → `[-π/2, +π/2]`
- **acos(x)** → `[0, π]`
- **atan(x)** → `(-π/2, +π/2)`

To effectively operate on angle measurements, the math module provides you with the following entities:

- **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)

Example:

```
import math

math.asin(1)    # OK
math.asin(1.1)  # ValueError: math domain error
math.atan(9999) # Totally fine
```

IT-practical takeaway:

- Clamp inputs before **asin** / **acos** if values come from **sensors**, **floats**, or **user input**.
- Domain errors aren't bugs — they're math enforcing reality, like a firewall for numbers.

So if you run this code, you'll see 4x True outputs:

```
from math import pi, radians, degrees, sin, cos, tan, asin

ad = 90
ar = radians(ad)
ad = degrees(ar)

print(ad == 90.)
print(ar == pi / 2.)
print(sin(ar) / cos(ar) == tan(ar))
print(asin(sin(ar)) == ar)
```

Apart from the circular functions (listed above) the math module also contains a set of their hyperbolic analogues:

- **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.

Another group of the math's functions is formed by functions which are connected with exponentiation:

- **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))

Note: the **pow()** function:

- **pow(x, y)** → finding the value of xy (mind the domains)

This is a built-in function, and doesn't have to be imported.

The last group consists of some general-purpose functions like:

- **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)

The random module

A module that delivers some mechanisms allowing you to operate with pseudorandom numbers.

The random module groups **more than 60 entities** designed to help you use pseudo-random numbers

A random number generator takes a value called a **seed**

- Treats it as an input value
    - Calculates a "random" number based on it (the method depends on a chosen algorithm)
        - Produces a new **seed** value.

The **random factor** of the process may be **augmented by setting the seed with a number taken from the current time** - this may ensure that each program launch will start from a different seed value

- **Fortunately, such an initialization is done by Python during module import.**

Functions from random

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)**.

The example program below will produce five pseudorandom values - as their values are determined by the current (rather unpredictable) seed value, you can't guess them:

```
from random import random

for i in range(5):
  print(random())
```

– – Continued – –

The seed function

The **seed()** function is able to directly set the generator's seed. We'll show you two of its variants:

seed() - sets the seed with the current time;

seed(int_value) - sets the seed with the integer value int_value.

This input removes all traces of randomness from the code:

```
seed(0)
```

The randrange and randint functions

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

- **randrange(end)**
- **randrange(beg, end)**
- **randrange(beg, end, step)**
- **randint(left, right)**

The first three invocations will generate an integer taken (pseudorandomly) from the range (respectively):

- **range(end)**
- **range(beg, end)**
- **range(beg, end, step)**
  - This is equivalent of **randrange(left, right+1)** - it generates the **integer value i**, which falls in the range [left, right] (no exclusion on the right side).

Note the implicit right-sided exclusion!

This sample program will consequently output a line consisting of three zeros and either a zero or one at the fourth place.

```
from random import randrange, randint

print(randrange(1), end=' ')
print(randrange(0, 1), end=' ')
print(randrange(0, 1, 1), end=' ')
print(randint(0, 1))
```

This outputs: **0 0 0 1**

This program very likely outputs a set of numbers in which some elements are not unique:

```
from random import randint

for i in range(10):
  print(randint(1, 10), end=',')
```

The choice and sample functions

A better solution than writing your own code to check the uniqueness of the "drawn" numbers.

- **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.

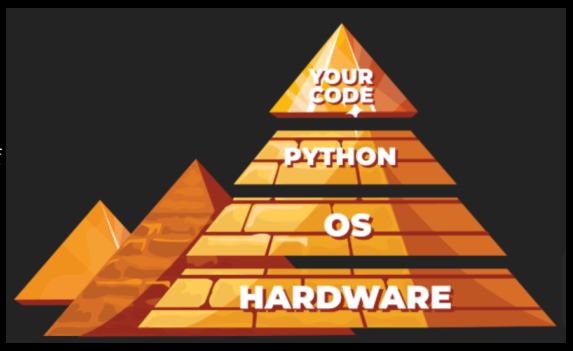
In other words, the function chooses some of the input elements, returning a list with the choice. The elements in the sample are placed in random order.

- 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))
```

The platform module

The platform module contains about 70 functions which let you dive into the underlying layers of the OS and hardware.



Imagine your program's environment as a pyramid consisting of a number of layers or platforms.

The layers are:

- Your (running) code is located at the top of it
- Python (more precisely - its runtime environment)
- The OS (operating system) - Python's environment provides some of its functionalities using the operating system's services; Python, although very powerful, isn't omnipotent - it's forced to use many helpers if it's going to process files or communicate with physical devices;
- Hardware - the processor (or processors), network interfaces, human interface devices (mice, keyboards, etc.) and all other machinery needed to make the computer run; the OS knows how to drive it, and uses lots of tricks to conduct all parts in a consistent rhythm.

This means than some of your program's actions have to travel a long way to be successfully performed:

- Your code wants to create a file, so it invokes one of Python's functions;
- Python accepts the order, rearranges it to meet local OS requirements (it's like putting the stamp "approved" on your request) and sends it down (this may remind you of a chain of command)
- The OS checks if the request is reasonable and valid (e.g., whether the file name conforms to some syntax rules) and tries to create the file; such an operation, seemingly very simple, isn't atomic - it consists of many minor steps taken by...
- The hardware, which is responsible for activating storage devices (hard disk, solid state devices, etc.) to satisfy the OS's needs.

Functions from platform

The platform function
The platform module lets you access the underlying platform's data

The platform function can show you all the underlying layers in one glance, named platform, too.
- It returns a string describing the environment

```
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)

```
from platform import platform

print(platform())
print(platform(1))
print(platform(0, 1))
```

The machine function
Will give the generic name of the processor which runs your OS together with Python and your code.

```
from platform import machine

print(machine())
```
Outputs: x86_64 on Dave Stacey's Linux

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

```
from platform import processor

print(processor())
```

The system function
Returns the generic OS name as a string.

```
from platform import system

print(system())
```

The Version function
Returns the OS version.

The python_implementation and the python_version_tuple functions

- **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.

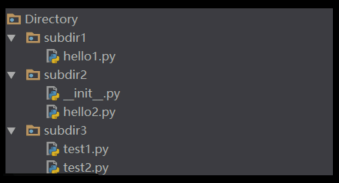
```
from platform import python_implementation, python_version_tuple

print(python_implementation())

for atr in python_version_tuple():
  print(atr)
```

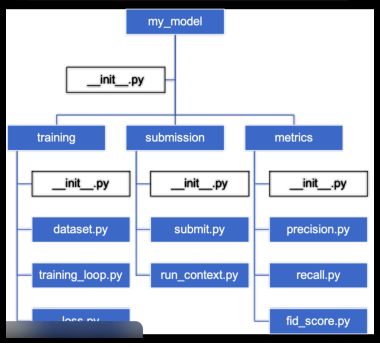
Python Module Index

You can read about all standard Python modules here: https://docs.python.org/3/py-modindex.html.

Don't worry - you won't need all these modules. Many of them are very specific.
All you need to do is find the modules you want, and teach yourself how to use them. It's easy.

1.3 - Modules and Packages



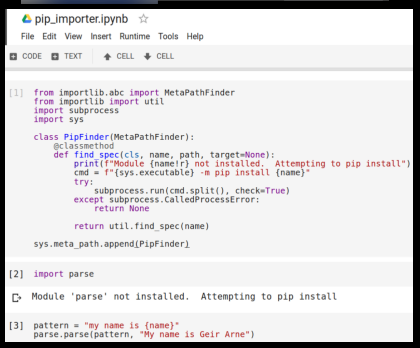


What is a package?

A Python package is a **structured directory that groups related modules together,** allowing large programs to be organized, reused, and imported cleanly.

3 bullet-points to lock it in:

- **Organizational container:**
  A package is to modules what a folder is to files—used to group related functionality logically and avoid chaos as projects scale.
- **Namespace control:**
  Packages prevent naming conflicts by creating clear paths (e.g., `network.tools.scan`), which is critical in large codebases and shared environments.
- **Distribution & reuse:**
  Packages are the standard unit for sharing code (internal or public), forming the basis of tools installed via `pip` and used across systems.



Your first module/Your first package

1. While a **module** is designed to couple together some related entities such as functions, variables, or constants, a package is a container which enables the coupling of several related modules under one common name. Such a container can be distributed as-is (as a batch of files deployed in a directory sub-tree) or it can be packed inside a zip file.

2. During the very first import of the actual module, Python translates its source code into a semi-compiled format stored inside the pyc files, and deploys these files into the **__pycache__** directory located in the module's home directory.

3. If you want to tell your module's user that a particular entity should be treated as private (i.e. not to be explicitly used outside the module) you can mark its name with either the _ or __ prefix. Don't forget that this is only a recommendation, not an order.

4. 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.

5. If you want to 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.

6. A Python file named **__init__.py** is implicitly run when a package containing it is subject to import, and is used to initialize a package and/or its sub-packages (if any). The file may be empty, but must not be absent.

EXTRA: To tell Python where to look on Mac:
$ nano ~/.bash_profile

- Add: **export PYTHONPATH = "/Users/..."**

EXTRA: To tell Python where to look on Windows:
> set PYTHONPATH=C:\path\to\your\modules

Permanent via Windows Environment Variables UI:

1. Open **Start** → search **Edit the system environment variables** → click **Environment Variables...**.
2. Under **User variables** (or **System variables**), click **New**.
    - **Name:** `PYTHONPATH`
    - **Value:** `C:\path\to\your\modules`
      (Add multiple paths separated by semicolons ; if needed.)
3. Click **OK** and reopen your terminal/IDE. Python will now see those directories on import.

OR (alternative):

$env:PYTHONPATH = "C:\path\to\your\modules"

Explanation (professional practice):
Python builds a list of directories to search for imports stored in `sys.path` when it starts. Anything you put in the `PYTHONPATH` environment variable gets inserted early in that list, so Python will search your specified folders before the default ones. Setting `PYTHONPATH` via the Windows system environment variables makes those custom paths available across all new command shells and IDE sessions, which is usually what you want during development. W3docs

Keep in mind that **over-using** `PYTHONPATH` can lead to confusion in larger projects; for real applications it's often better to install your code as a package or use virtual environments so that imports resolve correctly.

1.4 - Python Package Installer

1. A **repository** (or **repo** for short) is designed to collect and share free Python code that exists and works under the name Python Package Index (PyPI) although it's also likely that you come across the very niche name The Cheese Shop. The Shop's website is available at https://pypi.org/.

2. To make use of The Cheese Shop, a specialized tool has been created and its name is pip (pip installs packages while pip stands for... ok, never mind). As pip may not be deployed as a part of the standard Python installation, it is possible that you will need to install it manually. Pip is a console tool.

3. To check pip's version one the following commands should be issued:

pip --version
or
pip3 --version ←this one for Ubuntu in 2025

Check yourself which of these works for you in your OS's environment, then, on Debian based Linuxes:
$ sudo apt install python3-pip

4. A list of the main pip activities in **terminal** looks as follows (**works on Ubuntu 24.020**):

- **pip help operation** – shows a brief description of pip;
- **pip list** – shows a list of the currently installed packages;
- **pip show package_name** – shows package_name info including the package's dependencies;
- **pip search anystring** –❌ *deprecated / removed*; **use https://pypi.org/search (browser)** or `pip index versions package_name;`
- **pip install name** – installs name system-wide (expect problems when you don't have administrative rights);
- **pip install --user name** – installs name for you only; no other platform user will be able to use it;
- **pip install -U name** – updates a previously installed package;
- **pip uninstall name** – uninstalls a previously installed package.

This is the simple format for installing packages on Linux Ubuntu:
$ cd ~/Documents/Production/Python101 "Python101" is the folder containing the main.py
$ python3 -m venv .venv
$ source .venv/bin/activate
$ pip install pygame

This is ChatGPT 5.2's example "ideal setup" to install a package dependency:
$ cd ~/Documents/Production
$ mkdir -p my_pygame_project
$ cd my_pygame_project

$ /usr/bin/python3 -m venv .venv
$ source .venv/bin/activate
$ python -m pip install -U pip
$ python -m pip install pygame
$ python -m pygame.examples.aliens

Two simple habits to keep everything clean and predictable:

- Activate the virtual environment before installing anything:
  source .venv/bin/activate

- Install packages through Python, not bare pip:
  python -m pip install <thing>

Example code and explanation: **line**

```
import pygame      line 1

run = True         line 3
width = 400        line 4
height = 100       line 5
pygame.init()      line 6
screen = pygame.display.set_mode((width, height))       line 7
font = pygame.font.SysFont(None, 48)                          line 8
text = font.render("Welcome to pygame", True, (255, 255, 255))      line 9
screen.blit(text, ((width - text.get_width()) // 2, (height -
text.get_height()) // 2))            line 10
pygame.display.flip()                line 11
while run:                           line 12
  for event in pygame.event.get():
   if event.type == pygame.QUIT\
   or event.type == pygame.MOUSEBUTTONUP\
   or event.type == pygame.KEYUP:
    run = False
```

Let's comment on it briefly.

- line 1: import pygame and let it serve us;
- line 3: the program will run as long as the run variable is True;
- lines 4 and 5: determine the window's size;
- line 6: initialize the pygame environment;
- line 7: prepare the application window and set its size;
- line 8: make an object representing the default font of size 48 points;
- line 9: make an object representing a given text – the text will be anti-aliased (True) and white (255,255,255)
- line 10: insert the text into the (currently invisible) screen buffer;
- line 11: flip the screen buffers to make the text visible;
- line 12: the pygame main loop starts here;
- line 13: get a list of all pending pygame events;
- lines 14 through 16: check whether the user has closed the window or clicked somewhere inside it or pressed any key;
- line 15: if yes, stop executing the code.

Module 2 - Strings, String an dLists Methods, Exceptions

2.1 - Characters and Strings vs Computers

Python 3 fully supports Unicode and UTF-8:

- you can use Unicode/UTF-8 encoded characters to name variables and other entities;
- you can use them during all input and output.

This means that Python3 is completely I18Ned (read on).

ASCII - The **American Standard Code for Information Interchange**

- Used mainly to encode the Latin alphabet and some of its derivates
- Provides up to **256 different characters**
- I18N is the current ASCII Standard
    - It stands for Internationationalization in the way that there's an I 18 letter and an N

Code Point

A number which makes a character.

- 32 is a code point which makes a space in ASCII encoding.
    - We can say that **standard ASCII code consists of 128 code points**

Code Page

A standard for using the upper 128 code points to store specific national characters

- For example, the code point 200 makes Č (a letter used by some Slavic languages) when utilized by the ISO/IEC 8859-2 code page
- It makes Ш (a Cyrillic letter) when used by the ISO/IEC 8859-5 code page.

Character	Code	Character	Code	Character	Code	Character	Code
(NUL)	0	(space)	32	@	64	`	96
(SOH)	1	!	33	A	65	a	97
(STX)	2	"	34	B	66	b	98
(ETX)	3	#	35	C	67	c	99
(EOT)	4	$	36	D	68	d	100
(ENQ)	5	%	37	E	69	e	101
(ACK)	6	&	38	F	70	f	102

(CAN)	24	8	56	X	88	x	120
(EM)	25	9	57	Y	89	y	121
(SUB)	26	:	58	Z	90	z	122
(ESC)	27	;	59	[	91	{	123
(FS)	28	<	60	\	92	|	124
(GS)	29	=	61	]	93	}	125
(RS)	30	>	62	^	94	~	126
(US)	31	?	63	_	95		127

Unicode

Assigns unique (unambiguous) characters to more than a million code points..

- Able to encode virtually all alphabets being used by humans
- The first 128 Unicode code points are identical to ASCII
- The first 256 Unicode code points are identical to the ISO/IEC 8859-1 code page (a code page designed for western European languages).

UCS-4 (Universal Character Set)

- A substandard of Unicode
- 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**)
    - Considered a wasteful standard - it increases a text's size by four times compared to standard ASCII

UTF-8 (Unicode Transformation Format)

uses as many bits for each of the code points as it really needs to represent them.

- all Latin characters (and all standard ASCII characters) occupy eight bits;
- non-Latin characters occupy 16 bits;
- CJK (China-Japan-Korea) ideographs occupy 24 bits.

2.2 - The nature of Strings

Be sure to check the list of string() methods: https://docs.python.org/3.4/library/stdtypes.html#string-methods

String Review

- Take a look at Example 1. The len() function used for strings returns a number of characters contained by the arguments. The snippet outputs 2.
- Any string can be empty. Its length is 0 then – just like in Example 2.
- Don't forget that a backslash (\) used as an escape character is not included in the string's total length. The code in Example 3, therefore, outputs 3.

```
# Example 1
word = 'by'
print(len(word))

# Example 2
empty = ''
print(len(empty))

# Example 3
i_am = 'I\'m'
print(len(i_am))
```

Multiline Strings

Use ‘‘‘ (triple parenthesis) ‘‘‘ to enclose multiline text in a string.

```
multiline = """Line #1
Line #2"""

print(len(multiline))
```

Operations on Strings

In general, strings can be:

- concatenated (joined)
- replicated.

The ability to use the same operator against completely different kinds of data (like numbers vs. strings) is called overloading (as such an operator is overloaded with different duties).

```
str1 = 'a'
str2 = 'b'

print(str1 + str2)
print(str2 + str1)
print(5 * 'a')
print('b' * 4)
```

ord() (as in ordinal)
Used to output a specific character's ASCII/UNICODE code point value.

- Only takes a single character as its argument.

```
char_1 = 'a'
char_2 = ' '  # space

print(ord(char_1))
print(ord(char_2))
```

chr() (as in character)
This function takes a code point and returns a character.

```
print(chr(945))
```

Indexing

Chat GPT 5.2 taught this better:
Indexing (Python strings)
Indexing is how you grab a specific character from a string by its position using square brackets (`[]`)—because strings are **sequence** types and support the same "pick item #N" behavior as other sequences. Python documentation+1

- **Zero-based positions:** the first character is at index `0`, then `1`, `2`, etc. FreeCodeCamp+1
- **Negative indexing works:** `-1` means "last character," `-2` is "second-to-last," etc. Jessica Temporal+1
- **It's read-only for strings:** you can *access* `the_string[ix]`, but you can't assign into it (`the_string[0] = 'S'` fails) because strings are immutable. Real Python+1

Slices

ChatGPT did it again for this one:
Slices (Python strings)
A *slice* lets you take a **substring** from a string using the `[start:stop:step]` format—because strings are sequences, slicing works the same way it does on lists/tuples (but returns a **new** string).

- **`start` is inclusive, `stop` is exclusive:** `s[2:5]` grabs positions 2,3,4 — never 5. This is why slices chain cleanly without off-by-one chaos.
- **Defaults + negatives are built-in:** `s[:4]` (from start), `s[4:]` (to end), `s[:-1]` (everything except last).
- **`step` controls the stride (and can reverse):** `s[::2]` takes every 2nd char; `s[::-1]` reverses the string. Strings still stay immutable—slicing doesn't "change" them, it **creates** a new one.

```
s = "silly walks"
print(s[0:5])    # "silly"
print(s[6:])     # "walks"
print(s[::-1])   # "sklaw yllis"
```

in and not in operators on Strings

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

print("f" in alphabet)
print("F" in alphabet)
print("ghi" not in alphabet)
print("Xyz" not in alphabet)
```

Results in the following output:
False
True
False
True

Python Strings are Immutable

You cannot delete elements, only the whole string.

You cannot use **append()** or **insert()**

Operations (Continued)

min()

A function that finds the minimum element of the sequence passed as an argument.

- The sequence CANNOT BE EMPTY, ValueErrors are the result

```
# Demonstrating min() - Example 1:
print(min("aAbByYzZ"))

# Demonstrating min() - Examples 2 & 3:
t = 'The Knights Who Say "Ni!"'
print('[' + min(t) + ']')

t = [0, 1, 2]
print(min(t))
```

In Example 1: A is the first letter in order of the ASCII code.
In Example 2: Prints a white character, or space, which is demonstrated *between the brackets.
In Example 3: Of the list, 0 is the min.

max()

A function to find the maximum element of the sequence.

- Referred to as a method and not a function
- Searches the sequence from the beginning, in order to find the first element of the value specified in its argument.
- The method returns the index of the first occurrence of the argument

```
print("aAbByYzZaA".index("b"))
```

outputs: 2, as “b” is the 3rd element starting from position 0.

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

```
print(list("abcabc"))
```
Outputs: ['a', 'b', 'c', 'a', 'b', 'c']

count()
A function that counts all occurrences of the element inside the sequence.
- The absence of such elements doesn't cause any problems.

2.3 - String Methods

Summary for reference, followed by Standard Python string methods:

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?

The capitalize() method
This function creates a new string filled with characters taken from the source string, and attempts to do the following:
- **If the first character inside the string is a letter** (note: the first character is an element with an index equal to 0, not just the first visible character), **it will be converted to upper-case**
- All remaining letters from the string will be converted to lower-case.

The center() method
Makes a copy of the original string, trying to center it inside a field of a specified width.

- The centering is actually done by adding some spaces before and after the string.

The endswith() method
checks **if the given string ends with the specified argument and returns True** or **False**, depending on the check result.

The find() method
It looks for a substring and returns the index of the 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.

```
t = 'theta'
print(t.find('eta'))
print(t.find('ha'))
```

Outputs: 2, as "eta" begins at the 3rd position.

```
the_text = """A variation of the ordinary lorem ipsum
text has been used in typesetting since the 1960s
… text removed for notes
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)
```

The isalnum() method
Checks whether the string contains **only letters and digits**.

- Returns **True** if every character is **alphanumeric**.
  - ```
    print('lambda30'.isalnum())
    ```
- Returns **False** if the string contains **spaces, symbols, or is empty**.
  - ```
    print('lambda_30'.isalnum())
    ```

The isalpha() method
Checks whether the string contains **letters only**.

- **Digits, spaces, or symbols** cause it to return **False**.
- An **empty string** also returns **False**.

The isdigit() method
Checks whether the string contains **digits only**.

- Any non-digit character (**letters, spaces, symbols**) causes **False**.
- Commonly used for validating numeric input.

The islower() method
Checks whether **all letters** in the string are **lowercase**.
Non-letter characters are ignored.
Returns **False** **if any uppercase letter exists**.

The isspace() method
Checks whether the string contains **only whitespace characters**.

- Includes spaces, tabs, and newline characters.
- Returns **False** if ***any*** **non-whitespace character appears**.

The isupper() method
Checks whether **all letters** in the string are uppercase.
- Non-letter characters are ignored.
- Returns `False` if **any lowercase letter exists**.

The join() method
Joins a list (or tuple) of strings into **one string**.
- The string calling `join()` is used as the **separator**.
- **All elements must be strings** or a `TypeError` occurs.

The lower() method
Creates a copy of the string with **all letters converted to lowercase**.
- The original string remains unchanged.
- Takes no parameters.

The lstrip() method
Removes **leading whitespace characters** from the string.
- Returns a new string; the original is unchanged.
- An optional argument allows removing specific leading characters.

The replace() method
Replaces all occurrences of a substring with another substring.
- Returns a new string; the original remains unchanged.
```
print("www.cisco.com".lstrip("w."))
```
- 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
Searches for a substring starting from the **end** of the string.
- Returns the index of the first match from the right.
- Returns `-1` if the substring **is not found**.

The rstrip() method
Removes **trailing whitespace characters** from the string.
- Works the same as `lstrip()` but from the right side.
- An optional argument allows removing specific trailing characters.

The split() method
Splits the string into a **list of substrings**.
- By default, splits on whitespace.
```
print("phi        chi\npsi".split())
```
- The reverse operation is performed using `join()`.

The startswith() method
Checks whether the string **starts with** a specified substring.
- Returns `True` or `False`.
- **Case-sensitive**.

The strip() method
Removes both **leading and trailing whitespace**.
- Equivalent to applying `lstrip()` and `rstrip()` together.
- Returns a new string.

The swapcase() method
Swaps the case of every letter in the string.
- Lowercase becomes uppercase and vice versa.
- Non-letter characters remain unchanged.

The title() method
Capitalizes the **first letter of each word**.
- All remaining letters are converted to lowercase.
- Commonly used for formatting titles and headings.

The upper() method
Creates a copy of the string with **all letters converted to uppercase**.
- The original string remains unchanged.
- Takes no parameters.

2.4 - String in Action

1. Strings can be compared to other 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**.
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**
     - **In situ** means an operation is performed directly on the original object, without creating a new copy.
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).

– – Continued – –

2.5 - 4 simple programs

1. Strings are key tools in modern data processing, as most useful data are actually strings. For example, using a web search engine (which seems quite trivial these days) utilizes extremely complex string processing, involving unimaginable amounts of data.

2. Comparing strings in a strict way (as Python does) can be very unsatisfactory when it comes to advanced searches (e.g. during extensive database queries). Responding to this demand, a number of fuzzy string comparison algorithms has been created and implemented. These algorithms are able to find strings which aren't equal in the Python sense, but are similar.
One such concept is the Hamming distance, which is used to determine the similarity of two strings. If this problem interests you, you can find out more about it here: https://en.wikipedia.org/wiki/Hamming_distance.
Another solution of the same kind, but based on a different assumption, is the Levenshtein distance described here: https://en.wikipedia.org/wiki/Levenshtein_distance.

3. Another way of comparing strings is finding their acoustic similarity, which means a process leading to determine if two strings sound similar (like "raise" and "race"). Such a similarity has to be established for every language (or even dialect) separately.
An algorithm used to perform such a comparison for the English language is called Soundex and was invented – you won't believe – in 1918. You can find out more about it here: https://en.wikipedia.org/wiki/Soundex.

4. Due to limited native float and integer data precision, it's sometimes reasonable to store and process huge numeric values as strings. This is the technique Python uses when you force it to operate on an integer number consisting of a very large number of digits.

The Caesar Cipher: Encrypting a message

```
text = input("Enter your message: ")
cipher = ''
for char in text:
    if not char.isalpha():
        continue
    char = char.upper()
    code = ord(char) + 1
    if code > ord('Z'):
        code = ord('A')
    cipher += chr(code)

print(cipher)
```

We've written it using the following assumptions:

- it accepts Latin letters only (note: the Romans didn't use whitespaces or digits)
- all letters of the message are in upper case (note: the Romans knew only capitals)

Let's trace the code:

- line 02: ask the user to enter the open (unencrypted), one-line message;
- line 03: prepare a string for an encrypted message (empty for now)
- line 04: start the iteration through the message;
- line 05: if the current character is not alphabetic...
- line 06: ...ignore it;
- line 07: convert the letter to upper-case (it's preferable to do it blindly, rather than check whether it's needed or not)
- line 08: get the code of the letter and increment it by one;

- line 09: if the resulting code has "left" the Latin alphabet (if it's greater than the Z code)...
- line 10: ...change it to the A code;
- line 11: append the received character to the end of the encrypted message;
- line 13: print the cipher.

Caesar Cipher Decryption:

```
cipher = input('Enter your cryptogram: ')
text = ''
for char in cipher:
    if not char.isalpha():
        continue
    char = char.upper()
    code = ord(char) - 1
    if code < ord('A'):
        code = ord('Z')
    text += chr(code)

print(text)
```

The Numbers processor

```
# Numbers Processor.
from curses.ascii import isalnum

line = input("Enter a line of numbers - separate them with spaces: ")
strings = line.split()
total = 0

if isalnum(line) == False:
    print(f"Sorry, but \"{line}\" is not a number.")
else:
    try:
        for substr in strings:
            total += float(substr)
            print("The total is:", total)
    except:
        print(substr, "is not a number.")
```

Using list comprehension may make the code slimmer. You can do that if you want.

Let's present our version:

- line 03: ask the user to enter a line filled with any number of numbers (the numbers can be floats);
- line 04: split the line receiving a list of substrings;
- line 05: initiate the total sum to zero;
- line 06: as the string-float conversion may raise an exception, it's best to continue by using the try-except block;
- line 07: iterate through the list...
- line 08: ...and try to convert all its elements into float numbers; if it works, increase the sum;

- line 09: everything is good so far, so print the sum;
- line 10: the program ends here in the case of an error;
- line 11: print a diagnostic message showing the user the reason for the failure.

The code has one important weakness – it displays a bogus result when the user enters an empty line. Can you fix it? - I did, by creating an if-else block out of the original “try” block. Originally, the try block was independent.

– – Continued – –

The IBAN Validator

```
iban = input("Enter IBAN, please: ")
iban = iban.replace(' ', '')

if not iban.isalnum():
   print("You have entered invalid characters.")
elif len(iban) < 15:
   print("IBAN entered is too short.")
elif len(iban) > 31:
   print("IBAN entered is too long.")
else:
   iban = (iban[4:] + iban[0:4]).upper()
   iban2 = ''
   for ch in iban:
       if ch.isdigit():
           iban2 += ch
       else:
           iban2 += str(10 + ord(ch) - ord('A'))
   iban = int(iban2)
   if iban % 97 == 1:
       print("IBAN entered is valid.")
   else:
       print("IBAN entered is invalid.")
```

The standard named IBAN (International Bank Account Number) provides a simple and fairly reliable method for validating account numbers against simple typos that can occur during rewriting of the number, for example, from paper documents, like invoices or bills, into computers.

You can find more details here: https://en.wikipedia.org/wiki/International_Bank_Account_Number.

An IBAN-compliant account number consists of:

- a **two-letter country code taken from the ISO 3166-1 standard** (e.g., FR for France, GB for Great Britain, DE for Germany, and so on)
- **two check digits used to perform the validity checks** – fast and simple, but not fully reliable, tests, showing whether a number is invalid (distorted by a typo) or seems to be good;
- **the actual account number** (up to **30** alphanumeric characters – the length of that part depends on the country)

The standard says that validation requires the following steps (according to Wikipedia):

- (step 1) **Check that the total IBAN length is correct as per the country** (this program won't do that, but you can modify the code to meet this requirement if you wish; note: you have to teach the code all the lengths used in Europe)
- (step 2) Move the four initial characters to the end of the string (i.e., the country code and the check digits)
- (step 3) Replace each letter in the string with two digits, thereby expanding the string, where A = 10, B = 11 ... Z = 35;
- (step 4) Interpret the string as a decimal integer and compute the remainder of that number by modulo-dividing it by 97; If the remainder is 1, the check digit test is passed and the IBAN might be valid.

2.6 - Errors

Exceptions

The Try-Except method:

```
Try: first, you have to try to do something;
   :
   :
Except: next, you have to check whether everything went well.
   :
   :
```

- the **try** keyword **begins a block of the code** which may or may not be performing correctly;
- next, Python tries to perform the risky action; if it fails, an exception is raised and Python starts to look for a solution;
- if nothing is wrong with the execution and all instructions are performed successfully, the execution jumps to the point after the last line of the except: block, and the block's execution is considered complete;

- the **except** keyword **starts a piece of code which will be executed if anything inside the try block goes wrong** – if an exception is raised inside a previous try block, it will fail here, so the code located after the except keyword should provide an adequate reaction to the raised exception;
- returning to the previous nesting level ends the try-except section.
- **If anything goes wrong inside the try:** and **except:** block, the **execution immediately jumps out of the block and into the first instruction located after the except: keyword**; this means that some of the instructions from the block may be silently omitted.

```
try:
   print("1")
   x = 1 / 0
   print("2")
except:
   print("Oh dear, something went wrong...")

print("3")
```

This approach has **one important disadvantage** – if there is **a possibility that more than one exception may skip into an except: branch**, you may have trouble figuring out what actually happened.

As a reminder, here are some common exception handles to remember:

ZeroDivisionError
Raised when division is attempted with a divisor of zero (or effectively zero). Triggered by `/`, `//`, and `%`. Occurs at runtime and indicates a math or logic error.

ValueError
Raised when a value is of the correct type but is invalid for the operation. Common with `int()`, `float()`, and input parsing. Indicates bad or unacceptable data.

TypeError
Raised when an operation is applied to an incompatible data type. Often caused by mixing types or using the wrong type for indexing or arithmetic.

AttributeError
Raised when attempting to access an attribute or method that does not exist on an object. Commonly caused by typos or incorrect assumptions about an object's interface.

SyntaxError
Raised when Python code violates grammatical rules. Detected before execution completes and prevents the program from running.

IndexError
Raised when accessing a sequence index that does not exist. Common with lists, tuples, and strings when indexes are out of range.

KeyError
Raised when attempting to access a dictionary key that does not exist. Often avoided by using `.get()` or checking keys first.

NameError
Raised when a variable or function name has not been defined. Commonly caused by typos or scope issues.

ImportError / ModuleNotFoundError
Raised when an import fails because the module does not exist or is not available in the environment. Common in misconfigured systems.

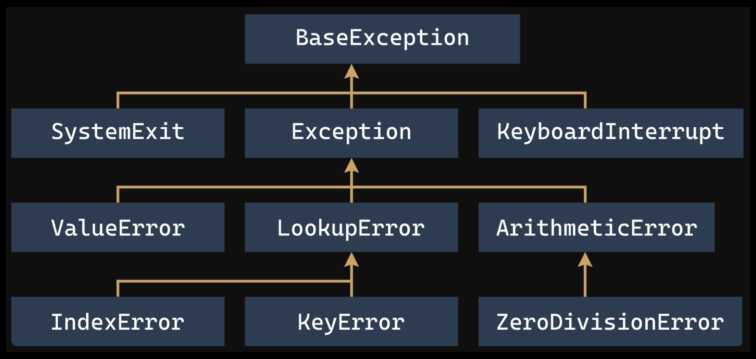
FileNotFoundError
Raised when attempting to open a file that does not exist. Common in scripts that rely on external files or paths.

PermissionError
Raised when access to a file, directory, or resource is denied due to insufficient permissions. Common on multi-user systems and relevant to security contexts.

2.7 - The Anatomy of Exceptions

Exceptions

A small section of the complete exception tree



Python 3 defines 63 built-in exceptions, and all of them form a tree-shaped hierarchy, although the tree is a bit weird as its root is located on top.

- **ZeroDivisionError** is a special case of a more general exception class named ArithmeticError;
- **ArithmeticError** is a special case of a more general exception class named just Exception;
- **Exception** is a special case of a more general class named BaseException;

- the order of the branches matters!
- don't put more general exceptions before more concrete ones;
- this will make the latter one unreachable and useless;
- moreover, it will make your code messy and inconsistent;
- Python won't generate any error messages regarding this issue.

A complete map of all exceptions is on the next page:

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

```
try:
  :
except (exc1, exc2):
  :
```

You simply have to put all the engaged exception names into a comma-separated list and not forget the parentheses.

- Personal note: Add a counter to each exception to track how many exceptions were triggered.

Note: **the exception raised can cross function and module boundaries**, and travel through the invocation chain looking for a matching except clause able to handle it.

If there is no such clause, **the exception remains unhandled**, and Python solves the problem in its standard way – by **terminating your code** and **emitting a diagnostic message**.

– – Continued – –

The raise instruction

```
def bad_fun(n):
    raise ZeroDivisionError
```

The instruction enables you to:

- **simulate raising actual exceptions** (e.g., to test your handling strategy)
- partially **handle an exception** and make another part of the code responsible for completing the handling (separation of concerns).

In this way, you can **test your exception handling routine** without forcing the code to do stupid things.

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**.

The assert instruction

```
import math

x = float(input("Enter a number: "))
assert x >= 0.0

x = math.sqrt(x)

print(x)
```

- It evaluates the expression;
- **if the expression evaluates to True**, or a **non-zero numerical value**, or a **non-empty string**, or any other value different than **None**, **it won't do anything else**;
- otherwise, it automatically and immediately raises an exception named AssertionError (in this case, we say that the assertion has failed)

How can it be used?

- you may want to put it into your code where you want to be **absolutely safe from evidently wrong data**, and where you aren't absolutely sure that the data has been carefully examined before (e.g., inside a function used by someone else)
- raising an **AssertionError** exception **secures your code from producing invalid results**, and clearly shows the nature of the failure;
- **assertions don't supersede exceptions or validate the data** – they are their supplements.

The program above runs flawlessly if you enter a valid numerical value greater than or equal to zero; otherwise, it stops and emits the following message:

```
Traceback (most recent call last):
File ".main.py", line 4, in <module>
assert x >= 0.0
AssertionError
```

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

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:

3. The Python statement raise ExceptionName can raise an exception on demand. The same statement, but lacking ExceptionName, can be used inside the except 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.

– – Continued – –

2.8 - Useful Exceptions

Built-in Exceptions

Along with the course material, a DIY sheet of the exceptions sits at the end of this chapter.

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

```
from math import tan, radians
angle = int(input('Enter integral angle in degrees: '))

# We must be sure that angle != 90 + k * 180
assert angle % 180 != 90
print(tan(radians(angle)))
```

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)

```
# The code shows an extravagant way
# of leaving the loop.

the_list = [1, 2, 3, 4, 5]
ix = 0
do_it = True

while do_it:
    try:
        print(the_list[ix])
        ix += 1
    except IndexError:
        do_it = False

print('Done')
```

– – Continued – –

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.

```
# This code cannot be terminated
# by pressing Ctrl-C.

from time import sleep

seconds = 0

while True:
    try:
        print(seconds)
        seconds += 1
        sleep(1)
    except KeyboardInterrupt:
        print("Don't do that!")
```

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.

```
# This code causes the MemoryError exception.
# Warning: executing this code may affect your OS.
# Don't run it in production environments!

string = 'x'
try:
    while True:
        string = string + string
        print(len(string))
except MemoryError:
    print('This is not funny!')
```

OverflowError

Location: BaseException ← Exception ← ArithmeticError ← OverflowError
Description: a concrete exception raised when an operation produces a number too big to be successfully stored

```
# The code prints subsequent
# values of exp(k), k = 1, 2, 4, 8, 16, ...

from math import exp

ex = 1

try:
    while True:
        print(exp(ex))
        ex *= 2
except OverflowError:
    print('The number is too big.')
```

ImportError
Location: BaseException ← Exception ← StandardError ← ImportError
Description: a concrete exception raised when an import operation fails

```
>
# One of these imports will fail - which one?

try:
    import math
    import time
    import abracadabra:

except:
    print('One of your imports has failed.')
```

KeyError
Location: BaseException ← Exception ← LookupError ← KeyError
Description: a concrete exception raised when you try to access a non-existent element in a collection (e.g., a dictionary's element)

```
# How to abuse the dictionary
# and how to deal with it?

dictionary = {'a': 'b', 'b': 'c', 'c': 'd'}
ch = 'a'

try:
    while True:
        ch = dictionary[ch]
        print(ch)
except KeyError:
    print('No such key:', ch)
```

– – All Built-In Exceptions Mapped on Next Page – –


BaseException
Base class for all exceptions
Exception
Base class for moot exceptions
SystemExit
KeyboardInterrupt
ValueError
Invalid value
LookupError
ArithmeticError
Base class for artinmetic errors
AssertionError
Base class for tebantion marremaitg
TypeError
Invalid operation
IndexError
Invalid index
KeyError
Invalid key
ArithmeticError
Base class for artinmetic errors
AssertionError
Assertion failed
StopAsyncIteration
Deprecated, use AsyncGeneratorExit instead
StopIteration
Deprecated, end of iteration
Exception
Base class for moot exceptions
OSeror
Base class for I/O errors
RuntimeError
Genessl runtime error
NameError
Invalid name
BlockingIOError
Operation Noalh Blacls
NotImplementedError
Unmphenented function
NameError
Invalid name
EncodingError
Base class for excoding errors
MemoryError
Menory allocation falled
EncodingError
Base class for encoding errort
UnicodeError
Error encoding Idcooling Unicode
RuntimeError
General runtime error
NameError
Invalid name
BrokenPipeError
NotImplementedError
Unmplemented function
EncodingError
FileExistsError
File/directory exists
ADLeor
IsADirectoryError
Module not found
TypeError
ModuleNotFoundError
Process lookup failed
ProcessLookupError
File/directory not found
MemoryError
Memory allocation failed
FileExistsError
FileExistsError
PermissionError
Timeout expired
StopAsyncIteration
Deprecated, use AsyncGeneratorExit instead
Stopiteration
Deprecated, end of iteration

Module 3 - Object-Oriented Programming

3.1 - The foundations of OOP

Basic Concepts of the Object-Oriented Approach

The procedural style of programming was the dominant approach to software development for decades of IT, and it is still in use today. Moreover, it isn't going to disappear in the future, as it works very well for specific types of projects (generally, not very complex ones and not large ones, but there are lots of exceptions to that rule).

The object approach is quite young (much younger than the procedural approach) and is particularly useful when applied to big and complex projects carried out by large teams consisting of many developers.

This kind of understanding of a project's structure makes many important tasks easier, for example, dividing the project into small independent parts, and developing different project elements independently.

Python is a universal tool for both object and procedural programming. It may be successfully utilized in both spheres.

Procedural vs. the object-oriented approach

In the **procedural approach**, it's possible to distinguish two different and completely separate worlds: the world of data, and the world of code. **The world of data is populated with variables of different kinds**, while **the world of code is inhabited by code grouped into modules and functions**.

Functions are able to use data, but not vice versa. Furthermore, functions are able to abuse data, in other words, to use the value in an unauthorized manner (e.g., when the sine function gets a bank account balance as a parameter).

We said in the past that data cannot use functions. But is this entirely true? Are there some special kinds of data that can use functions?

Yes, there are – the ones named **methods**. These are **functions which are invoked from within the data**, not beside them. If you can see this distinction, you've taken the first step into object programming.

The object approach suggests a completely different way of thinking. The data and the code are enclosed together in the same world, divided into classes.

Every **class** is like a recipe which can be used when you want to create a useful object (this is where the name of the approach comes from). You may produce as many objects as you need to solve your problem.

Every **object** has a set of **traits** (they are called **properties or attributes – we'll use both words synonymously**) and is **able to perform a set of activities** (which are called methods).

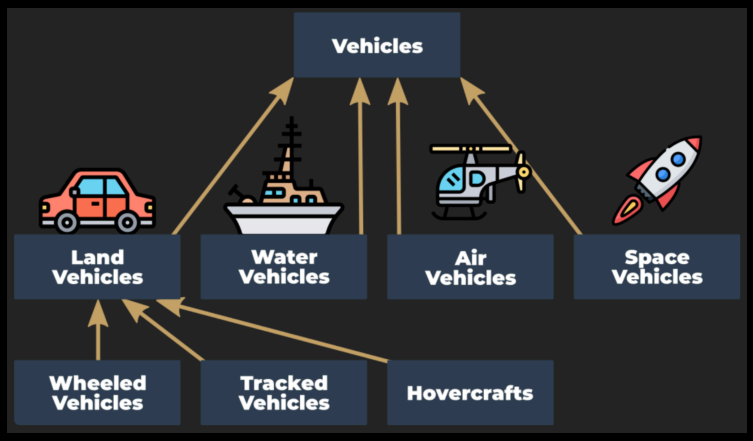
The recipes may be modified if they are inadequate for specific purposes and, in effect, new classes may be created. These new classes inherit properties and methods from the originals, and usually add some new ones, creating new, more specific tools.

Objects are incarnations of **ideas expressed in classes**, like a cheesecake on your plate is an incarnation of the idea expressed in a recipe printed in an old cookbook.

The objects interact with each other, exchanging data or activating their methods. A properly constructed class (and thus, its objects) are able to protect the sensible data and hide it from unauthorized modifications.

There is no clear border between data and code: they live as one in **objects**.

Class hierarchies



A class in Python is better defined as a category.

The hierarchy grows from top to bottom, like tree roots, not branches.

The top (the **superclass**) ↘
its descendants located below (the **subclasses**).

What is an object?

A **class** is a set of objects.

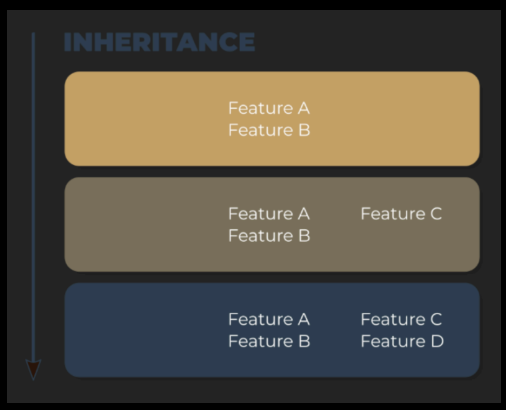
An **object** is a being belonging to a **class**.

An object is an incarnation of the requirements, traits, and qualities assigned to a specific class.

This may mean that an object belonging to a specific class belongs to all the superclasses at the same time. It may also mean that any object belonging to a superclass may not belong to any of its subclasses.

Each **subclass is more specialized** (or more concrete) **than its superclass**.
Conversely, **each superclass is more general** (more abstract) **than any of its subclasses**.

Inheritance



Any object bound to a specific level of a class hierarchy inherits all the traits (as well as the requirements and qualities) defined inside any of the superclasses.

- The object's home class may define new traits (as well as requirements and qualities) which will be inherited by any of its subclasses.

What does an object have?

The object programming convention assumes that **every existing object may be equipped with three groups of attributes**:

- an object **has a name that uniquely identifies it within its home namespace** (although there may be some anonymous objects, too)
- an object **has a set of individual properties which make it original, unique, or outstanding** (although it's possible that some objects may have no properties at all)
- an object **has a set of abilities to perform specific activities**, able to change the object itself, or some of the other objects.

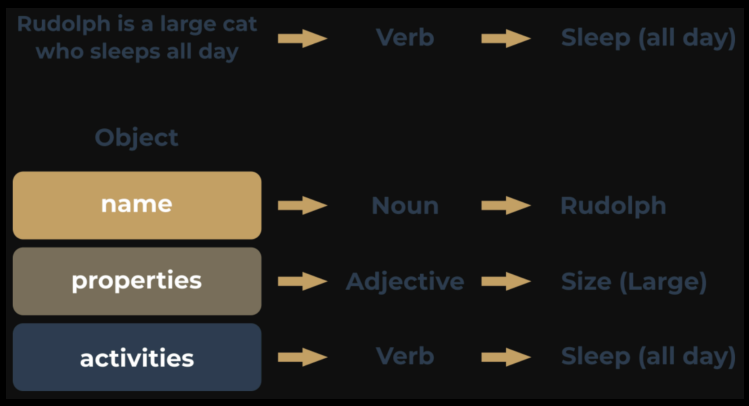
There is a hint (although this doesn't always work) which can help you identify any of the three spheres above.

Whenever you describe an object and you use:

- a noun – you probably define the object's name;
- an adjective – you probably define the object's property;
- a verb – you probably define the object's activity.

Rudolph is a large cat who sleeps all day.

Object name = Rudolph
Home class = Cat
Property = Size (large)
Activity = Sleep (all day)



Your first class

Object programming is **the art of defining and expanding classes**.

- A **class** is a model of a very specific part of reality, reflecting properties and activities found in the real world.

There's no obstacle to defining new, more precise subclasses. They'll inherit everything from their superclass, so the work that went into its creation isn't wasted.

The new class may add new properties and new activities, and therefore may be more useful in specific applications. Obviously, it may be used as a superclass for any number of newly created subclasses.

The process doesn't need to have an end. You can create as many classes as you need.

The class you define has nothing to do with the object: the existence of a class does not mean that any of the compatible objects will automatically be created. The class itself isn't able to create an object – you have to create it yourself, and Python allows you to do this.

```
class TheSimplestClass:
    pass
```

The definition begins with the keyword **class**. The keyword is followed by an identifier which will name the class (note: don't confuse it with the object's name – these are two different things).

Next, you add a colon (:), as **classes**, like functions, **form their own nested block**.

- The content inside the block defines all the class's properties and activities.

The **pass** keyword **fills the class with nothing**. It doesn't contain any methods or properties.

Your first object

The act of creating an object of the selected class is also called an **instantiation**
(as the object becomes an instance of the class).

To store an object in this new class:

```
my_first_object = TheSimplestClass()
```

- Note: the class name tries to pretend that it's a function

– – Continued – –

3.2 - A short journey from procedural to object approach

What is a Stack?

A stack is **a structure developed to store data in a very specific way**.

An alternative name for stack in IT terminology is **LIFO**: Last In - First Out

A stack is an object with **two elementary operations**

- **push** (when a new element is put on the top)
- **pop** (when an existing element is taken away from the top).

The stack - The procedural approach

First, you have to decide how to store the values which will arrive onto the stack. We suggest using the simplest of methods, and employing a list for this job. Let's assume that the size of the stack is not limited in any way. Let's also assume that the last element of the list stores the top element.

```
stack = []
```

We're ready **to define a function that puts a value onto the stack**.

Here are the presuppositions for it:

- the name for the function is **push**;
- the function gets one parameter (this is the value to be put onto the stack)
- the function returns nothing;
- the function **appends the parameter's value to the end of the stack**;

```
def push(val):
    stack.append(val)
```

Now it's time for a function to take a value off the stack. This is how you can do it:

- the name of the function is **pop**;
- the function doesn't get any parameters;
- the function returns the value taken from the stack
- the function **reads the value from the top of the stack and removes it**.

```
def pop():
    val = stack[-1]
    del stack[-1]
    return val
```

The complete program pushes three numbers onto the stack, pulls them off, and prints their values on the screen.

– – Continued – –

The Stack - Procedural vs Object Oriented Approaches

Disadvantages to the procedural approach:

- **the essential variable** (the stack list) **is highly vulnerable**; anyone can modify it in an uncontrollable way, destroying the stack, in effect; this doesn't mean that it's been done maliciously – on the contrary, it may happen as a result of carelessness, e.g., when somebody confuses variable names
- the **functioning** of the stack **will be completely disorganized**
- **it may also happen that one day you need more than one stack**; you'll have to create another list for the stack's storage, and probably other **push** and **pop** functions too
- it may also happen that you need not only **push** and **pop** functions, but also some other conveniences; you could certainly implement them, but try to imagine what would happen if you had dozens of separately implemented stacks.

The object-oriented approach delivers solutions for each of these problems:

- the ability to hide (protect) selected values against unauthorized access is called **encapsulation**; the encapsulated values can be neither accessed nor modified if you want to use them exclusively
- when you have a class implementing all the needed stack behaviors, you can produce as many stacks as you want; you needn't copy or replicate any part of the code
- the ability to enrich the stack with new functions comes from inheritance; **you can create a new class** (a subclass) **which inherits all the existing traits from the superclass, and adds some new ones.**

The stack - The object approach

```
class Stack:
```

Now, we expect two things from it:

- we want the class to have **one property as the stack's storage** – we have to **"install" a list inside each object of the class** (note: each object has to have its own list – the list mustn't be shared among different stacks)
- then, we want **the list to be hidden from the class users' sight**.
    - (The properties have to be added to the class manually)

To equip the class with a specific function – its specificity is dual:

- it has to be named in a strict way;
- it is invoked implicitly, when the new object is created.

Such a function is called a **constructor**, as its general purpose is to construct a new object. The constructor should know everything about the object's structure, and must perform all the needed initializations.

```
class Stack:  # Defining the Stack class.
   def __init__(self):  # Defining the constructor function.
       print("Hi!")


stack_object = Stack()  # Instantiating the object.
```

- the constructor's name is always **__init__**
- it **has to have at least one parameter** (we'll discuss this later); the parameter is used to represent the newly created object – you can use the parameter to manipulate the object, and to enrich it with the needed properties; you'll make use of this soon;
- Note: **the obligatory parameter is usually named self** – it's only a convention, but **you should follow it** – it simplifies the process of reading and understanding your code.

Any change you make inside the constructor that modifies the state of the **self parameter** will be reflected in the newly created object.

- This means you can add any property to the object and the property will remain there until the object finishes its life or the property is explicitly removed.

Now let's add just one property to the new object – a list for a stack. We'll name it **stack_list**.

```
class Stack:
   def __init__(self):
       self.stack_list = []


stack_object = Stack()
print(len(stack_object.stack_list))
```

- We've used dot notation, just like when invoking methods; **this is the general convention for accessing an object's properties**:
  - you need to name the object,
  - put a dot (**.**) after it,
  - and specify the desired property's name;
    - **don't use parentheses!** You don't want to invoke a method – you want to access a property
- if you set a property's value for the very first time (like in the constructor), you are creating it; from that moment on, the object has got the property and is ready to use its value;
- we've done something more in the code – we've tried to access the **stack_list** property from outside the class immediately after the object has been created; we want to check the current length of the stack – have we succeeded?

In order to keep stack_list hidden from the out world, add 2 underscores (__) :

```
print(len(stack_object.__stack_list))
```

When any class component has **a name starting with two underscores (__), it becomes private** – this means that it can be accessed only from within the class.

The object approach: a stack from scratch

Now it's time for the two functions (methods) implementing the **push** and **pop** operations. Python assumes that a function of this kind (a class activity) should be immersed inside the class body – just like a constructor.

We want to invoke these functions to **push** and **pop** values. This means that they should both be accessible to every class's user (in contrast to the previously constructed list, which is hidden from the ordinary class's users).

Such a component is called **public**, so you **can't begin its name with two (or more) underscores**. There is one more requirement – **the name must have no more than one trailing underscore**. As no trailing underscores at all fully meets the requirements, you can assume that the name is acceptable.

– – Continued – –

```
class Stack:
   def __init__(self):
       self.__stack_list = []


   def push(self, val):
       self.__stack_list.append(val)


   def pop(self):
       val = self.__stack_list[-1]
       del self.__stack_list[-1]
       return val


stack_object = Stack()

stack_object.push(3)
stack_object.push(2)
stack_object.push(1)

print(stack_object.pop())
print(stack_object.pop())
print(stack_object.pop())
```

This is what we expect from the **push** functionality:

- to add the value to the **__sum** variable;
- to **push the value onto the stack**.

```
def push(self, val):
    self.__sum += val
    Stack.push(self, val)
```

- we have to specify the superclass's name; this is necessary in order to clearly indicate the class containing the method, to avoid confusing it with any other function of the same name;
- we have to specify the target object and to pass it as the first argument (it's not implicitly added to the invocation in this context.)

The new **pop** function:

```
def pop(self):
    val = Stack.pop(self)
    self.__sum -= val
    return val
```

We've defined the **__sum** variable, but **we haven't provided a method to get its value**. It seems to be hidden.

To reveal it and do it in a way that still protects it from modifications:

```
def get_sum(self):
    return self.__sum
```

– – Final Code on Next Page – –

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val


class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0

    def get_sum(self):
        return self.__sum

    def push(self, val):
        self.__sum += val
        Stack.push(self, val)

    def pop(self):
        val = Stack.pop(self)
        self.__sum -= val
        return val


stack_object = AddingStack()

for i in range(5):
    stack_object.push(i)
print(stack_object.get_sum())

for i in range(5):
    print(stack_object.pop())
```

As you can see,

1. We add five subsequent values onto the stack
2. Print their sum
3. Take them all off the stack.

Summary

1. A stack is an object designed to store data using the **LIFO** model. The stack usually performs at least two operations, named **push()** and **pop()**.

2. Implementing the stack in a procedural model raises several problems which can be solved by the techniques offered by **OOP** (Object Oriented Programming).

3. A class method is actually a function declared inside the class and able to access all the class's components.

4. The part of the Python class responsible for creating new objects is called the **constructor**, and it's implemented as a method of the name **__init__**.

5. Each class method declaration must contain at least one parameter (always the first one) usually referred to as **self**, and is used by the objects to identify themselves.

6. If we want to hide any of a class's components from the outside world, we should start its name with __. Such components are called **private**.

3.3 - OOP: Properties

3 Really great YouTube examples of the topics in this section:

1. **Stack implementation using Lists (Push/Pop/Peek)**
   a. https://www.youtube.com/watch?v=jwcnSv9tGdY
2. **Name Mangling Python (double-underscore "privacy")**
   a. https://www.youtube.com/watch?v=6cvFzLB6hbA
3. **Intro to OOP: Public / Private / Protected attributes**
   a. https://www.youtube.com/watch?v=foPVgCjJIhI YouTube

Instance variables

A class can store data in more than one way.

The first and most common way is through **instance variables**.

- An instance variable **exists only when it is created and attached to a specific object**.
  - Most of the time, this happens inside the constructor (**__init__**), but Python allows instance **variables to be added or removed at any point during the object's lifetime**.
- different objects of the **same class may possess different sets of properties**.
- there must be a way to **safely check if a specific object owns the property you want to utilize** (unless you want to provoke an exception – it's always worth considering)
- **each object carries its own set of properties** – they don't interfere with one another in any way.

```
class Vehicle:
    def __init__(self, horsepower=150):
        self.horsepower = horsepower    # instance variable created at init

    def set_transmission(self, transmission):
        self.transmission = transmission    # instance variable added later


car_1 = Vehicle()             # default horsepower
car_2 = Vehicle(300)          # custom horsepower

car_2.set_transmission("Manual")

car_3 = Vehicle(450)
car_3.__turbo = True            # attribute added dynamically


print(car_1.__dict__)
print(car_2.__dict__)
print(car_3.__dict__)
```

Take a look at the last three lines of the code.

Python objects, when created, **are gifted with a small set of predefined properties and methods**.

Each object has got them, whether you want them or not. One of them is a variable named **__dict__** (it's a dictionary).

The variable contains the names and values of all the properties (variables) the object is currently carrying. Let's make use of it to safely present an object's contents.

Let's dive into the code now:

- the class named **Vehicle** has a constructor, which **unconditionally creates an instance variable named horsepower**, and sets it with the value passed through the first argument (from the class user's perspective) or the second argument (from the constructor's perspective); note the default value of the parameter – any trick you can do with a regular function parameter can be applied to methods, too;
- the class also has a method which creates another instance variable, named **transmission**;
- we've created three objects of the class **Vehicle**, but all these instances differ:
    - **car_1** only has the property named horsepower;
    - **car_2** has two properties: horsepower and transmission;
    - **car_3** has been enriched with a property named **turbo** just on the fly, outside the class's code - this is possible and fully permissible.

The final output:
{'horsepower': 150}
{'horsepower': 300, 'transmission': 'Manual'}
{'horsepower': 450, 'turbo': True}

When Python sees that you want to add an instance variable to an object and you're going to do it inside any of the object's methods, **it mangles the operation** in the following way:

- it puts a class name before your name;
- it puts an additional underscore at the beginning.
- Name-mangling **only happens for attributes that begin with __** inside a class.
    - **Single-underscore or normal names are left alone.**

The name is now fully accessible from outside the class.

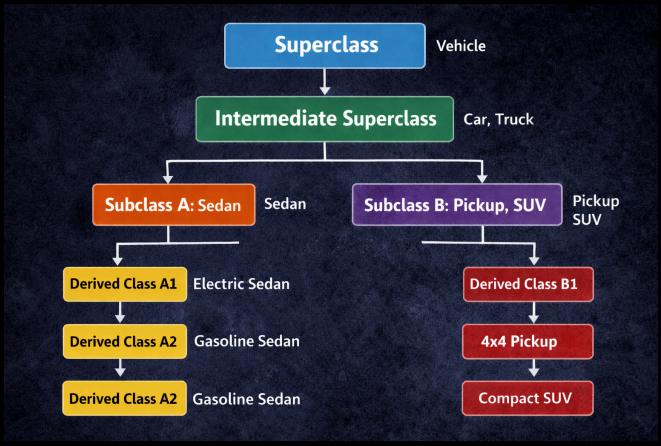
This is the end of this chapter, as oddly as that felt...

– – Continued – –

Class Variables

A property which exists in just one copy and is stored outside any object.

- Class variables are created differently to their instance siblings.



Original Lecture Code:

```
class ExampleClass:
    counter = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.counter += 1


example_object_1 = ExampleClass()
example_object_2 = ExampleClass(2)
example_object_3 = ExampleClass(4)

print(example_object_1.__dict__, example_object_1.counter)
print(example_object_2.__dict__, example_object_2.counter)
print(example_object_3.__dict__, example_object_3.counter)
```

ChatGPT's mod:

```
class Vehicle:
   total_vehicles = 0   # class variable (shared across all vehicles)

   def __init__(self, horsepower=150):
       self.__engine_power = horsepower   # instance variable (name-mangled)
       Vehicle.total_vehicles += 1

car_1 = Vehicle()
car_2 = Vehicle(300)
car_3 = Vehicle(450)

print(car_1.__dict__, car_1.total_vehicles)
print(car_2.__dict__, car_2.total_vehicles)
print(car_3.__dict__, car_3.total_vehicles)
```

- there is an assignment in the first line of the class definition – it sets the variable named **total_vehicles** to **0**; initializing the variable inside the class but outside any of its methods makes the variable a **class variable**;
- accessing such a variable looks the same as accessing any instance attribute – you can see it in the constructor body; as you can see, the constructor increments the variable by one; in effect, the variable counts all the created objects.

The output:
{'_Vehicle__engine_power': 150} 3
{'_Vehicle__engine_power': 300} 3
{'_Vehicle__engine_power': 450} 3

Two important conclusions come from the example:

- **class variables aren't shown in an object's __dict__** (this is natural as class variables aren't parts of an object) but you can always try to look into the variable of the same name, but at the class level – we'll show you this very soon;
- **a class variable always presents the same value** in all class instances (objects)

Mangling a class variable's name has the same effects as those you're already familiar with.

```
class Vehicle:
    base_model_year = 2020    # class variable

    def __init__(self, year):
        Vehicle.base_model_year = year    # modifies the class variable


print(Vehicle.__dict__)
car = Vehicle(2024)

print(Vehicle.__dict__)
print(car.__dict__)
```

Checking an attribute's existence
Python's attitude to object instantiation raises one important issue – in contrast to other programming languages, **you may not expect that all objects of the same class have the same sets of properties**.

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1


example_object = ExampleClass(1)

print(example_object.a)
print(example_object.b)
```

The object created by the constructor can have only one of two possible attributes: **a** or **b**.

The result:

```
1
Traceback (most recent call last):
  File ".main.py", line 11, in
    print(example_object.b)
AttributeError: 'ExampleClass' object has no attribute 'b'
```

The try-except instruction gives you the chance to avoid issues with non-existent properties.

```
class ExampleClass:
   def __init__(self, val):
       if val % 2 != 0:
           self.a = 1
       else:
           self.b = 1


example_object = ExampleClass(1)
print(example_object.a)

try:
   print(example_object.b)
except AttributeError:
   pass
```

This action isn't very sophisticated. Essentially, we've just swept the issue under the carpet.

Python provides a function which is able to safely check if any object/class contains a specified property.

The function is named **hasattr**, and **expects two arguments to be passed to it**:

- the class or the object being checked;
- **the name of the property whose existence has to be reported** (note: it has to be a string containing the attribute name, not the name alone)

The function returns **True** or **False**.

```
if hasattr(example_object, 'b'):
   print(example_object.b)
```

The **hasattr()** function can operate on classes, too. You can use it to find out if a class variable is available.

- The function returns **True** if the specified class contains a given attribute, and **False** otherwise.

```
class ExampleClass:
   attr = 1


print(hasattr(ExampleClass, 'attr'))
print(hasattr(ExampleClass, 'prop'))
```

Summary

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.

```
class Vehicle:
   fleet_count = 0   # Class variable

   def __init__(self):
       self.mileage = 10000        # Instance variable
       self.__vin_code = "X123YZ"  # Private (name-mangled) instance variable


car = Vehicle()
car.color = "Red"    # Instance variable added after creation

print(car.__dict__)
```

The output:
{'mileage': 10000, '_Vehicle__vin_code': 'X123YZ', 'color': 'Red'}

3.4 - OOP Fundamentals: Inheritance

Watch this in full: https://www.youtube.com/watch?v=BJ-VvGyQxho

Methods in detail

Remembering that: a method is a function embedded inside a class.

- There is one fundamental requirement – **a method is obliged to have at least one parameter**

The first (or only) parameter is usually named self.

- Best practice is to follow this method – it's commonly used, and will confuse others if not followed.

The name self suggests the parameter's purpose – **it identifies the object for which the method is invoked**.

```
class Classy:
   def method(self):
       print("method")


obj = Classy()
obj.method()
```

Note the way we've created the object – we've **treated the class name like a function**, returning a newly instantiated object of the class.

If you want the method to accept parameters other than **self**, you should:

- place them after **self** in the method's definition;
- deliver them during invocation without specifying **self** (as previously)

```
class Classy:
    def method(self, par):
        print("method:", par)


obj = Classy()
obj.method(1)
obj.method(2)
obj.method(3)
```

Output:
method: 1
method: 2
method: 3

The **self** parameter is used to obtain access to the object's instance and class variables.

```
class Classy:
    varia = 2
    def method(self):
        print(self.varia, self.var)


obj = Classy()
obj.var = 3
obj.method()
```

Output: > 2 3

The self parameter is also used to invoke other object/class methods from inside the class.

a very simple constructor at work:

```
class Classy:
   def other(self):
       # A regular instance method
       # Can be called by any object of this class
       print("other")

   def method(self):
       # Another instance method
       # 'self' refers to the current object calling this method
       print("method")
       self.other()  # Call another method on the SAME object


# Create an instance of the Classy class
obj = Classy()

# Call the 'method' function on this object
# Python automatically passes 'obj' as 'self'
obj.method()
```

Output:

```
method
other
```

If you name a method like this: **__init__**, it won't be a regular method – it will be a **constructor**.
If a class has a constructor, it is invoked automatically and implicitly when the object of the class is instantiated.

The **constructor**:

- is obliged to have the **self** parameter (it's set automatically, as usual)
- It may have more parameters than just **self**
  - if this happens, the way in which the class name is used to create the object must reflect the **__init__** definition;
- **It can be used to set up the object**, i.e., properly initialize its internal state, create instance variables, instantiate any other objects if their existence is needed, etc.
- **It cannot return a value**, as it is designed to return a newly created object and nothing else;
- **It cannot be invoked directly either from the object or from inside the class** (you can invoke a constructor from any of the object's subclasses, but we'll discuss this issue later.)

As **__init__** is a method, and a method is a **function**

- you can do the same tricks with **constructors/methods** as you do with ordinary **functions**.

– – Continued – –

how to define a constructor with a default argument value:

```
class Classy:
   def __init__(self, value=None):
        # Constructor: runs automatically when a new object is created
        # 'value' defaults to None if no argument is provided
        self.var = value  # Instance variable unique to each object

# Create an object and pass a value to the constructor
obj_1 = Classy("object")

# Create another object without passing a value
# 'var' will be set to the default None
obj_2 = Classy()

# Access and print the instance variable from the first object
print(obj_1.var)
# Access and print the instance variable from the second object
print(obj_2.var)
```

Output:

object
None

Everything we've said about property name mangling applies to method names, too

– a method whose name starts with __ is (**partially**) **hidden**.

```
class Classy:
   def visible(self):
        # Public instance method
        # Accessible directly from outside the class
        print("visible")

   def __hidden(self):
        # "Private" method (name-mangled by Python)
        # Renamed internally to _Classy__hidden
        print("hidden")


# Create an instance of the Classy class
obj = Classy()

# Call the public method normally
obj.visible()

try:
   # This fails because __hidden is name-mangled
   # There is no attribute literally named '__hidden'
   obj.__hidden()
except:
   # Exception confirms the method is not directly accessible
   print("failed")

# Access the name-mangled method explicitly
# This bypasses the "privacy" convention
obj._Classy__hidden()
```

Output:
visible
failed
hidden

The inner life of classes and objects

Each Python class and each Python object is pre-equipped with a set of useful attributes which can be used to examine its capabilities.

- one of these being the **__dict__** property.

How it deals with methods:

```
class Classy:
   varia = 1          # Class variable (stored in the class namespace)

   def __init__(self):
       # Constructor runs when an object is created
       # Creates an instance variable in the object namespace
       self.var = 2

   def method(self):
       # Public instance method
       # Stored in the class namespace, not the object
       pass

   def __hidden(self):
       # Name-mangled "private" method
       # Stored as _Classy__hidden in the class namespace
       pass


# Create an instance of the class
obj = Classy()

# Instance namespace:
# Contains ONLY data created on the object itself
print(obj.__dict__)

# Class namespace:
# Contains class variables and all method definitions
# (including name-mangled private methods)
print(Classy.__dict__)
```

Output:

{'var': 2}

{'__module__': '__main__', '__firstlineno__': 1, 'varia': 1, '__init__': <function Classy.__init__ at 0x7e6686fa3060>, 'method': <function Classy.method at 0x7e6686dd6f00>, '_Classy__hidden': <function Classy.__hidden at 0x7e6686dd71c0>, '__static_attributes__': ('var',), '__dict__': <attribute '__dict__' of 'Classy' objects>, '__weakref__': <attribute '__weakref__' of 'Classy' objects>, '__doc__': None}

Another built-in property worth mentioning is **__name__**, which is **a string**.

- The property contains the name of the class. It's nothing exciting, just a string.

Note: the **__name__** attribute is absent from the object – **it exists only inside classes**.

If you want to find the class of a particular object, you can use a function named type(), which is able (among other things) to find a class which has been used to instantiate any object.

```
class Classy:
   pass

print(Classy.__name__)
obj = Classy()
print(type(obj).__name__)
```

Output:

Classy
Classy

Note: This alternative results in an **error**:

```
print(obj.__name__)
```

__module__ is a string, too – it stores the name of the module which contains the definition of the class.

Any module named **__main__** is actually not a module, but **the file currently being run**.

__bases__ is a **tuple**. The **tuple contains classes** (not class names) which are direct superclasses for the class.

The order is the same as that used inside the class definition.

- Note: only classes have this attribute – objects don't.

We've defined a function named printbases(), designed to present the tuple's contents clearly.

```
class SuperOne:
   # Base (super) class with no attributes or methods
   pass

class SuperTwo:
   # Another independent base (super) class
   pass

class Sub(SuperOne, SuperTwo):
   # Subclass that inherits directly from SuperOne and SuperTwo
   # The order here matters and is preserved in __bases__
   pass

def printBases(cls):
   # Function that prints the direct base classes of a class
   # Accepts a CLASS, not an object instance
   print('( ', end='')

   # __bases__ is a tuple containing the direct superclasses
   for x in cls.__bases__:
       # Each element is a class object
       # __name__ gives the class's name as a string
```

Note: a class without explicit superclasses points to an object (a predefined Python class) as its direct ancestor.

Reflection and introspection

All these means allow the Python programmer to perform two important activities specific to many objective languages. They are:

- **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.

In other words, you don't have to know a complete class/object definition to manipulate the object, as the object and/or its class contain the metadata allowing you to recognize its features during program execution.

Investigating classes

```
class MyClass:
   # Empty class definition
   # Attributes will be added dynamically at runtime
   pass

# Create an instance of MyClass
obj = MyClass()

# Dynamically add attributes to the object
obj.a = 1
obj.b = 2
obj.i = 3
obj.ireal = 3.5
obj.integer = 4
obj.z = 5

def incIntsI(obj):
   # Function that inspects and modifies object attributes
   # Operates directly on the object's instance namespace
   for name in obj.__dict__.keys():
       # Process only attribute names starting with 'i'
       if name.startswith('i'):
           # Retrieve the attribute value by name
           val = getattr(obj, name)
           # Modify only attributes whose value is an integer
           if isinstance(val, int):
               # Update the attribute in place
               setattr(obj, name, val + 1)

# Display the object's attributes before modification
print(obj.__dict__)

# Increment integer attributes starting with 'i'
incIntsI(obj)

# Display the object's attributes after modification
print(obj.__dict__)
```

Output:

```
{'a': 1, 'integer': 4, 'b': 2, 'i': 3, 'z': 5, 'ireal': 3.5}
{'a': 1, 'integer': 5, 'b': 2, 'i': 4, 'z': 5, 'ireal': 3.5}
```

The function named **incIntsI()** gets an object of any class, scans its contents in order to find all integer attributes with names starting with **i**, and increments them by **one**.

This is how it works:

- line 1: define a very simple class...
- lines 5 through 11: ... and fill it with some attributes;
- line 14: this is our function!
- line 15: scan the __dict__ attribute, looking for all attribute names;
- line 16: if a name starts with i...
- line 17: ... use the getattr() function to get its current value; note: getattr() takes two arguments: an object, and its property name (as a string), and returns the current attribute's value;
- line 18: check if the value is of type integer, and use the function isinstance() for this purpose (we'll discuss this later);
- line 19: if the check goes well, increment the property's value by making use of the setattr() function; the function takes three arguments: an object, the property name (as a string), and the property's new value.

Summary

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.

```
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()
```

Output:

```
My name is Sample living in a __main__
```

3.5 - OOP Fundamentals: Inheritance

Inheritance - why and how?

Before we start talking about inheritance, we want to present a new, handy mechanism utilized by Python's classes and objects – it's the way in which the object is able to introduce itself.

```
class Star:
    def __init__(self, name, galaxy):
        self.name = name
        self.galaxy = galaxy

sun = Star("Sun", "Milky Way")
print(sun)
```

Output:
<__main__.Star object at 0x7f1074cc7c50>

If you run the same code on your computer, you'll see something very similar, although the hexadecimal number (the substring starting with 0x) will be different, as it's just an internal object identifier used by Python, and it's unlikely that it would appear the same when the same code is run in a different environment.

As you can see, the printout here isn't really useful, and something more specific, or just prettier, may be more preferable.

Customizing How Objects Introduce Themselves:
When Python needs to convert an object into a string (for example, when using `print()`), it follows this process:

1. It looks for a method named **`__str__()`** in the object.
2. If found, it calls that method and prints the returned string.
3. If not found, it falls back to `object.`**`__str__()`**.

That default fallback is what produced the “ugly” output above.

You can override this behavior by defining your own **`__str__()`** method:

```
class Star:
    def __init__(self, name, galaxy):
        self.name = name
        self.galaxy = galaxy

    def __str__(self):
        return self.name + " in " + self.galaxy


sun = Star("Sun", "Milky Way")
print(sun)
```

Output:
Sun in Milky Way

This works because:

- print(sun) implicitly calls sun.__str__()
- Your method replaces the inherited default behavior
- The method must return a string

💡 In modern IDEs like PyCharm, you may see a **gutter** icon next to **__str__()** **indicating that you are overriding a built-in method.**
This is expected and correct.

Why This Matters for Inheritance

- Every Python class implicitly inherits from the built-in object class unless another superclass is specified.
- Methods like **__str__**, **__init__**, and **__repr__** already exist
  - You are customizing inherited behavior, not inventing something new
- This is your first real encounter with inheritance, even if it doesn't look like it yet
  - **Understanding this prepares you for explicit inheritance.**

Inheritance allows a new class (a **subclass**) to reuse and extend the behavior of an existing class (**a superclass**).

```
class Vehicle:
   pass


class LandVehicle(Vehicle):
   pass


class TrackedVehicle(LandVehicle):
   pass
```

Even though these classes are empty, they already form an inheritance chain:

- **Vehicle** is the **superclass**
- **LandVehicle** is a **subclass of Vehicle**
- **TrackedVehicle** is a **subclass of both** LandVehicle and Vehicle

This relationship is **transitive**:
If **TrackedVehicle** inherits from **LandVehicle**
and **LandVehicle** inherits from **Vehicle**, then..
TrackedVehicle also inherits from **Vehicle**

Python tracks inheritance **internally** and exposes it through special attributes and inspection tools
Inheritance is not about copying code — it's about extending behavior that already exists.

issubclass()

A function that will check if a particular class is a subclass of any other class.

- The function **returns True** if **ClassOne** is a subclass of **ClassTwo**
- Returns **False** otherwise.

– – Continued – –

There are two nested loops. Their purpose is to check all possible ordered pairs of classes, and to print the results of the check to determine whether the pair matches the subclass-superclass relationship.

```
class Vehicle:
    pass

class LandVehicle(Vehicle):
    pass

class TrackedVehicle(LandVehicle):
    pass

for cls1 in [Vehicle, LandVehicle, TrackedVehicle]:
    for cls2 in [Vehicle, LandVehicle, TrackedVehicle]:
        print(issubclass(cls1, cls2), end="\t")
    print()
```

Output:

```
True    False   False
True    True    False
True    True    True
```

isinstance()

An object is an instance of a class. This means the object was created using the class definition.

In many situations, it's important to determine what class an object was created from, because that tells you what attributes and methods the object is guaranteed to have.

Python provides the built-in **isinstance()** function for this purpose:

```
isinstance(object_name, ClassName)
```

- The function returns **True** if the object is an instance of the specified class or of any of its superclasses.
- Otherwise, it returns **False**.

If a class inherits from another class, its **objects automatically inherit the attributes and methods of the superclass**.

- As a result, an object created from a subclass is considered an instance of both the subclass and all of its superclasses.

```
class Vehicle:
    pass

class LandVehicle(Vehicle):
    pass

class TrackedVehicle(LandVehicle):
    pass

my_vehicle = Vehicle()
my_land_vehicle = LandVehicle()
my_tracked_vehicle = TrackedVehicle()

for obj in [my_vehicle, my_land_vehicle, my_tracked_vehicle]:
    for cls in [Vehicle, LandVehicle, TrackedVehicle]:
        print(isinstance(obj, cls), end="\t")
    print()
```

The is operator

The **is** operator checks whether two variables (object_one and object_two here) **refer to the same object**.

Don't forget that variables don't store the objects themselves, but only the handles pointing to the internal Python memory. Assigning a value of an object variable to another variable doesn't copy the object, but only its handle.

This is why an operator like **is** may be very useful in particular circumstances.

```
class SampleClass:
   def __init__(self, val):
       # Store the value as an instance variable
       self.val = val

# Create two separate objects with different initial values
object_1 = SampleClass(0)
object_2 = SampleClass(2)

# object_3 is NOT a new object
# It is another reference to the SAME object as object_1
object_3 = object_1

# Modify the value through object_3
# Since object_3 and object_1 reference the same object,
# this change affects object_1 as well
object_3.val += 1

# 'is' checks object identity (same object in memory)
print(object_1 is object_2)  # False: two different objects
print(object_2 is object_3)  # False: object_2 is separate
print(object_3 is object_1)  # True: same object, two references

# Show the stored values
# object_1 and object_3 share the same value
print(object_1.val, object_2.val, object_3.val)

# Strings are immutable objects
string_1 = "Mary had a little "
string_2 = "Mary had a little lamb"

# This creates a NEW string object
# string_1 is reassigned to reference it
string_1 += "lamb"

# '==' checks value equality
# 'is' checks object identity
print(string_1 == string_2, string_1 is string_2)
```

- there is a very simple class equipped with a simple constructor, creating just one property.
    - The class is **used to instantiate two objects**.
    - The former is then assigned to another variable, and its **val** property is **incremented by one**.
- afterward, the **is** operator is applied three times to check all possible pairs of objects, and all **val** property values are also printed.
- the last part of the code carries out another experiment. After three assignments, both strings contain the same texts, but these texts are stored in different objects.

The results prove that object_1 and object_3 are actually the same objects, while string_1 and string_2 aren't, despite their contents being the same.

How Python finds properties and methods

Time for an educational video!

- https://www.youtube.com/watch?v=BJ-VvGyQxho

```
class Super:
    def __init__(self, name):
        # Constructor for the superclass
        # Initializes the 'name' attribute on the object
        self.name = name

    def __str__(self):
        # String representation defined in the superclass
        # Will be used if the subclass does not override it
        return "My name is " + self.name + "."

class Sub(Super):
    def __init__(self, name):
        # Constructor for the subclass
        # Explicitly call the superclass constructor
        # This ensures 'name' is initialized correctly
        Super.__init__(self, name)

# Create an instance of the subclass
obj = Sub("Andy")

# print() looks for __str__ in the following order:
# 1. Sub class
# 2. Super class
# 3. object (built-in base class)
print(obj)
```

- There is a class named **Super**, which defines its own constructor used to assign the object's property, named **name**.
- the class defines the **__str__()** method, too, which makes the class able to present its identity in clear text form.
- The class is next used as a base to create a subclass named **Sub**. **The Sub class defines its own constructor**, which invokes the one from the superclass.
    - Note how we've done it: **Super.__init__(self, name)**.
- we've explicitly named the superclass, and pointed to the method to invoke **__init__()**, providing all needed arguments.
- we've instantiated one object of class Sub and printed it.

– – Continued – –

Note: As there is no **__str__()** method within the Sub class, the printed string is to be produced within the **Super** class. This means that the **__str__()** method has been inherited by the **Sub** class.

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

```
super().__init__(name)
```

The **super()** function creates a context in which you don't have to (moreover, you mustn't) pass the self argument to the method being invoked – this is why it's possible to activate the superclass constructor using only one argument.

Note: 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.

Class Variables

```
class Super:
    # Class variable defined in the Super (parent) class.
    # This variable belongs to the class itself, not to any specific object.
    supVar = 1

class Sub(Super):
    # Sub inherits from Super, so it automatically has access
    # to all public class variables and methods defined in Super.
    # This is the inheritance relationship.
    subVar = 2  # Class variable defined in the Sub (child) class.

# Create an object (instance) of the Sub class.
# The object itself has no instance variables at this point.
obj = Sub()

# Python tries to find 'subVar':
# 1) It looks inside the object itself → not found.
# 2) It looks inside the Sub class → found (value = 2).
print(obj.subVar)

# Python tries to find 'supVar':
# 1) It looks inside the object itself → not found.
# 2) It looks inside the Sub class → not found.
# 3) It looks inside the Super class (via inheritance) → found (value = 1).
print(obj.supVar)
```

Output:

```
2
1
```

As you can see, the **Super** class defines one class variable named **supVar**, and the **Sub** class defines a variable named **subVar**.

- Both these variables are visible inside the object of class Sub

instance variables

```
class Super:
   def __init__(self):
       # Instance variable defined in the Super (parent) class.
       # This variable is created only when the Super constructor is executed.
       self.supVar = 11

class Sub(Super):
   def __init__(self):
       # Call the constructor of the Super class.
       # This ensures that 'supVar' is created inside the object.
       super().__init__()

       # Instance variable defined in the Sub (child) class.
       # This variable is added directly to the object after the Super constructor runs.
       self.subVar = 12

# Create an object (instance) of the Sub class.
# Both constructors are executed:
# 1) Super.__init__() creates 'supVar'
# 2) Sub.__init__() creates 'subVar'
obj = Sub()

# Python looks for 'subVar':
# 1) Inside the object itself → found (value = 12).
print(obj.subVar)

# Python looks for 'supVar':
# 1) Inside the object itself → found (value = 11).
# The value exists only because the Super constructor was called.
print(obj.supVar)
```

Output:
12
11

The **Sub** class constructor creates an instance variable named **subVar**, while the **Super** constructor does the same with a variable named **supVar**. As previously, both variables are accessible from within the object of class Sub.

- Note: the existence of the **supVar** variable is obviously conditioned by the **Super** class constructor invocation. Omitting it would result in the absence of the variable in the created object.

A general statement describing Python's behavior:

When you try to access any object's entity, Python will try to (in this order):

- find it inside the object itself;
- find it in all classes involved in the object's inheritance line from bottom to top;

If both of the above fail, an exception (**AttributeError**) is raised.

– – Continued – –

Multiple inheritance

occurs when a class has more than one **superclass**. Syntactically, such inheritance is presented as a comma-separated list of superclasses put inside parentheses after the new class name.

```
class SuperA:
   # Class variable defined in the first superclass.
   var_a = 10

   def fun_a(self):
       # Instance method defined in SuperA.
       # It becomes available to all subclasses of SuperA.
       return 11

class SuperB:
   # Class variable defined in the second superclass.
   var_b = 20

   def fun_b(self):
       # Instance method defined in SuperB.
       # It also becomes available to subclasses.
       return 21

class Sub(SuperA, SuperB):
   # Sub inherits from SuperA first, then SuperB.
   # The order is important and defines the Method Resolution Order (MRO).
   # Python will search SuperA before SuperB when looking for attributes.
   pass

# Create an object (instance) of the Sub class.
# The object itself has no instance variables or methods of its own.
obj = Sub()

# Python looks for 'var_a' and 'fun_a':
# 1) Inside the object → not found.
# 2) Inside class Sub → not found.
# 3) Inside SuperA (first in MRO) → found.
print(obj.var_a, obj.fun_a())

# Python looks for 'var_b' and 'fun_b':
# 1) Inside the object → not found.
# 2) Inside class Sub → not found.
# 3) Inside SuperA → not found.
# 4) Inside SuperB → found.
print(obj.var_b, obj.fun_b())
```

Output:

10 11
20 21

The **Sub** class has two **superclasses**: **SuperA** and **SuperB**. This means that the **Sub** class **inherits all the goods offered by both SuperA** and **SuperB**.

– – Continued – –

Overriding

```
class Level1:
    # Class variable defined in the top-level class.
    var = 100
    def fun(self):
        # Method defined in Level1.
        return 101

class Level2(Level1):
    # Class variable redefined in Level2.
    # This overrides the variable with the same name from Level1.
    var = 200
    def fun(self):
        # Method redefined in Level2.
        # This overrides the method from Level1.
        return 201

class Level3(Level2):
    # Level3 does not define its own variables or methods.
    # It inherits everything from Level2 and Level1.
    pass

# Create an object (instance) of the Level3 class.
obj = Level3()

# Python looks for 'var':
# 1) Inside the object itself → not found.
# 2) Inside class Level3 → not found.
# 3) Inside class Level2 → found (value = 200).

# Python looks for 'fun()':
# 1) Inside the object itself → not found.
# 2) Inside class Level3 → not found.
# 3) Inside class Level2 → found (returns 201).

# Definitions in Level1 are ignored because Level2 overrides them.
print(obj.var, obj.fun())
```

The entity defined later (in the inheritance sense) **overrides the same entity defined earlier**.

This feature can be intentionally used to modify default class behaviors when any of its classes needs to act in a different way to its ancestor.

We can say that Python looks for object components in the following order:

- inside the object itself;
- in its superclasses, **from bottom to top**;
- if there is more than one class on a particular inheritance path, Python **scans them from left to right**.

– – Continued – –

How to build a hierarchy of classes

Note: the situation in which the **subclass** is able to modify its superclass behavior is called **polymorphism**.

- The method is called **virtual**.

```
import time

class Vehicle:
    def change_direction(self, left, on):
        # This method defines a common interface for changing direction.
        # It is intended to be overridden in subclasses.
        # The base class does not implement any behavior.
        pass

    def turn(self, left):
        # This method defines a general turning algorithm.
        # It does not know *how* the vehicle turns—only *that* it can.
        # The actual behavior depends on the subclass implementation
        # of the change_direction() method.
        self.change_direction(left, True)
        time.sleep(0.25)
        self.change_direction(left, False)

class TrackedVehicle(Vehicle):
    def control_track(self, left, stop):
        # Method specific to tracked vehicles.
        # 'left' selects the track, 'stop' controls its movement.
        pass

    def change_direction(self, left, on):
        # Overrides Vehicle.change_direction().
        # Tracked vehicles change direction by controlling one track.
        self.control_track(left, on)

class WheeledVehicle(Vehicle):
    def turn_front_wheels(self, left, on):
        # Method specific to wheeled vehicles.
        # 'left' selects direction, 'on' enables steering.
        pass

    def change_direction(self, left, on):
        # Overrides Vehicle.change_direction().
        # Wheeled vehicles change direction by turning the front wheels.
        self.turn_front_wheels(left, on)
```

- we defined a **superclass** named **Vehicle**, which uses the **turn()** method to implement a general scheme of turning, while the turning itself is done by a method named **change_direction()**;
  - note: the former method is empty, as we are going to put all the details into the **subclass**
  - such a method is often called an abstract method, as it only demonstrates some possibility which will be instantiated later
- we defined a **subclass** named **TrackedVehicle** (note: it's derived from the **Vehicle class**) which instantiated the **change_direction()** method by using the specific (concrete) method named **control_track()**
- respectively, the subclass named **WheeledVehicle** does the same trick, but uses the **turn_front_wheels()** method to force the vehicle to turn.

The most important advantage is that this form of code enables you to implement a brand new turning algorithm **just by modifying the turn() method**, **which can be done in just one place**, as all the vehicles will obey it.

- This is how **polymorphism helps the developer to keep the code clean and consistent**.

Composition

The process of composing an object using other different objects.

- The objects used in the composition deliver a set of desired traits (properties and/or methods)
    - They act like blocks used to build a more complicated structure.
- **inheritance extends a class's capabilities** by adding new components and modifying existing ones;
    - In other words, the complete recipe is contained inside the class itself and all its ancestors; the object takes all the class's belongings and makes use of them.
- **composition projects a class as a container** able to store and use other objects where each of the objects implements a part of a desired class's behavior.

The **class** – like in the previous example – is aware of how to turn the **vehicle**, but the actual turn is done by a specialized object stored in a property named **controller**. The **controller** is able to control the vehicle by manipulating the relevant vehicle's parts.

```
import time

class Tracks:
   def change_direction(self, left, on):
       # Method implementation for tracked vehicles.
       # Prints how tracks respond to a direction change.
       print("tracks: ", left, on)

class Wheels:
   def change_direction(self, left, on):
       # Method implementation for wheeled vehicles.
       # Prints how wheels respond to a direction change.
       print("wheels: ", left, on)

class Vehicle:
   def __init__(self, controller):
       # The Vehicle class does not care what type of controller it uses.
       # It only assumes that the controller object has a
       # change_direction(left, on) method.
       self.controller = controller

   def turn(self, left):
       # The turning algorithm is defined once.
       # The actual behavior depends on the controller object provided.
       self.controller.change_direction(left, True)
       time.sleep(0.25)
       self.controller.change_direction(left, False)

# Create Vehicle objects with different controllers.
# Both controllers implement the same method name.
wheeled = Vehicle(Wheels())
tracked = Vehicle(Tracks())

# The same method call ('turn') produces different behavior
# depending on the controller object.
wheeled.turn(True)
tracked.turn(False)
```

Output:

```
wheels:  True True
wheels:  True False
tracks:  False True
tracks:  False False
```

There are two **classes** named **Tracks** and **Wheels** – they know how to control the vehicle's direction. There is also a **class** named **Vehicle** which can use any of the available controllers (the two already defined, or any others defined in the future) – **the controller itself is passed to the class during initialization**.

In this way, **the vehicle's ability to turn is composed using an external object**, not implemented inside the Vehicle class.

In other words, we have a universal vehicle and can install either tracks or wheels onto it.

Single inheritance vs. multiple inheritance

As you already know, there are no obstacles to using multiple inheritance in Python. You can derive any new class from more than one previously defined classes.
There is only one "but". The fact that you can do it does not mean you have to.
Don't forget that:

- **a single inheritance class is always simpler, safer, and easier to understand and maintain;**
- multiple inheritance is always risky, as you have many more opportunities to make a mistake in identifying these parts of the superclasses which will effectively influence the new class;
- multiple inheritance may make overriding extremely tricky; moreover, using the super() function becomes ambiguous;
- multiple inheritance violates the single responsibility principle (more details here: https://en.wikipedia.org/wiki/Single_responsibility_principle) as it makes a new class of two (or more) classes that know nothing about each other;
- we strongly **suggest multiple inheritance as the last of all possible solutions** – if you really need the many different functionalities offered by different classes, composition may be a better alternative.

What is Method Resolution Order and why is it that not all inheritances make sense?

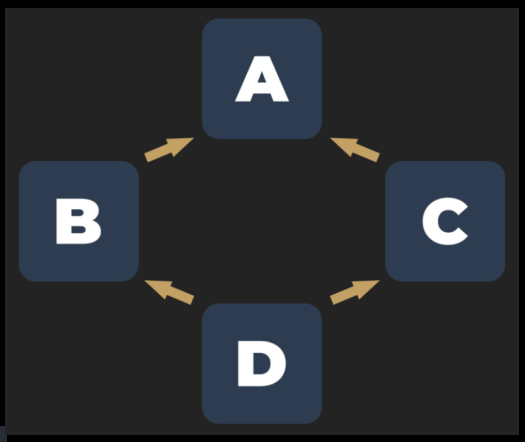
MRO, in general, is a way (call it a **strategy**) in which a particular programming language scans through the upper part of a class's hierarchy in order to find the method it currently needs. It's worth emphasizing that different languages use slightly (or even completely) different MROs. Python is a unique creature in this respect, however, and its customs are a bit specific.

This chapter was poorly written and over-bloated. Chat GPT Did it better:

MRO (Method Resolution Order): the rule-set Python uses to turn a multiple-inheritance class *graph* into a single, predictable *search list* (a "linearization") so it can decide which method/attribute to use. Python documentation

- **What it does:** defines the exact order Python checks classes when you access `obj.method` (or any attribute) and more than one parent could provide it.
- **How Python decides:** Python uses the **C3 linearization** rules (e.g., respects local left-to-right base order *and* keeps inheritance consistent/monotonic).
- **Why your "Top, Middle" swap breaks:** if the requested base-class order contradicts the required inheritance constraints, Python can't build a consistent linearization and raises **`TypeError: Cannot create a consistent method resolution order (MRO)`**.

The diamond problem



The second example of the spectrum of issues that can possibly arise from multiple inheritance is illustrated by a classic problem named the diamond problem. The name reflects the shape of the inheritance diagram - take a look at the picture.

- there is the top-most superclass named A;
- there are two subclasses derived from A: B and C;
- and there is also the bottom-most subclass named D, derived from B and C (or C and B, as these two variants mean different things in Python)

The same structure expressed in **Python**:

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

d = D()
```

Some programming languages don't allow multiple inheritance at all, and as a consequence, they won't let you build a diamond – this is the route that Java and C# have chosen to follow since their origins.

Python, however, has chosen a different route – it allows multiple inheritance, and it doesn't mind if you write and run code like the one in the editor. But don't forget about MRO – it's always in charge.

– – Continued – –

Summary:

1. A method named **__str__()** is responsible for converting an object's contents into a 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. An **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):
    tpass

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)
```

3.6 - Exceptions once again

More about exceptions

The object-oriented nature of Python's exceptions makes them a very flexible tool, able to fit to specific needs, even those you don't yet know about.

```
def reciprocal(n):
    try:
        n = 1 / n
    except ZeroDivisionError:
        print("Division failed")
        return None
    else:
        print("Everything went fine")
        return n

print(reciprocal(2))
print(reciprocal(0))
```

Output:
Everything went fine
0.5
Division failed
None

A code labelled in this way is executed when no exception has been raised inside the **try:** part.

We can say that exactly one branch can be executed after **try:** – either the one beginning with except (don't forget that there can be more than one branch of this kind) or the one starting with else.

Note: the **else:** branch has to be located after the last **except** branch.

The **try-except** block can be extended in one more way – adding a part headed by the **finally keyword**

- It must be the last branch of the code designed to handle exceptions.
- Note: **else** and **finally**, aren't dependent in any way, and they can coexist or occur independently.

The finally block is always executed, no matter what happened earlier, even when raising an exception, no matter whether this has been handled or not.

- It finalizes the try-except block execution, hence its name.

```
def reciprocal(n):
    try:
        n = 1 / n
    except ZeroDivisionError:
        print("Division failed")
        n = None
    else:
        print("Everything went fine")
    finally:
        print("It's time to say goodbye")
        return n


print(reciprocal(2))
print(reciprocal(0))
```

Output:
Everything went fine
It's time to say goodbye
0.5
Division failed
It's time to say goodbye
None

Exceptions are classes

```
try:
   i = int("Hello!")
except Exception as e:
   print(e)
   print(e.__str__())
```

Output:

```
invalid literal for int() with base 10: 'Hello!'
invalid literal for int() with base 10: 'Hello!'
```

As you can see, the **except** statement is **extended**, and contains an additional phrase starting with the **as keyword**, followed by an identifier.

- **The identifier is designed to catch the exception object** so you can analyze its nature and draw some useful conclusions.
- Note: the identifier's scope covers its except branch, and doesn't go any further.

The example presents a very simple way of utilizing the received object – just print it outand it contains a brief message describing the reason.

- The output is produced by the object's **__str__()** method.

The same message will be printed if there is no fitting except block in the code, and Python is forced to handle it alone.

All the built-in Python exceptions form a hierarchy of classes. There is no obstacle to extending it if you find it reasonable.

This program dumps all predefined exception classes in the form of a tree-like printout.

```
def print_exception_tree(thisclass, nest=0):
   if nest > 1:
       print("   |" * (nest - 1), end="")
   if nest > 0:
       print("   +---", end="")

   print(thisclass.__name__)

   for subclass in thisclass.__subclasses__():
       print_exception_tree(subclass, nest + 1)

print_exception_tree(BaseException)
```

As a tree is a perfect example of a recursive data structure, a recursion seems to be the best tool to traverse through it. The **print_exception_tree()** function takes two arguments:

- a point inside the tree from which we start traversing the tree;
- a nesting level
  - we'll use it to build a simplified drawing of the tree's branches

The root of Python's exception classes is the **BaseException** class (it's a superclass of all other exceptions).

For each of the encountered classes, perform the same set of operations:

- print its name, taken from the **__name__** property;
- iterate through the list of subclasses delivered by the **__subclasses__()** method, and recursively invoke the **print_exception_tree()** function, incrementing the nesting level respectively.

Note how we've drawn the branches and forks. The printout isn't sorted in any way – you can try to sort it yourself, if you want a challenge. Moreover, there are some subtle inaccuracies in the way in which some branches are presented. That can be fixed, too, if you wish.

Detailed anatomy of exceptions

```
def print_args(args):
    # 'args' is expected to be a tuple.
    # In this program, it will always be Exception.args,
    # which stores all arguments passed to the Exception constructor.
    lng = len(args)  # Determine how many arguments the exception contains.
    if lng == 0:
        # If the exception was raised with no arguments, Exception.args is an empty tuple.
        print("")
    elif lng == 1:
        # If exactly one argument was passed,
        # Python conventionally treats it as the exception's message.
        print(args[0])
    else:
        # If more than one argument was passed,
        # Exception.args remains a tuple and should be displayed as such.
        print(str(args))
try:
    # Raise a generic Exception with no arguments.
    raise Exception
except Exception as e:
    # 'e' is the caught exception object.
    # Printing 'e' implicitly calls e.__str__().
    print(e, e.__str__(), sep=' : ', end=' : ')

    # Pass the tuple of exception arguments to the helper function.
    print_args(e.args)
try:
    # Raise an Exception with a single string argument.
    # This becomes the exception's message.
    raise Exception("my exception")
except Exception as e:
    # 'e' and 'e.__str__()' produce identical output
    # because __str__ returns the string form of the single argument.
    print(e, e.__str__(), sep=' : ', end=' : ')
    print_args(e.args)
try:
    # Raise an Exception with multiple arguments.
    # These are stored as a tuple inside e.args.
    raise Exception("my", "exception")
except Exception as e:
    # When multiple arguments are present,
    # __str__ returns the tuple's string representation.
    print(e, e.
```

We've used the function to print the contents of the args property in three different cases, where the exception of the Exception class is raised in three different ways. To make it more spectacular, we've also printed the object itself, along with the result of the **__str__()** invocation.

Output:

```
:  :
my exception : my exception : my exception
('my', 'exception') : ('my', 'exception') : ('my', 'exception')
```

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, and so on.
    - We don't count the self argument here

The first case looks routine – there is just the name **Exception** after the **raise** keyword.

- This means that the object of this class has been created in a most routine way.

The second and third cases may look a bit weird at first glance, but there's nothing odd here – these are just the constructor invocations. In the second raise statement, the constructor is invoked with one argument, and in the third, with two.

How to create your own exception

The exceptions hierarchy is neither closed nor finished, and you can always extend it if you want or need to create your own world populated with your own exceptions.

It may be useful when you create a complex module which detects errors and raises exceptions, and you want the exceptions to be easily distinguishable from any others brought by Python.

This is done by defining your own, new exceptions as subclasses derived from predefined ones.

- Note: if you want to create an exception which will be utilized as a specialized case of any built-in exception, derive it from just this one.
- If you want to build your own hierarchy, and don't want it to be closely connected to Python's exception tree, derive it from any of the top exception classes, like Exception.

Imagine that you've created a brand new arithmetic, ruled by your own laws and theorems. It's clear that division has been redefined, too, and has to behave in a different way than routine dividing. It's also clear that this new division should raise its own exception, different from the built-in ZeroDivisionError, but it's reasonable to assume that in some circumstances, you (or your arithmetic's user) may want to treat all zero divisions in the same way.

Demands like these may be fulfilled in the manner shown here:

```
class MyZeroDivisionError(ZeroDivisionError):
    # Custom exception class.
    # It INHERITS from ZeroDivisionError, which means:
    # 1) It is a ZeroDivisionError
    # 2) It can be caught by handlers for ZeroDivisionError
    # This is critical for understanding exception hierarchy and handler order.
    pass
def do_the_division(mine):
    # This function deliberately raises different exceptions
    # depending on the boolean argument 'mine'.
    if mine:
        # Raise the custom exception.
        # The argument becomes the exception's message.
        raise MyZeroDivisionError("some worse news")
    else:
        # Raise the built-in ZeroDivisionError.
        raise ZeroDivisionError("some bad news")
for mode in [False, True]:
    try:
        # Call the function with False, then True.
        do_the_division(mode)
    except ZeroDivisionError:
        # This handler catches BOTH:
        # - ZeroDivisionError
        # - MyZeroDivisionError (because it is a subclass)
        # This demonstrates polymorphism in exception handling.
        print('Division by zero')

for mode in [False, True]:
    try:
        do_the_division(mode)
```

```
    except MyZeroDivisionError:
        # This handler catches ONLY the custom exception.
        # It must appear BEFORE ZeroDivisionError,
        # otherwise it would never be reached.
        print('My division by zero')

    except ZeroDivisionError:
        # This handler catches the original exception ONLY.
        # It will NOT catch MyZeroDivisionError here,
        # because that case is already handled above.
        print('Original division by zero')
```

- We've defined our own exception, named **MyZeroDivisionError**, derived from the built-in **ZeroDivisionError**. As you can see, we've decided not to add any new components to the class.
    - In effect, an exception of this class can be treated like a plain ZeroDivisionError, or considered separately.
- The **do_the_division()** function raises either a **MyZeroDivisionError** or **ZeroDivisionError exception**, depending on the argument's value.

  The function is invoked four times in total, while the first two invocations are handled using only one except branch (the more general one) and the last two ones with two different branches, able to distinguish the exceptions
- **Don't forget: the order of the branches makes a fundamental difference!**

When you're going to build a completely new universe filled with completely new creatures that have nothing in common with all the familiar things, you may want to build your own exception structure.

For example, if you work on a large simulation system which is intended to model the activities of a pizza restaurant, it can be desirable to form a separate hierarchy of exceptions.

You can start building it by defining a general exception as a new base class for any other specialized exception.

We've done it in the following way:

```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(self, message)
        self.pizza = pizza
```

Note: we're going to collect more specific information here than a regular Exception does, so our constructor will take two arguments:

- one specifying a pizza as a subject of the process,
- and one containing a more or less precise description of the problem.

As you can see, we pass the second parameter to the superclass constructor, and save the first inside our own property.

A more specific problem (like an excess of cheese) can require a more specific exception. It's possible to derive the new class from the already defined PizzaError class, like we've done here:

```
class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError._init__(self, pizza, message)
        self.cheese = cheese
```

The **TooMuchCheeseError** exception needs more information than the regular **PizzaError** exception, so we add it to the constructor – the name **cheese** is then stored for further processing.

Below: Coupled together the two previously defined exceptions and harnessed them to work in a small example snippet.

```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(self, message)
        self.pizza = pizza

class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese

def make_pizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa', 'calzone']:
        raise PizzaError(pizza, "no such pizza on the menu")
    if cheese > 100:
        raise TooMuchCheeseError(pizza, cheese, "too much cheese")
    print("Pizza ready!")

for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:
    try:
        make_pizza(pz, ch)
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)
```

One of these is raised inside the **make_pizza()** function when any of these two erroneous situations is discovered:

- a wrong pizza request
- a request for too much cheese.

Note:

- removing the branch starting with except **TooMuchCheeseError** will cause all appearing exceptions to be classified as **PizzaError**;
- removing the branch starting with except **PizzaError** will cause the **TooMuchCheeseError** exceptions to remain **unhandled**, and will **cause the program to terminate**.

The previous solution, although elegant and efficient, has one important weakness. Due to the somewhat easygoing way of declaring the constructors, **the new exceptions cannot be used as they are without a full list of required arguments**.

Remove this weakness by setting the default values for all constructor parameters:

```
class PizzaError(Exception):
    def __init__(self, pizza='unknown', message=''):
        Exception.__init__(self, message)
        self.pizza = pizza

class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza='uknown', cheese='>100', message=''):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese

def make_pizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa', 'calzone']:
        raise PizzaError
    if cheese > 100:
        raise TooMuchCheeseError
    print("Pizza ready!")
```

Now, if the circumstances permit, it is possible to use the class names alone.

Summary

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.

```
try:
    assert __name__ == "__main__"
except:
    print("fail", end=' ')
else:
    print("success", end=' ')
finally:
    print("done")
```

Output: **success done**.

Module 4

4.1 - Generators, iterators, and closures

Generators

A Python generator is a piece of specialized code able to produce a series of values, and to control the iteration process. This is why generators are very often called iterators, and although some may find a very subtle distinction between these two, we'll treat them as one.

```
for i in range(5):
print(i)
```

The **range()** function is, in fact, a generator, which is an iterator.

- The **range()** generator is **invoked six times**, **providing five subsequent values** from zero to four, and finally signaling that the series is **complete**.

Difference between a generator and a function:

- A function returns one, well-defined value only once.
- A generator returns a series of values, and in general, is (implicitly) invoked more than once.

The **iterator** protocol:
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, and **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 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.

```
class Fib:
    # This class implements a CUSTOM ITERATOR.
    # It follows the iterator protocol by defining:
    #   1) __iter__()
    #   2) __next__()
    def __init__(self, nn):
        # Constructor runs once when the object is created.
        # 'nn' defines how many Fibonacci numbers will be produced.
        print("__init__")
        self.__n = nn      # Maximum number of values to generate
        self.__i = 0       # Counter tracking how many values were returned
        self.__p1 = 1      # Previous Fibonacci value (F(n-2))
        self.__p2 = 1      # Previous Fibonacci value (F(n-1))
    def __iter__(self):
        # __iter__ must return an iterator object.
        # Since this object IS the iterator,
        # it returns itself.
        print("__iter__")
        return self
    def __next__(self):
        # __next__ is called automatically by:
        # - the for loop
        # - next()
        # It must either:
        # - return the next value
        # - or raise StopIteration when exhausted
        print("__next__")
```

```
        self.__i += 1

        if self.__i > self.__n:
            # Signals the end of iteration to the for loop.
            raise StopIteration
        if self.__i in [1, 2]:
            # The first two Fibonacci numbers are defined as 1.
            return 1

        # Calculate the next Fibonacci number.
        ret = self.__p1 + self.__p2

        # Update internal state for the next iteration.
        self.__p1, self.__p2 = self.__p2, ret
        return ret
# The for loop:
# 1) Calls Fib(10)  -> __init__()
# 2) Calls iter()   -> __iter__()
# 3) Repeatedly calls __next__()
# 4) Stops when StopIteration is raised
for i in Fib(10):
    print(i)
```

We've built a class able to iterate through the first n values (where n is a constructor parameter) of the Fibonacci numbers.

Let us remind you – the Fibonacci numbers (Fibi) are defined as follows:
Fib1 = 1
Fib2 = 1
Fibi = Fibi-1 + Fibi-2

In other words:

- the first two Fibonacci numbers are equal to 1;
- any other Fibonacci number is the sum of the two previous ones (e.g., Fib3 = 2, Fib4 = 3, Fib5 = 5, and so on)

Let's dive into the code:

- **lines 6 through 13**: the class constructor prints a message (we'll use this to trace the class's behavior), prepares some variables
  - **__n** to store the series limit, **__i** to track the current **Fibonacci** number to provide, and **__p1** along with **__p2** to save the two previous numbers
- **lines 14 through 19**: the **__iter__** method is obliged to return the iterator object itself
  - its purpose may be a bit ambiguous here, but there's no mystery; try to imagine an object which is not an iterator (e.g., it's a collection of some entities), but one of its components is an iterator able to scan the collection
  - The **__iter__** method should extract the iterator and entrust it with the execution of the iteration protocol; as you can see, the method starts its action by printing a message;
- **lines 20 through 38**: the **__next__** method is responsible for creating the sequence
  - It's somewhat wordy, but this should make it more readable
    - first, it prints a message,
    - then it updates the number of desired values,
      - if it reaches the end of the sequence, the method breaks the iteration by raising the StopIteration exception
    - the rest of the code is simple, and it precisely reflects the definition we showed you earlier;
- **lines 53 and 55** make use of the iterator.

What ends up happening:

- the iterator object is instantiated first;
- next, Python invokes the **__iter__** method to get access to the actual iterator;
- the **__next__** method is invoked eleven times – the first ten times produce useful values, while the eleventh terminates the iteration.

The yield statement

This lesson was **terrible**, so I replaced it with **ChatGPT**:

Why yield exists

- Manual iterators (__iter__, __next__) must store iteration state themselves
- This makes code longer, harder to read, and easier to break
- yield lets Python handle state management automatically

return vs yield

```
def fun(n):
    for i in range(n):
        return i
```

- **return** ends the function immediately
- State is discarded
- Every call starts from the beginning
  ❌ Cannot act as an iterator or generator

```
def fun(n):
    for i in range(n):
        yield i
```

- **yield** produces a value without ending the function
- Execution state is paused and saved
- Next call resumes exactly where it stopped
  ✔ This function is now a generator

What actually changes

- yield turns the function into a generator object
- Calling it does not run the code
- Iteration (for, next()) drives execution
- Python automatically implements:
  - __iter__()
  - __next__()
  - StopIteration

One-line takeaway: **yield** = **return** + **state preservation**

– – Continued – –

How to build a generator

This lesson was **terrible**, so I replaced it with **ChatGPT**:

1. A generator is just a function with yield:

```
def fun(n):
    for i in range(n):
        yield i
```

- Presence of **yield** turns the function into a **generator**
- Calling it returns a **generator object**, not values
- **Values** appear only when iterated

```
for v in fun(5):
    print(v)
```

Output: 0 2 3 3 4

2. Generators keep state automatically

```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2
```

- Local variables (power) **retain their values between yields**
- No manual counters, **no __next__**, **no StopIteration**

Output: 1 2 4 8 16 32 64 128

3. Generators work anywhere an iterable is expected

```
t = [x for x in powers_of_2(5)]
```

- Generator is consumed to build the list

Output: [1, 2, 4, 8, 16]

list() Conversion

```
t = list(powers_of_2(3))
```

- Forces full evaluation of the generator

Output: [1, 2, 4]

in Operator

```
if i in powers_of_2(4):
```

- Generator is **iterated internally**
- Stops early if match is found
- Generator is **single-use**

Checks membership against: 1, 2, 4, 8

4. Generator vs iterator (why this matters)

```
def fibonacci(n):
    p = pp = 1
    for i in range(n):
        if i in [0, 1]:
            yield 1
        else:
            n = p + pp
            pp, p = p, n
            yield n
```

- Same logic as iterator class
- No class
- No protocol methods
- Cleaner and safer

Result: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

More about list comprehensions

Key note: **Brackets** make a **comprehension**, the **parentheses** make a **generator**.

A quick reminder of **list comprehensions**:

```
list_1 = []

for ex in range(6):
   list_1.append(10 ** ex)

list_2 = [10 ** ex for ex in range(6)]

print(list_1)
print(list_2)
```

list_1 employs a routine way to utilize the for loop,
list_2 makes use of the list comprehension and builds the list in situ, without needing a loop, or any other extended code.

Conditional Expressions:

A way of selecting one of two different values based on the result of a Boolean expression.

- it is not a conditional instruction,
    - it's not an instruction at all.
        - **It's an operator.**

```
the_list = []

for x in range(10):
   the_list.append(1 if x % 2 == 0 else 0)

print(the_list)
```

The code fills a list with 1s and 0s

- if the index of a particular element is odd, the element is set to **0**
- Set to 1 otherwise.

This is considered “elegant”

– – Continued – –

The lambda function

PEP 8, the Style Guide for Python Code, **recommends** that **lambdas should not be assigned to variables**, but rather they should be defined as **functions**.

This means that it is better to use a **def statement**, and **avoid using an assignment statement that binds a lambda expression to an identifier**. Analyze the code below:

ChatGPT defines it better:

Think of a lambda as: "I need a tiny transformation right here, and I don't want to pollute the namespace."

A Python `lambda` is:

- **A function object** (same type as one made with `def`)
- **Defined at expression-time**, not statement-time
- **Restricted to a single expression**
- **Automatically returns** the value of that expression

What it cannot have:

- Statements (if, for, while, try, return, print, etc.)
- Multiple expressions
- Assignments
- Docstrings
- An explicit name

They exist to support functional-style operations, such as:

- **map()**
- **filter()**
- **sorted(key=...)**

Where:

- The function is simple
- Used once
- Has no semantic meaning worth naming

Cisco Notes Continued:

A lambda function is a **function without a name**

- You can also call it an anonymous function
- you can name such a function if you really need
- in many cases the lambda function can exist and work while remaining fully incognito

```
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()))
```

- The first **lambda** is an anonymous parameterless function that always returns **2**.
    - As we've assigned it to a variable named **two**, we can say that **the function is not anonymous anymore**, and we can use the name to invoke it.
- The second one (**sqr**) is a **one-parameter anonymous function** that returns the value of its squared argument. We've named it as such, too.
- The third **lambda** (**pow**) takes two parameters and returns the value of the first one raised to the power of the second one. The name of the variable which carries the lambda speaks for itself.

- **We don't use `pow` in order to avoid confusion with the built-in function of the same name and the same purpose.**

How to use lambdas and what for

The most interesting thing about lambdas is that you can use them in their **pure form**
– as anonymous parts of code intended to evaluate a result.

Imagine that we need a function (**print_function**) which prints the values of a given function for a set of selected arguments.

We want **print_function** to be universal
– it should accept a set of arguments put in a list and a function to be evaluated, both as arguments
– we don't want to hardcode anything.

```
def print_function(args, fun):
    for x in args:
        print('f(', x, ')=', fun(x), sep='')

def poly(x):
    return 2 * x ** 2 - 4 * x + 2

print_function([x for x in range(-2, 3)], poly)
```

The **print_function()** function takes **two parameters**:

- the first, a list of arguments for which we want to print the results;
- the second, a function which should be invoked as many times as the number of values that are collected inside the first parameter.

Note: we've also defined a function named **poly()** – this is the function whose values we're going to print. The calculation the function performs isn't very sophisticated – it's the polynomial (hence its name) of the form:

f(x) = 2x2 - 4x + 2

The name of the function is then passed to the **print_function()** along with a set of five different arguments – the set is built with a list comprehension clause.

Output:

```
f(-2)=18
f(-1)=8
f(0)=2
f(1)=0
f(2)=2
```

Can we avoid defining the **poly()** function, as we're not going to use it more than once? Yes, we can – this is the benefit a lambda can bring:

```
def print_function(args, fun):
    for x in args:
        print('f(', x, ')=', fun(x), sep='')


print_function([x for x in range(-2, 3)], lambda x: 2 * x ** 2 - 4 * x + 2)
```

The print_function() has remained exactly the same, but there is no **poly()** function. We don't need it anymore, as the polynomial is now directly inside the **print_function()** invocation in the form of a **lambda** defined in the following way: `lambda x: 2 * x**2 - 4 * x + 2`

Lambdas and the map() function

`map(function, list)`

It takes two arguments:

- a function;
- a list.

The above description is extremely simplified, as:

- 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.

The **map()** function applies the function passed by its first argument to all its second argument's elements, and returns an iterator delivering all subsequent function results.

You can use the resulting iterator in a **loop**, or convert it into a list using the **list()** function.

```
# Create a list of integers from 0 to 4 using a list comprehension.
list_1 = [x for x in range(5)]

# map() applies the lambda function to EACH element of list_1.
# lambda x: 2 ** x
#   - is an anonymous function
#   - takes one argument (x)
#   - returns 2 raised to the power of x
# map() returns a map object, so list() is used to store the results.
list_2 = list(map(lambda x: 2 ** x, list_1))

# Print the list produced by map().
print(list_2)

# map() is used again with a different lambda.
# lambda x: x * x
#   - squares each element from list_2
# The map object is iterated directly by the for loop.
for x in map(lambda x: x * x, list_2):
   print(x, end=' ')

# Print a newline after the loop output.
print()
```

– – Continued – –

Lambdas and the filter() function

It expects the same kind of arguments as **map()**, but does something different – it filters its second argument while being guided by directions flowing from the function specified as the first argument

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

```
from random import seed, randint

# Initialize the random number generator.
# Calling seed() with no arguments seeds from system entropy,
# making each run produce different values.
seed()

# Create a list of 5 random integers between -10 and 10 (inclusive).
# This is the data set that will be processed by filter().
data = [randint(-10, 10) for x in range(5)]

# filter() applies the lambda function to EACH element of 'data'.
# The lambda must return True or False.
# lambda x: x > 0 and x % 2 == 0
#   - keeps only numbers that are:
#       • positive
#       • even
# filter() returns an iterator, so list() is used to store the result.
filtered = list(filter(lambda x: x > 0 and x % 2 == 0, data))

# Print the original data set.
print(data)

# Print only the values that passed the lambda condition.
print(filtered)
```

ChatGPT below:

What this lesson is emphasizing (don't miss this)

- `lambda` defines a **boolean test**, not a transformation
- `filter()` **keeps or discards elements** based on that test
- The lambda must return **True** **or** **False**
- `filter()` returns an **iterator**, not a list
- Lambdas are ideal when the condition is **simple and disposable**

Core mental model

- `map()` → *change* *values*
- `filter()` → *remove* *values*
- `lambda` → *temporary* *rule*

A brief look at closures

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**

```
def outer(par):
    loc = par

var = 1
outer(var)

print(par)
print(loc)
```

The last two lines will cause a **NameError** exception – neither par nor loc is accessible outside the function. Both the variables exist when and only when the **outer()** function is being executed.

```
def outer(par):
    loc = par
    def inner():
        return loc
    return inner

var = 1
fun = outer(var)
print(fun())
```

There is a brand new element in it – a function **inner** inside another function outer.

Just like any other function except for the fact that **inner()** may be invoked only from within **outer()**. We can say that **inner()** is **outer()**'s private tool – no other part of the code can access it.

Look carefully:

- the **inner()** function returns the value of the variable accessible inside its scope, as **inner()** can use any of the entities at the disposal of **outer()**
- the **outer()** function returns the **inner()** function itself; more precisely, it returns a copy of the **inner()** function, the one which was frozen at the moment of **outer()**'s invocation; the frozen function contains its full environment, including the state of all local variables, which also means that the value of loc is successfully retained, although **outer() ceased to exist a long time ago**.

The function returned during the **outer()** invocation is a **closure**.

- A closure has to be invoked in exactly the same way in which it has been declared.

```
def make_closure(par):
    loc = par

    def power(p):
        return p ** loc

    return power

fsqr = make_closure(2)
fcub = make_closure(3)

for i in range(5):
    print(i, fsqr(i), fcub(i))
```

This means that the closure not only makes use of the frozen environment, but **it can also modify its behavior by using values taken from the outside**.

Also, you can create as many closures as you want using one and the same piece of code. This is done with a function named **make_closure()**. Note:

- The first closure obtained from **make_closure()** defines a tool squaring its argument;
- The second one is designed to cube the argument.

Summmary

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 an 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)
```

Output: 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)
```

Output: 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))
```

Output: 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)
```

Output: ['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)
```

Output: ['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'))
```

Output: Monty Python

4.2 - Files (File streams, file processing, diagnosing stream problems)

File names

This chapter was the absolute worst! ChatGPT saved us.

File name differences by OS

- **Windows and Unix/Linux use different file naming rules**
- **Unix/Linux is case-sensitive**
    - `File.txt` ≠ `file.txt`
- **Windows stores the case of letters used in the file name, but can't distinguish between their cases**
    - `File.txt` == `file.txt`

Path separators (the important part for Python)

- **Windows:** `\`
- **Unix/Linux:** `/`

This matters in Python because:

- `\` is an **escape character** in strings (`\n`, `\t`, etc.)

```
# Unix/Linux
name = "/dir/file"

# Windows (escaped backslashes)
name = "\\dir\\file"
```

The Python-friendly solution

Python accepts **forward slashes on all platforms**:

```
name = "/dir/file"
name = "c:/dir/file"
```

✔ Works on Windows and Unix/Linux
✔ Preferred for cross-platform code

Files are accessed through streams (not directly)

- Programs **do not work with files directly**
- They work with **streams** (also called handles)
- The OS manages the actual file access

File lifecycle (always in this order)

1. **Open** the file (bind stream → file)
2. **Operate** on the stream
3. **Close** the file (unbind stream)

If opening fails, common reasons include:

- File does not exist
- Permission denied
- Too many files already open

✔ Well-written programs **detect and handle open failures**

File streams

Summarized by ChatGPT:

When a file is opened, Python does not access the file directly. Instead, it creates a **stream object** that represents the connection between the program and the file.

- Opening a stream requires specifying an **open mode**, which defines **how the stream may be used**.

If the file opens successfully, the program is restricted to **only those operations permitted by the selected mode**.

Stream operations

There are **two fundamental operations** that can be performed on a stream:

- **Read**: data is transferred from the file into program memory (e.g., variables)
- **Write**: data is transferred from program memory into the file

Open modes

Python supports **three basic stream modes**:

- **Read mode**: allows reading only; writing raises `UnsupportedOperation`
- **Write mode**: allows writing only; reading raises `UnsupportedOperation`
- **Update mode**: allows both reading and writing

The `UnsupportedOperation` exception originates from the **`io` module** and inherits from both `OSError` and `ValueError`.

Stream position (file pointer)

Each stream maintains an internal **current file position**:

- **Reading** moves the position **forward by the number of bytes read**
- **Writing** moves the position **forward by the number of bytes written**

All read and write operations occur **relative to this position**.

File handles

A 30-minute in-depth overview of how Python handles different file types:
https://www.youtube.com/watch?v=pOw7BEuvw_k&t=12s

Files and file handle objects

- Python does **not** work with files directly.
  Every opened file is represented by a **file handle object** created by `open()`.
  The object:
  - Exists **only between `open()` and `close()`**
  - Controls **how the stream is accessed**
  - Enforces the **allowed operations** based on open mode
- File handle objects are **never created directly** using constructors.

– – Continued – –

File handle lifecycle

1. `open()`
    - Analyzes arguments
    - Creates the appropriate file handle object
2. File operations
    - Reading / writing through the stream
3. `close()`
    - Breaks the connection to the file
    - Destroys the handle object

Stream categories (by data type)

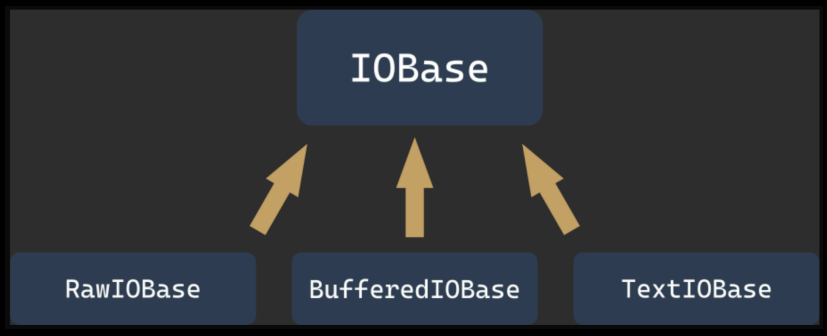
All streams fall into **two major categories**:

Text streams

- Contain **human-readable characters**
- Organized into **lines**
- Processed:
    - Character by character
    - Line by line
- Represented by **TextIOBase**

Binary streams

- Contain **raw bytes**
- No concept of lines
- Used for:
    - **Executables**
    - **Images**
    - **Audio/video**
    - **Databases**
- Processed:
    - Byte by byte
    - Block by block
- Represented by **BufferedIOBase**



Newline portability problem

- **Unix/Linux** uses:
    - Line ends are marked by a character named **LF** [ASCII code 10]
        - Designated in Python programs as `\n`
- **Windows** uses:
    - The end of a line ends in **CR** and **LF** [ASCII codes 13 and 10]
    - Coded as `\r\n`
- Hard-coding line endings causes **non-portable programs**

Python solves the **portability** issue at the **stream class level**:

Text mode behavior

- **Default when opening text files**
- The process is called **translation of newline characters**:
    - Windows `\r\n` ↔ Python `\n`
- Program behaves as if:
    - Only `\n` exists
- Translation is **transparent** to the developer

Binary mode behavior

- No translation at all
- Data is read/written **exactly as stored**
- Required for non-text files

Opening the streams

```
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
- 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.

Note: **the mode and encoding arguments may be omitted** – their default values are assumed then.

- The default opening mode is read in text mode, while the default encoding depends on the platform used.

Opening Stream 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.

– – Continued – –

Selecting text and binary modes

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

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

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

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

Finally, the successful opening of a file will set the current file position before the first byte of the file if the mode is not **a**

- and after the last byte of the file if the mode is set to **a**.

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

Opening the stream for the first time

We want to develop a program that reads the contents of the text file named: C:\Users\User\Desktop\file.txt

```
try:
    stream = open("C:\Users\User\Desktopile.txt", "rt")
    # Processing goes here.
    stream.close()
except Exception as exc:
    print("Cannot open the file:", exc)
```

- we open the **try-except** block as we want to handle runtime errors softly;
- we use the **open()** function to try to open the specified file (note the way we've specified the file name)
- the open mode is defined as text to read (as text is the default setting, we can skip the **t** in the mode string)
- if we're successful, we get an object from the **open()** function and we assign it to the stream variable;
- if **open()** fails, we handle the exception by printing the full error information
    - It's definitely good to know what exactly happened

– – Continued – –

Pre-opened streams

```
import sys
```

There are 3 exceptions to the open() requirement for stream operations.

Utilizing the import sys module allows streams to be used explicitly

The names of these streams are: **sys.stdin**, **sys.stdout**, and **sys.stderr**.

- **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 (soon to come)
  - The separation of **stdout** from the **stderr** 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.**

Closing streams

The last operation performed on a stream (this doesn't include the stdin, stdout, and stderr streams, which don't require it) should be closing.

That action is performed by a method invoked from within the open stream object: **stream.close()**

- the function expects exactly no arguments; **the stream doesn't need to be opened**
- the function returns nothing, but raises an **IOError exception** in case of error;
- Most developers believe that the **close()** function always succeeds and thus there is no need to check if it's done its task properly.

  This belief is only partly justified. If the stream was opened for writing and then a series of write operations were performed, it may happen that the data sent to the stream has not been transferred to the physical device yet (due to a mechanism called caching or buffering).

  Since the closing of the stream forces the buffers to flush them, it may be that the flushes fail and therefore the **close() fails too**.

We have already mentioned failures caused by functions operating with streams, but we haven't said a word about how exactly we can identify the cause of the failure.

The possibility of making a diagnosis exists and is provided by one of streams' **exception components**.

Diagnosing stream problems

```
try:
    # Some stream operations.
except IOError as exc:
    print(exc.errno)
```

The **IOError** object is equipped with a property named **errno** (the name comes from the phrase error number)

The value of the errno attribute can be compared with one of the **predefined symbolic constants defined in the errno module**.

A few 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.

The complete list is much longer and also includes some error codes not related to the stream processing

strerror() is a function that can dramatically simplify the error handling code:

```
from os import strerror

try:
   s = open("c:/users/user/Desktop/file.txt", "rt")
   # Actual processing goes here.
   s.close()
except Exception as exc:
   print("The file could not be opened:", strerror(exc.errno))
```

Its role is simple: **you give an error number and get a string describing the meaning of the error**.

- A non-existent error code in the function will raise a ValueError exception.

Summary

File Names

- Use / in paths for portability
- Backslashes must be escaped in Python
- Files are accessed via streams
- Always open → use → close
- Opening a file can fail — handle it

File Streams
A stream object is responsible for:

- Managing access permissions (via open mode)
- Tracking the current file position
- Coordinating data transfer between memory and the physical file

Understanding streams is essential before learning how to manipulate files programmatically.

File Handles

- Files are accessed via **file handle objects**
- Handles are created by `open()` and destroyed by `close()`
- Streams are **text or binary**
- Text mode ensures **cross-platform portability**
- Binary mode preserves **raw data**
- Newline translation happens **automatically** in text mode

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.

4.3 - Working with real files

Processing text files

2 main rules regarding text documents:

1. Our understanding of a text file is very strict, it's a plain text file – it may contain only text, without any additional decorations (formatting, different fonts, etc)..
2. Avoid creating the file using any advanced text processor like MS Word, LibreOffice Writer, or something like this.
    a. Use the very basics your OS offers: Notepad, vim, gedit, etc.

If your text files contain some national characters not covered by the standard **ASCII charset**, your **open()** function invocation may require an argument denoting specific text encoding.

```
stream = open('file.txt', 'rt', encoding='utf-8')
```

the encoding argument has to be set to a value which is a string representing proper text encoding (UTF-8, here)

When **read()** is applied to a text file, the function is able to:

- read a desired number of characters (including just one) from the file
- return them as a string; read all the file contents, and return them as a string
- if there is nothing more in the file for Python to read, the function returns an empty string.

Imagine a file named **text.txt**→

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
```

```
from os import strerror

try:
    counter = 0
    stream = open('text.txt', "rt")
    char = stream.read(1)
    while char != '':
        print(char, end='')
        counter += 1
        char = stream.read(1)
    stream.close()
    print("\n\nCharacters in file:", counter)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

- use the try-except mechanism and open the file of the predetermined name (text.txt in our case)
- try to read the very first character from the file (**char = stream.read(1)**)
- if you succeed, output the character
    - Success is proven by a positive result of the **while** condition check
    - note the **end=** argument – it's important! You don't want to skip to a new line after every character!);
- update the (**counter**), too
- try to read the next character, and the process repeats.

Warning: **Reading a terabyte-long file using this method may corrupt your OS.**

– – Continued – –

readline()

If you want to treat the file's contents as a set of lines, not a bunch of characters, the **readline()** method will help you with that.

- The method tries to read a complete line of text from the file, and **returns it as a string** in the case of success.
- Otherwise, it **returns an empty string**.

```
from os import strerror

try:
   character_counter = line_counter = 0
   stream = open('text.txt', 'rt')
   line = stream.readline()
   while line != '':
       line_counter += 1
       for char in line:
           print(char, end='')
           character_counter += 1
       line = stream.readline()
   stream.close()
   print("\n\nCharacters in file:", character_counter)
   print("Lines in file:     ", line_counter)
except IOError as e:
   print("I/O error occurred:", strerror(e.errno))
```

readlines()

Another method, which treats text file as a set of lines, not characters, is

- when invoked without arguments, tries to read all the file contents, and returns a list of strings, one element per file line.
- User has the ability to limit readlines() to a specified number of bytes

```
stream = open("text.txt")
print(stream.readlines(20))
print(stream.readlines(20))
print(stream.readlines(20))
print(stream.readlines(20))
stream.close()
```

- The maximum accepted input buffer size is passed to the method as its argument.
- **readlines()** can process a file's contents more effectively than **readline()**, as it may need to be invoked fewer times.
- Note: when there is nothing to read from the file, the method returns an empty list.
    - **Use it to detect the end of the file.**

```
import strerror      # Converts OS error numbers into readable messages

try:
   ccnt = lcnt = 0          # Initialize counters for characters and lines

   # Iterating over the stream reads the file LINE BY LINE automatically
   for line in open('text.txt', 'rt'):
       lcnt += 1            # Count each line read from the file


       for ch in line:       # Iterate through each character in the current line
           print(ch, end='')    # Print characters exactly as stored
           ccnt += 1            # Count each character

   # Display the final counts after processing the entire file
   print("\n\nCharacters in file:", ccnt)
   print("Lines in file:     ", lcnt)
except IOError as e:         # Handle I/O-related errors (e.g., missing file, access denied)
   print("I/O error occurred: ", strerror(e.errno))
```

Dealing with text files: write()

Text files are written using the **write()** method.

write() accepts one argument: **a string**.

The file must be opened in a write-capable mode (w, a, or r+).

write() does not add a newline automatically.

- Add **\n** yourself to create multiple lines.

Opening a file in **w** mode:

- Creates the file if it does not exist
- Overwrites the file if it already exists

A simple code that creates a file named newtext.txt:

```
from os import strerror      # Converts OS error codes into readable messages

try:          # Open a text file in write mode ('wt')
   file = open('newtext.txt', 'wt')  # A new file (newtext.txt) is created.
   for i in range(10):
       s = "line #" + str(i + 1) + "\n"      # Build a single line of text, add \n
       for char in s:       # Write the line character by character
           file.write(char)
   file.close()     # Close the file to flush buffers and release the handle
except IOError as e:
   print("I/O error occurred: ", strerror(e.errno))
```

Note: you can use the same method to write to the stderr stream, but don't try to open it, as it's always open implicitly.

For example, if you want to send a message string to stderr to distinguish it from normal program output, it may look like this:

```
import sys
sys.stderr.write("Error message")
```

What is a bytearray

Amorphous data is data with **no inherent structure at the point of use**.

- It is simply a sequence of bytes
- The meaning of the data may exist (e.g., an image), but the program does not interpret it

Such data cannot be stored as:

- Strings (text-oriented)
- Lists (not byte-focused or memory-efficient)

Python provides specialized containers for raw binary data.

- One of them is bytearray

bytearray:

- Is a mutable sequence of bytes
- Stores values in the range 0–255 inclusive
- Is designed for binary file processing
- Allows direct byte-level manipulation

bytearray must be explicitly created using a constructor

- Commonly used when working with:
  - Images
  - Executables
  - Binary file formats
  - Network data

```
data = bytearray(10)
```

Such an invocation creates a bytearray object **able to store ten bytes** and fills the array with zeroes.

To write a **bytearray** to a file:

```
from os import strerror      # Converts OS error codes into readable messages

data = bytearray(10)         # Create a bytearray of length 10, All elements are initialized to 0

for i in range(len(data)):
   data[i] = 10 + i          # Fill the bytearray with values from 10 to 19

try:
          bf = open('file.bin', 'wb')      # Open a file in binary write mode ('wb'),
                                           # Binary mode disables any data translation

   bf.write(data)       # Write the entire bytearray to the file
   bf.close()       # Close the binary stream and release the file handle
except IOError as e:
   print("I/O error occurred:", strerror(e.errno))

# Your code that reads bytes from the stream should go here.
```

- first, we initialize the **bytearray** with subsequent values starting from 10;
  - if you want the file's contents to be clearly readable, replace 10 with something like ord('a') – this will produce bytes containing values corresponding to the alphabetical part of the ASCII code
- then, we create the file using the **open()** function – the only difference compared to the previous variants is the open mode containing the **b** flag;
- the **write()** method takes its argument (**bytearray**) and sends it to the file;
- The stream is then closed in a routine way.

– – Continued – –

How to read bytes from a stream

Reading from a binary file requires the 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.

- the method returns the number of successfully read bytes;
- the method tries to fill the whole space available inside its argument
  - if there are more data in the file than space in the argument, the read operation will stop before the end of the file
  - otherwise, the method's result may indicate that the byte array has only been filled fragmentarily
    - the result will show you that, too, and the part of the array not being used by the newly read contents remains untouched

```
from os import strerror

data = bytearray(10)

try:
   binary_file = open('file.bin', 'rb')
   binary_file.readinto(data)
   binary_file.close()

   for b in data:
       print(hex(b), end=' ')
except IOError as e:
   print("I/O error occurred:", strerror(e.errno))
```

- first, we open the file (the one you created using the previous code) with the mode described as **rb**;
- then, we read its contents into the byte array named data, of size **ten bytes**;
- finally, we **print the byte array contents**

rt

An alternative way to read the contents of a binary file is to use **read()**

- When called without arguments, it reads the entire file into memory.
- If the **read()** method is invoked with an argument, it specifies the maximum number of bytes to be read.
- The data is returned as a bytes object.

bytes vs bytearray

- bytes
  - **Immutable**
  - Suitable for read-only binary data
- bytearray
  - **Mutable**
  - Suitable for modifying binary data

- A bytearray can be directly created from a bytes object.

```
try:
   binary_file = open('file.bin', 'rb')
   data = bytearray(binary_file.read(5))
   binary_file.close()

   for b in data:
       print(hex(b), end=' ')

except IOError as e:
   print("I/O error occurred:", strerror(e.errno))
```

Copying files - a simple and functional tool

```
from os import strerror  # Converts OS error codes into readable messages

# Prompt for the source file name
srcname = input("Enter the source file name: ")
try:
   # Open the source file in binary read mode
   src = open(srcname, 'rb')
except IOError as e:
   # Handle errors such as missing file or access denial
   print("Cannot open the source file: ", strerror(e.errno))
   exit(e.errno)

# Prompt for the destination file name
dstname = input("Enter the destination file name: ")
try:
   # Open the destination file in binary write mode
   # If the file exists, it will be overwritten
   dst = open(dstname, 'wb')
except Exception as e:
   # Handle errors during destination file creation
   print("Cannot create the destination file: ", strerror(e.errno))
   src.close()  # Ensure source file is closed before exiting
   exit(e.errno)

# Allocate a fixed-size buffer for copying data
# 65,536 bytes (64 KB) is a common and efficient block size
buffer = bytearray(65536)

# Counter to track the total number of bytes copied
total = 0

try:
   # Read the first block of data into the buffer
   # readinto() returns the number of bytes read
   readin = src.readinto(buffer)

   # Continue copying until end-of-file (readin == 0)
   while readin > 0:
       # Write only the valid portion of the buffer
       written = dst.write(buffer[:readin])

       # Accumulate the number of bytes written
       total += written

       # Read the next block from the source file
       readin = src.readinto(buffer)

except IOError as e:
   # Handle I/O errors that occur during copying
   print("Cannot create the destination file: ", strerror(e.errno))
   exit(e.errno)

# Report the result of the copy operation
print(total, 'byte(s) succesfully written')

# Close both files to release system resources
src.close()
dst.close()
```

lines 3 through 11: ask the user for the name of the file to copy, and try to open it to read; terminate the program execution if the open fails; note: use the exit() function to stop program execution and to pass the completion code to the OS; any completion code other than 0 says that the program has encountered some problems; use the errno value to specify the nature of the issue;

lines 14 through 35: repeat nearly the same action, but this time for the output file;

line 27: prepare a piece of memory for transferring data from the source file to the target one; such a transfer area is often called a buffer, hence the name of the variable; the size of the buffer is arbitrary

- in this case, we've decided to use 64 kilobytes;
- technically, a larger buffer is faster at copying items, as a larger buffer means fewer I/O operations;
- there is always a limit, the crossing of which renders no further improvements

line 30: count the bytes copied – this is the counter and its initial value

line 35: try to fill the buffer for the very first time

line 38: as long as you get a non-zero number of bytes, repeat the same actions

line 40: write the buffer's contents to the output file (note: we've used a slice to limit the number of bytes being written, as write() always prefers to write the whole buffer)

line 43: update the counter

line 46: read the next file chunk

lines 54 through 58: some final cleaning – the job is done.

Summary

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 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.

4.4 - The os module - interacting with the OS

Intro to the OS module

The os module allows Python to interact with Unix/Linux and Windows systems.

- Get information about the OS
- Manage processes
- Operate on I/O streams using file descriptors.

Getting information about the OS

```
import os
print(os.uname())
```

The **uname()** function returns an object containing 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.

Output:

```
posix.uname_result(sysname='Linux', nodename='192d19f04766', release='4.4.0-164-generic',
version='#192-Ubuntu SMP Fri Sep 13 12:02:50 UTC 2019', machine='x86_64')
```

Unfortunately, **the uname function only works on some Unix systems**. If you use Windows, you can use the uname function in the platform module, which returns a similar result.

The os module allows you to quickly distinguish the operating system using the name attribute, which supports one of the following names:

- **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.

For Ubuntu 16.04.6 LTS, the name attribute returns the name posix:

```
import os
print(os.name)
```

NOTE: On Unix systems, there's a command called **uname** that returns the same information if you run it with the **-a** option as the **uname function**.

Creating directories in Python

The os module provides a function called **mkdir**, which, like the mkdir command in Unix and Windows, **allows you to create a directory**. The **mkdir** function **requires a path that can be relative or absolute**. Let's recall what both paths look like in practice:

- **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.

Look at the code here. It shows an example of how to create the **my_first_directory** directory using a relative path. This is the simplest variant of the relative path, which consists of passing only the directory name.

```
import os

os.mkdir("my_first_directory")
print(os.listdir())
```

The **mkdir** function creates a directory in the specified path.

- Note that **running the program twice will raise a FileExistsError**.
- This means that we cannot create a directory if it already exists.

In addition to the path argument, the **mkdir** function can optionally take the **mode argument**, which specifies directory permissions.

- **On some systems the mode argument is ignored**.

To change the directory permissions, we recommend the **chmod** function, which works similarly to the **chmod** command on **Unix** systems. You can find more information about it in the documentation.

In the above example, another function provided by the **os** module named **listdir** is used. 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 (as in the example above).
- It's important that the result of the listdir function **omits the entries** '**.**' and '**..**', which are displayed, e.g., when using the **ls -a** command on **Unix systems**.

NOTE: In both Windows and Unix, there's a command called **mkdir**, which requires a directory path. The equivalent of the above code that creates the **my_first_directory** directory is the **mkdir my_first_directory** command.

Recursive directory creation

The **makedirs function enables recursive directory creation**

- Meaning all directories in the path will be created.

```
import os

os.makedirs("my_first_directory/my_second_directory")
os.chdir("my_first_directory")
print(os.listdir())
```

Output: ['my_second_directory']

The code creates two directories.

- First, it creates **my_first_directory in the current working directory**.
- Next, it creates **my_second_directory** as a **subdirectory inside my_first_directory**.

You do not need to change into **my_first_directory** to create **my_second_directory**, because the **makedirs** function can create **nested directories** in one call.

- In the example, the code changes into my_first_directory only to demonstrate that my_second_directory was created inside it.

To move between directories, you can use the **chdir** function.

- This function changes the current working directory to the path you specify.
- It accepts either a relative path or an absolute path.
- In this example, it is given the name of the first directory.

NOTE: The equivalent of the **makedirs** function on **Unix** systems is the **mkdir command with the -p flag**
- while in Windows, simply the **mkdir** command with the path:

Unix-like sytems:
```
mkdir -p my_first_directory/my_second_directory
```

Windows:
```
mkdir my_first_directory/my_second_directory
```

Where am I now?

getcwd is the **os module's** "get working directory" **function**.

```
import os

os.makedirs("my_first_directory/my_second_directory")
os.chdir("my_first_directory")
print(os.getcwd())
os.chdir("my_second_directory")
print(os.getcwd())
```

Output:
```
.../my_first_directory
.../my_first_directory/my_second_directory
```

In the example, we create the **my_first_directory directory**, and the **my_second_directory directory inside it**.

In the next step, we change the current working directory to the **my_first_directory directory**, and then display the current working directory (first line of the result).

Next, we go to the **my_second_directory directory** and again display the current working directory (second line of the result).
- As you can see, the **getcwd** function **returns the absolute path to the directories**.

NOTE: On Unix-like systems, **the equivalent of the getcwd function is the pwd command**, which prints the name of the current working directory.

Deleting directories in Python

```
import os

os.mkdir("my_first_directory")
print(os.listdir())
os.rmdir("my_first_directory")
print(os.listdir())
```

The **rmdir function** is utilized to delete a **single directory** or a **directory with its subdirectories**.

- First, the **my_first_directory directory** is created, and then it's removed using the **rmdir function**.
  - The **listdir function** is used as proof that the directory has been removed successfully.
    - In this case, it returns an empty list.
  - When deleting a directory, make sure it exists and is empty, or **an exception will be raised**.
- 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:

```
import os

os.makedirs("my_first_directory/my_second_directory")
os.removedirs("my_first_directory/my_second_directory")
print(os.listdir())
```

As with the rmdir function, if one of the directories doesn't exist or isn't empty, an **exception** will be raised.

NOTE: In both Windows and Unix, there's a command called **rmdir**, which, just like the rmdir function, removes directories. What's more, both systems have commands to delete a directory and its contents.

- In Unix, this is the **rm** command with the **-r flag**.

The system() function

```
import os

returned_value = os.system("mkdir my_first_directory")
print(returned_value)
```

The **system function** is available in both Windows and Unix. Depending on the system, it returns a different result.

- In Windows, it returns the value returned by the shell after running the command given,
- In Unix, it returns the exit status of the process.

In our case, we receive **exit status 0**, which indicates **success on Unix systems**.
This means that the **my_first_directory directory has been created**.

Summary

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**

- 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 the shell after running the command given, while on Unix it returns the exit status of the process.

4.5 - The datetime module

Here's the short, no-fluff distinction to remember going through this module:

- **time class** (**datetime.time**)
  - Represents a time of day only (hours, minutes, seconds)
  - No date, no timezone logic by default

- **time module** (**import time**)
  - Low-level system time utilities
  - Sleeping, timestamps, performance counters

- **datetime module** (**import datetime**)
  - High-level date and time types
  - Provides date, time, datetime, timedelta

- **time() function** (**time.time()**)
  - Returns current time as seconds since the Unix epoch
  - Used for timestamps and measuring elapsed time

Also -

When trying to work with **strftime** and/or **strptime**:

The core difference (think "direction of travel")

- **strptime** = string ➜ time object (PARSE input)
  You use it when you receive a date/time as text (log file, CSV column, user input, API payload) and you need a datetime/date/time object you can compare, sort, add/subtract, etc. In datetime, it's a class method like datetime.strptime("2025-12-23", "%Y-%m-%d").

- **strftime** = time object ➜ string (FORMAT output)
  You use it when you have a date/time object and you need a string representation (UI display, filenames, reports, logs). Example: dt.strftime("%Y-%m-%d").

- Both use the same family of % format codes (e.g., %Y %m %d %H %M %S).

Quick mnemonic (because humans are fallible):

- strPtime = Parse (string → datetime)
- strFtime = Format (datetime → string)

Ideal use-cases :

- Use **strptime** when:
  - Verify that a date string conforms to a predefined format..
  - You need to do math (add days, compare timestamps, sort chronologically).
  - You're normalizing messy inputs into a consistent internal representation.

- Use **strftime** when:
  - You're producing human-readable output ("Dec 23, 2025 18:04").
  - You're generating stable machine-friendly strings (filenames like backup_2025-12-23_1804.log).
  - You need locale-style output via codes like %x / %X (be careful: output varies by locale).

Two gotchas that bite people in labs (and in prod)

- **2-digit years**: parsing **%y** follows POSIX rules (69–99 → 1969–1999, 00–68 → 2000–2068). If you don't want surprises, **prefer %Y**.

- Time zones: **strptime**/**strftime** don't magically "understand" time zones unless your format includes them and you're using aware datetimes properly.
    - When in doubt, store/compute in UTC internally, format for humans at the edges.

Store & compute in UTC (internal, boring, correct)

```
from datetime import datetime, timezone

# Internal timestamp (UTC)
event_utc = datetime.now(timezone.utc)

# Safe computations
expires_utc = event_utc.replace(hour=23, minute=59)

print("Stored (UTC):", event_utc.isoformat())
```

Format for humans at the edges (local, friendly)

```
from zoneinfo import ZoneInfo

# Convert ONLY when displaying
event_local = event_utc.astimezone(ZoneInfo("America/Los_Angeles"))

print("Displayed (Local):", event_local.strftime("%Y-%m-%d %I:%M %p %Z"))
```

Intro to the datetime module

The many uses of date and time:

- **event logging** — thanks to the knowledge of date and time, we are able to determine when exactly a critical error occurs in our application. When creating logs, you can specify the date and time format;
- **tracking changes in the database** — sometimes it's necessary to store information about when a record was created or modified. The datetime module will be perfect for this case;
- data validation — you'll soon learn how to read the current date and time in Python. Knowing the current date and time, you'll be able to validate various types of data, e.g., whether a discount coupon entered by a user in our application is still valid;
- **storing important information** — can you imagine bank transfers without storing the information of when they were made? The date and time of certain actions must be preserved, and we must deal with it.

Date and time are used in almost every area of our lives, so it's important to familiarize yourself with the Python datetime module. Are you ready for a new dose of knowledge?

Getting the current local date and creating date objects

```
from datetime import date

today = date.today()

print("Today:", today)
print("Year:", today.year)
print("Month:", today.month)
print("Day:", today.day)
```

The today method returns a date object representing the current local date. Note that the date object has three attributes: year, month, and day.

Be careful, because these attributes are read-only. To create a date object, you must pass the year, month, and day parameters as follows:

```
from datetime import date

my_date = date(2019, 11, 4)
print(my_date)
```

When creating a date object, keep the following restrictions in mind:

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.

Creating a date object from a timestamp

Unix epoch: when the counting of time began on a Unix system, since **January 1, 1970, 00:00:00 (UTC)**

- Generally 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()
print("Timestamp:", timestamp)

d = date.fromtimestamp(timestamp)
print("Date:", d)
```

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

Creating a date object using the ISO format

```
from datetime import date

d = date.fromisoformat("2019-11-04")
print(d)
```

(ChatGPT had to generate this code because the lesson's author somehow omitted the original code. Wild. Did anyone QC this??)

date.fromisoformat():

- Accepts only "**YYYY-MM-DD**"
- Requires zero-padded month/day
- Returns a date object
- Available since Python 3.7

ISO 8601 is a string-based date format (not timestamps)

The replace() method

```
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
```

The year, month, and day parameters are optional.

- You may pass 1-3 parameter(s) to the replace method, e.g., year, or all three as in the example.

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

What day of the week is it?

```
from datetime import date

d = date(2019, 11, 4)
print(d.weekday())
```

Output: 0

weekday() returns the day of the week as an integer, where **0** is **Monday** and **6** is **Sunday**.

The date class has a similar method called isoweekday, which also returns the day of the week as an integer, but 1 is Monday, and 7 is Sunday:

```
from datetime import date

d = date(2019, 11, 4)
print(d.isoweekday())
```

The integer returned by the **isodayweek method** follows the **ISO 85601 specification**.

Creating time objects

```
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 **datetime module** provides the **date class** for dates and the **time** class for **time values**.

The **time class** constructor accepts the following optional parameters:

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).

The **tzinfo** parameter is associated with time zones, while **fold** is associated with wall times.

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)
```

```
I'm very tired. I have to take a nap. See you later.
I slept well! I feel great!
```

The sleep() function suspends program execution for the given number of seconds.

- In this example, it's a **5 second nap**.
- the **sleep function** accepts only an **integer** or a **floating point** number.

The ctime() function

```
import time

timestamp = 1572879180
print(time.ctime(timestamp))
```

```
Mon Nov  4 14:53:00 2019
```

The time module provides a function called **ctime**, which **converts the time in seconds** since January 1, 1970 (Unix epoch) **to a string**.

The **ctime function** returns a string for the passed **timestamp** (1572879180).

It's also possible to call the **ctime function** without specifying the time in seconds.

In this case, **the current time will be returned**:

```
import time
print(time.ctime())
```

The gmtime() and localtime() functions

Some of the functions available in the time module require knowledge of the **struct_time class**:

```
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 **struct_time class** also **allows access to values using indexes**.

- Index 0 returns the value in **tm_year**
- **8** returns the value in **tm_isdst**.
- The **exceptions** are tm_zone and **tm_gmoff**, **they cannot be accessed using indexes**

```
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)
```

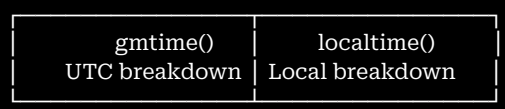
gmtime is (**Greenwich Mean Time**) & never applies **Daylight Saving Time**

The example shows two functions that convert the elapsed time from the **Unix epoch** to the **struct_time object**.

The difference between them is that the **gmtime function** returns the **struct_time object** in UTC,
the **localtime function** returns local time.

For the **gmtime function**, the **tm_isdst** attribute is always **0**.

[System clock]
↓
[Unix timestamp] ← same everywhere
↓



The asctime() and mktime() functions

```
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
```

The first of the 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.

mktime tuple values:

```
2019 => tm_year
11 => tm_mon
4 => tm_mday
14 => tm_hour
53 => tm_min
0 => tm_sec
0 => tm_wday
308 => tm_yday
0 => tm_isdst
```

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**.

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).

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**

```
from datetime import datetime

# Create a datetime object:
# November 4, 2019 at 14:53:00
dt = datetime(2019, 11, 4, 14, 53, 0)

# Print the full datetime object
print("Datetime:", dt)

# Extract and print the date part
print("Date:", dt.date())

# Extract and print the time part
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

```
from datetime import datetime

print("today:", datetime.today())
print("now:", datetime.now())
print("utcnow:", datetime.utcnow())
```

The datetime class has several methods that return the current date and time. These methods are:

- **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**, must be an object of the tzinfo subclass
- **utcnow()** — returns the current **UTC date and time with the tzinfo attribute set to None**.

Getting a timestamp

```
from datetime import datetime

dt = datetime(2020, 10, 4, 14, 55)
print("Timestamp:", dt.timestamp())
```

```
Timestamp: 1601823300.0
```

the timestamp method provided by the datetime class.

Food for Thought: In this instance, a **timestamp** is a **distance on a timeline**, not a structured date.

Date and time formatting

```
from datetime import date

d = date(2020, 1, 4)
print(d.strftime('%Y/%m/%d'))
```

```
2020/01/04
```

All **datetime module classes** presented so far have a method called **strftime**, which allows us to **return the date and time in the format we specify**.

A **directive** is a string consisting of the character **%** and a **lowercase or uppercase letter**.

- For example above, the directive **%Y** means the **year with the century as a decimal number**.

```
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 first of the formats used concerns only time.

- **%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
    - In our example
        - **%H** is replaced by **14**
        - **%M** by **53**
        - **%S** by **00.**

The second format used combines date and time directives.

- The directive **%Y** returns the **year without a century** as a zero-padded decimal number (example: **20**).
- The **%B** directive returns the **month as the locale's full name** (example: **November**).

Note: **You can find all available directives** here.
https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes

The strftime() function in the time module

```
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
2020/10/12 12:19:40
```

strftime in the time module differs slightly from the strftime methods in the classes provided by the datetime module because,

- in addition to the format argument
- it can also take a tuple or struct_time object.
- If you don't pass a tuple or struct_time object, the formatting will be done using the current local time.

Creating a format **looks the same** as **for the strftime methods** in the **datetime module**.

All time module directives: https://docs.python.org/3/library/time.html#time.strftime

The strptime() method

```
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
```

Unlike the strftime method, **strptime** creates a datetime object from a string representing a date and time.

if the format you specify doesn't match the date and time in the string, it'll **raise a ValueError**.

Note: In the **time module**, you can find a function called **strptime**, which parses a string representing a time to a **struct_time object**.

- Its use is **analogous** to the **strptime method** in the **datetime class**:

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

```
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=-1)
```

strftime or strptime?????

When trying to work with **strftime** and/or **strptime**:

The core difference (think “direction of travel”)

- **strptime** = string ➜ time object (PARSE input)
  You use it when you receive a date/time as text (log file, CSV column, user input, API payload) and you need a datetime/date/time object you can compare, sort, add/subtract, etc. In datetime, it’s a class method like datetime.strptime("2025-12-23", "%Y-%m-%d").

- **strftime** = time object ➜ string (FORMAT output)
  You use it when you have a date/time object and you need a string representation (UI display, filenames, reports, logs). Example: dt.strftime("%Y-%m-%d").

- Both use the same family of % format codes (e.g., %Y %m %d %H %M %S).

Quick mnemonic (because humans are fallible):

- strPtime = Parse (string → datetime)
- strFtime = Format (datetime → string)

Ideal use-cases :

- Use **strptime** when:
  - Verify that a date string conforms to a predefined format..
  - You need to do math (add days, compare timestamps, sort chronologically).
  - You're normalizing messy inputs into a consistent internal representation.

- Use **strftime** when:
  - You're producing human-readable output ("Dec 23, 2025 18:04").
  - You're generating stable machine-friendly strings (filenames like backup_2025-12-23_1804.log).
  - You need locale-style output via codes like %x / %X (be careful: output varies by locale).

Two gotchas that bite people in labs (and in prod)

- **2-digit years**: parsing **%y** follows POSIX rules (69–99 → 1969–1999, 00–68 → 2000–2068). If you don't want surprises, **prefer %Y**.

- Time zones: **strptime**/**strftime** don't magically "understand" time zones unless your format includes them and you're using aware datetimes properly.
  - When in doubt, store/compute in UTC internally, format for humans at the edges.

Store & compute in UTC (internal, boring, correct)

```
from datetime import datetime, timezone

# Internal timestamp (UTC)
event_utc = datetime.now(timezone.utc)

# Safe computations
expires_utc = event_utc.replace(hour=23, minute=59)

print("Stored (UTC):", event_utc.isoformat())
```

Format for humans at the edges (local, friendly)

```
from zoneinfo import ZoneInfo

# Convert ONLY when displaying
event_local = event_utc.astimezone(ZoneInfo("America/Los_Angeles"))

print("Displayed (Local):", event_local.strftime("%Y-%m-%d %I:%M %p %Z"))
```

– – Continued – –

Date and time operations

```
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
```

The resulting data from the above subtraction is a timedelta object, which represents the difference between two date or datetime values.

Creating timedelta objects

```
from datetime import timedelta

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

```
16 days, 3:00:00
```

The result of **16 days** is obtained by converting the weeks argument to days (**2 weeks = 14 days**) and adding the **days argument** (**2 days**).

The **timedelta object** only stores **days, seconds,** and **microseconds internally**.

Similarly, the hour argument is converted to minutes.

```
from datetime import timedelta

delta = timedelta(weeks=2, days=2, hours=3)
print("Days:", delta.days)
print("Seconds:", delta.seconds)
print("Microseconds:", delta.microseconds)
```

```
Days: 16
Seconds: 10800
Microseconds: 0
```

The result of 10800 is obtained by converting 3 hours into seconds.

In this way the **timedelta object** stores the arguments passed during its **creation**.

- Weeks are converted to days
- hours and minutes to seconds
- milliseconds to microseconds.

– – Continued – –

```python
from datetime import timedelta
from datetime import date
from datetime import datetime

# Create a timedelta object representing a span of time.
delta = timedelta(weeks=2, days=2, hours=2)

# Printing a timedelta shows its total duration in days, hours, minutes, and seconds.
print(delta)

# Timedelta objects support arithmetic operations, Multiplying a timedelta scales the duration.
delta2 = delta * 2

print(delta2)        # This is now twice the original time span.

# Timedelta objects can be added to date objects, The result is a new date, shifted forward in
time.
d = date(2019, 10, 4) + delta2
print(d)

# Timedelta objects can also be added to datetime objects, The result is a new datetime with both
date and time adjusted.
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
```

The **timedelta object** can be multiplied by an integer.

- We multiplied the object representing **16 days** and **2 hours** by **2**.
- Result: we receive a **timedelta object** representing **32 days and 4 hours**.

Note that both days and hours have been multiplied by **2**.

In the example, we've added the **timedelta object** to the **date** and **datetime objects**.

As a result of these operations, we receive date and datetime objects increased by days and hours stored in the timedelta object.

The presented multiplication operation allows you to quickly increase the value of the timedelta object, while multiplication can also help you get a date from the future.

Of course, the timedelta, date, and datetime classes support many more operations.

– – Continued – –

Summary

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).

4.6 - The calendar module

Introduction to the calendar module

Day of the week	Integer value	Constant
Monday	0	`calendar.MONDAY`
Tuesday	1	`calendar.TUESDAY`
Wednesday	2	`calendar.WEDNESDAY`
Thursday	3	`calendar.THURSDAY`
Friday	4	`calendar.FRIDAY`
Saturday	5	`calendar.SATURDAY`
Sunday	6	`calendar.SUNDAY`

The table above shows the representation of the days of the week in the **calendar module**.

Your first calendar

```
import calendar
print(calendar.calendar(2020))
```

The result displayed is **similar to the result of the cal command available in Unix**.

To change the default calendar formatting, you can use the following parameters:

- **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**)

The calendar function requires you to specify the year, while the other parameters responsible for formatting are optional. We encourage you to try these parameters yourself.

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**. Its use looks like this:

```
import calendar
calendar.prcal(2020)
```

Calendar for a specific month

```
import calendar
print(calendar.month(2020, 11))
```

The **month function** allows you to display a calendar for a specific month.

- Just specify the year and month

you can change the default formatting using the following parameters:

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

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

The setfirstweekday() function

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
calendar.prmonth(2020, 12)
```

The calendar function can be either the value 6 or the constant SUNDAY.

- Saying the day makes code easier to read.

The weekday() function

```
import calendar
print(calendar.weekday(2020, 12, 24))
```

Output: 3

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
```

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 three characters.

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

```
import calendar
print(calendar.isleap(2020))
print(calendar.leapdays(2010, 2021))   # Up to but not including 2021.
```

```
True
3
```

The calendar module provides two useful functions to check whether years are leap years.

The first one, called **isleap**, returns **True if the passed year is leap**, or **False otherwise**.

The second one, called **leapdays**, returns the **number of leap years** in the **given range of years**.

Classes for creating calendars

Additional classes for the **calendar module**.

- **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.

– – Calendar – –

Creating a calendar object

```
import calendar

c = calendar.Calendar(calendar.SUNDAY)

for weekday in c.iterweekdays():
   print(weekday, end=" ")
```

```
6 0 1 2 3 4 5
```

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

The **firstweekday parameter must be an integer between 0-6**.

- For this purpose, we can use the already-known constants.

the **Calendar class method** named **iterweekdays** returns an iterator for week day numbers.

- The first value returned is always equal to the value of the **firstweekday** property.

iterweekdays returns an iterator for week day numbers.

The itermonthdates() method

```
import calendar

c = calendar.Calendar()

for date in c.itermonthdates(2019, 11):
   print(date, end=" ")
```

The **itermonthdates method** requires specifying the year and month.

- All days in the specified month and year are returned, as well as all days before the beginning of the month or the end of the month that are necessary to get a complete week.

Each day is represented by a **datetime.date object**.

Other methods that return iterators

```
import calendar

c = calendar.Calendar()

for iter in c.itermonthdays(2019, 11):
   print(iter, end=" ")
```

```
0 0 0 0 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 0
```

The **itermonthdates** takes **year** and **month** as **parameters**

- It returns the iterator to the days of the week represented by numbers.

The **0**s are days outside the specified month range that are added to keep the complete week.

There are four other similar methods in the Calendar class that differ in data returned:

- **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 Python 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

```
import calendar

c = calendar.Calendar()

for data in c.monthdays2calendar(2020, 12):
   print(data)
```

```
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6)]
[(7, 0), (8, 1), (9, 2), (10, 3), (11, 4), (12, 5), (13, 6)]
[(14, 0), (15, 1), (16, 2), (17, 3), (18, 4), (19, 5), (20, 6)]
[(21, 0), (22, 1), (23, 2), (24, 3), (25, 4), (26, 5), (27, 6)]
[(28, 0), (29, 1), (30, 2), (31, 3), (0, 4), (0, 5), (0, 6)]
```

The **monthdays2calendar method** takes the year and month

- It returns a list of weeks in a specific month.
- Each week is a tuple consisting of day numbers and weekday numbers.

Summary

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:

```
print("Hello world!")
```

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.