Python Introduction

Where is Python used?

Python is a versatile programming language used in a wide range of applications, from game development to scientific and numeric computing, web development, desktop GUIs, and software development. Major companies and services such as Dropbox, UBER, Spotify, Pinterest, and BuzzFeed use Python extensively in their technology stacks. Python is also popular in education and is often used to teach programming due to its simplicity and ease of use. It's an excellent choice for building web applications, scientific computing, desktop GUIs, and business applications.

Compiler vs Interpreter

Two ways of translating high-level programming language to machine language: compilation and interpretation.

Compilation translates source code once and distributes as machine code file, while interpretation translates source code each time it is run.

Most programming languages are designed to be either compiled or interpreted, and not both.

A high-level programming language is usually designed to fall into one of these two categories depending on whether it is intended to be compiled or interpreted.

Interpreter

When a computer program is written, it exists as a text file that contains the source code. The source code must be in plain text format without any decorations. To execute the program, an interpreter is used to read and process the source code line by line from top to bottom and left to right. The interpreter checks if all subsequent lines are correct and if it finds an error, it immediately stops and displays an error message indicating the location and cause of the error. However, the error message may be misleading and not always located at the actual cause of the error. If the line is correct, the interpreter tries to execute it, and the "read-check-execute" trio can be repeated multiple times. It is possible for a significant part of the code to be executed successfully before an error is detected. There is no clear answer as to which method, compiling or interpreting, is better as both have their advantages and disadvantages.

COMPILATION

the execution of the translated code is usually faster; only the user has to have the compiler - the end-user may use the code without it;

the translated code is stored using machine language - as it is very hard to understand it, your own inventions and programming tricks are likely to remain your secret.

the compilation itself may be a very time-consuming process - you may not be able to run your code immediately after making an amendment;

you have to have as many compilers as hardware platforms you want your code to be run on.

INTERPRETER

you can run the code as soon as you complete it - there are no additional phases of translation;the code is stored using programming language, not machine language - this means that it can be run on computers using different machine languages; you don't compile your code separately for each different architecture. don't expect interpretation to ramp up your code to high speed - your code will share the computer's power with the interpreter, so it can't be really fast both you and the end user have to have the interpreter to run your code.

Introduction

- Python is a high-level programming language with dynamic semantics, widely used for general-purpose programming.
- The name of the Python programming language comes from the BBC television comedy sketch series called Monty Python's Flying Circus.
- Python was created by Guido van Rossum, born in 1956 in Haarlem, the Netherlands.
- Python is one of the few languages whose authors are known by name.
- The speed with which Python has spread around the world is a result of the continuous work of thousands of programmers, testers, users, and enthusiasts.

Reasons to use Python:

- Easy to learn, allowing for faster programming
- Easy to teach, allowing for more focus on general programming techniques
- Allows for faster writing of new software
- Code is often easier to understand, both for writing and reading
- Free, open, and multiplatform

Drawbacks of Python:

- Not the fastest programming language
- May be resistant to simpler testing techniques, making debugging more difficult
- However, making mistakes is also harder in Python.

Types of Python

1. CPython: The standard implementation of Python written in C.
2. Cython: automatically translates the Python code (clean and clear, but not too swift) into "C" code (complicated and talkative, but agile).
3. Jython: A Python implementation written in Java and designed to run on the Java Virtual Machine (JVM).
4. IronPython: An implementation of Python for .NET and Mono platforms.
5. PyPy: A fast, compliant alternative implementation of Python that includes a Just-In-Time (JIT) compiler. a Python within a Python. In other words, it represents a Python environment written in Python-like language named RPython (Restricted Python). It is actually a subset of Python.

6. The source code of PyPy is not run in the interpretation manner, but is instead translated into the C programming language and then executed separately.
7. MicroPython: A lightweight implementation of Python that is designed to run on microcontrollers and other small devices.

8. Anaconda: A popular distribution of Python used for data science and scientific computing that includes a number of scientific libraries and tools.

Print() function

1. The print() function is a **built-in** function. It prints/outputs a specified message to the screen/console window.

2. Built-in functions, contrary to user-defined functions, are always available and don't have to be imported. Python 3.8 comes with 69 built-in functions. You can find their full list provided in alphabetical order in the Python Standard Library.

3. To call a function (this process is known as **function invocation** or **function call**), you need to use the function name followed by parentheses. You can pass arguments into a function by placing them inside the parentheses. You must separate arguments with a comma, e.g., print("Hello,", "world!"). An "empty" print() function outputs an empty line to the screen.

4. Python strings are delimited with **quotes**, e.g., "I am a string" (double quotes), or 'I am a string, too' (single quotes).

5. Computer programs are collections of **instructions**. An instruction is a command to perform a specific task when executed, e.g., to print a certain message to the screen.

6. In Python strings the **backslash** (\) is a special character which announces that the next character has a different meaning, e.g., \n (the **newline character**) starts a new output line.

7. **Positional arguments** are the ones whose meaning is dictated by their position, e.g., the second argument is outputted after the first, the third is outputted after the second, etc.

8. **Keyword arguments** are the ones whose meaning is not dictated by their location, but by a special word (keyword) used to identify them.

9. The end and sep parameters can be used for formatting the output of the print() function.
The sep parameter specifies the separator between the outputted arguments, e.g., print("H", "E", "L", "L", "O", sep="-"), whereas the end parameter specifies what to print at the end of the print statement.

Function Invocation

- A function invocation is a statement that tells Python to execute a function with a specific name and arguments.
- The function invocation consists of the function name followed by parentheses that contain the argument(s) passed to the function. For example, in the statement print("Hello, World!"), "print" is the function name, and "Hello, World!" is the argument passed to the function.
- When Python encounters a function invocation, it checks if the function name is valid and if the number of arguments passed to the function matches the function's requirements. If either condition fails, Python will abort the code and raise an error.

- If the function name and arguments are valid, Python will execute the code inside the function and perform any desired effects or evaluations. For example, the print() function will output the argument(s) to the console.
- After the function finishes executing, Python returns to the point in the code where the function was invoked and continues executing the remaining code.
- Functions are an essential part of Python programming as they allow you to break down complex tasks into smaller, more manageable parts that can be reused and combined in various ways.

Literals

1. **Literals** are notations for representing some fixed values in code. Python has various types of literals - for example, a literal can be a number (numeric literals, e.g., 123), or a string (string literals, e.g., "I am a literal.").

2. The **binary system** is a system of numbers that employs *2* as the base. Therefore, a binary number is made up of 0s and 1s only, e.g., 1010 is *10* in decimal.

Octal and hexadecimal numeration systems, similarly, employ *8* and *16* as their bases respectively. The hexadecimal system uses the decimal numbers and six extra letters.

3. **Integers** (or simply **int**s) are one of the numerical types supported by Python. They are numbers written without a fractional component, e.g., 256, or -1 (negative integers). Python allows underscore Therefore, you can write this number either like this: 11111111, or like that: 11_111_111.

If an integer number is preceded by an 0O or 0o prefix (zero-o), it will be treated as an octal value. This means that the number must contain digits taken from the [0..7] range only.

0o123 is an octal number with a (decimal) value equal to 83.

The second convention allows us to use hexadecimal numbers. Such numbers should be preceded by the prefix 0x or 0X (zero-x).

0x123 is a hexadecimal number with a (decimal) value equal to 291.

4. **Floating-point** numbers (or simply **float**s) are another one of the numerical types supported by Python. They are numbers that contain (or are able to contain) a fractional component, e.g., 1.27.

5. To encode an apostrophe or a quote inside a string you can either use the escape character, e.g., 'I\'m happy.', or open and close the string using an opposite set of symbols to the ones you wish to encode, e.g., "I'm happy." to encode an apostrophe, and 'He said "Python", not "typhoon"' to encode a (double) quote.

6. **Boolean values** are the two constant objects True and False used to represent truth values (in numeric contexts 1 is True, while 0 is False.

EXTRA

There is one more, special literal that is used in Python: the None literal. This literal is a so-called NoneType object, and it is used to represent **the absence of a value**.

Operators and Expression

1. An **expression** is a combination of values (or variables, operators, calls to functions – you will learn about them soon) which evaluates to a certain value, e.g., 1 + 2.

2. **Operators** are special symbols or keywords which are able to operate on the values and perform (mathematical) operations, e.g., the * operator multiplies two values: x * y.

3. Arithmetic operators in Python: + (addition), - (subtraction), * (multiplication), / (classic division – always returns a float), % (modulus – divides left operand by right operand and returns the remainder of the operation, e.g., 5 % 2 = 1), ** (exponentiation – left operand raised to the power of right operand, e.g., 2 ** 3 = 2 * 2 * 2 = 8), // (floor/integer division – returns a number resulting from division, but rounded down to the nearest whole number, e.g., 3 // 2.0 = 1.0)

4. A **unary** operator is an operator with only one operand, e.g., -1, or +3.

5. A **binary** operator is an operator with two operands, e.g., 4 + 5, or 12 % 5.

6. Some operators act before others – **the hierarchy of priorities**:

- the ** operator (exponentiation) has the highest priority;
- then the unary + and - (note: a unary operator to the right of the exponentiation operator binds more strongly, for example: 4 ** -1 equals 0.25)
- then *, /, //, and %;
- and, finally, the lowest priority: the binary + and -.

7. Subexpressions in **parentheses** are always calculated first, e.g., 15 - 1 * (5 * (1 + 2)) = 0.

8. The **exponentiation** operator uses **right-sided binding**, e.g., 2 ** 2 ** 3 = 256.

Variables

1. A **variable** is a named location reserved to store values in the memory. A variable is created or initialized automatically when you assign a value to it for the first time. (2.1.4.1)

2. Each variable must have a unique name - an **identifier**. A legal identifier name must be a non-empty sequence of characters, must begin with the underscore(_), or a letter, and it cannot be a Python keyword. The first character may be followed by underscores, letters, and digits. Identifiers in Python are case-sensitive. (2.1.4.1)

3. Python is a **dynamically-typed** language, which means you don't need to *declare* variables in it. (2.1.4.3) To assign values to variables, you can use a simple assignment operator in the form of the equal (=) sign, i.e., var = 1.

4. You can also use **compound assignment operators** (shortcut operators) to modify values assigned to variables, e.g., var += 1, or var /= 5 * 2.

5. You can assign new values to already existing variables using the assignment operator or one of the compound operators,

Comments

1. Comments can be used to leave additional information in code. They are omitted at runtime. The information left in source code is addressed to human readers. In Python, a comment is a piece of text that begins with #. The comment extends to the end of line.

2. If you want to place a comment that spans several lines, you need to place # in front of them all. Moreover, you can use a comment to mark a piece of code that is not needed at the moment (see the last line of the snippet below), e.g.:

```
# This program prints an introduction to the screen.
print("Hello!")  # Invoking the print() function
# print("I'm Python.")
```

Input and string

1. The print() function **sends data to the console**, while the input() function **gets data from the console**.

2. The input() function comes with an optional parameter: **the prompt string**. It allows you to write a message before the user input, e.g.:

```
name = input("Enter your name: ")
print("Hello, " + name + ". Nice to meet you!")
```

3. When the input() function is called, the program's flow is stopped, the prompt symbol keeps blinking (it prompts the user to take action when the console is switched to input mode) until the user has entered an input and/or pressed the *Enter* key.

```
name = input("Enter your name: ")
print("Hello, " + name + ". Nice to meet you!")
print("\nPress Enter to end the program.")
input()
print("THE END.")
```

4. The result of the input() function is a string. You can add strings to each other using the concatenation (+) operator. Check out this code:

```
num_1 = input("Enter the first number: ") # Enter 12
num_2 = input("Enter the second number: ") # Enter 21
print(num_1 + num_2) # the program returns 1221
```

5. You can also multiply (* – replication) strings, e.g.:

```
num_1 = input("Enter the first number: ") # Enter 12
num_2 = input("Enter the second number: ") # Enter 2
print(num_1 * num_2) # the program returns 1212
```

6. You can change type of input from string to number with:

```
num_1 = int(input("Enter the first number: ")) # Enter 12
```

Comparison Operators

The comparison (otherwise known as relational) operators are used to compare values. The table below illustrates how the comparison operators work, assuming that x = 0, y = 1, and z = 0:

Operator	Description	Example
==	returns True if operands' values are equal, and False otherwise	
!=	returns True if operands' values are not equal, and False otherwise	x != y # True x != z # False
>	True if the left operand's value is greater than the right operand's value, and False otherwise	x > y # False y > z # True
<	True if the left operand's value is less than the right operand's value, and False otherwise	x < y # True y < z # False
≥	True if the left operand's value is greater than or equal to the right operand's value, and False otherwise	x >= y # False x >= z # True y >= z # True
≤	True if the left operand's value is less than or equal to the right operand's value, and False otherwise	x <= y # True x <= z # True y <= z # False

Conditional(if)

When you want to execute some code only if a certain condition is met, you can use a conditional statement:

a single if statement, e.g.:

```
x = 10
if x == 10: # condition
    print("x is equal to 10")  # Executed if the condition is True.
```

A series of if statements, e.g.:

```
x = 10
if x > 5: # condition one
    print("x is greater than 5")  # Executed if condition one is True.

if x < 10: # condition two
    print("x is less than 10")  # Executed if condition two is True.

if x == 10: # condition three
    print("x is equal to 10")  # Executed if condition three is True.
```

Each if statement is tested separately.

an if-else statement, e.g.:

```
x = 10
if x < 10:  # Condition
    print("x is less than 10")  # Executed if the condition is True.
else:
    print("x is greater than or equal to 10")  # Executed if the condition is False.
```

a series of if statements followed by an else, e.g.:

```
x = 10
if x > 5:  # True
    print("x > 5")

if x > 8:  # True
    print("x > 8")

if x > 10:  # False
    print("x > 10")

else:
    print("else will be executed")
```

Each if is tested separately. The body of else is executed if the last if is False.

The if-elif-else statement, e.g.:

```
x = 10
if x == 10:  # True
   print("x == 10")

if x > 15:  # False
   print("x > 15")

elif x > 10:  # False
   print("x > 10")

elif x > 5:  # True
   print("x > 5")

else:
   print("else will not be executed")
```

If the condition for if is False, the program checks the conditions of the subsequent elif blocks – the first elif block that is True is executed. If all the conditions are False, the else block will be executed.

Nested conditional statements, e.g.:

```
x = 10
if x > 5:  # True
   if x == 6:  # False
      print("nested: x == 6")
   elif x == 10:  # True
      print("nested: x == 10")
   else:
      print("nested: else")
else:
   print("else")
```

Loops

There are two types of loops in Python: while and for:

While

The while loop executes a statement or a set of statements as long as a specified boolean condition is true, e.g.:

```
# Example 1
while True:
   print("Stuck in an infinite loop.")

# Example 2
counter = 5
while counter > 2:
   print(counter)
   counter -= 1
```

For

The for loop executes a set of statements many times; it's used to iterate over a sequence (e.g., a list, a dictionary, a tuple, or a set - you will learn about them soon) or other objects that are iterable (e.g., strings). You can use the for loop to iterate over a sequence of numbers using the built-in range function. Look at the examples below:

```
# Example 1
word = "Python"
for letter in word:
    print(letter, end="*")

# Example 2
for i in range(1, 10):
    if i % 2 == 0:
        print(i)
```

You can use the break and continue statements to change the flow of a loop:

- You use break to exit a loop, e.g.:

```
text = "OpenEDG Python Institute"
for letter in text:
    if letter == "P":
        break
    print(letter, end="")
```

You use continue to skip the current iteration, and continue with the next iteration, e.g.:

```
text = "pyxpyxpyx"
for letter in text:
    if letter == "x":
        continue
    print(letter, end="")
```

The while and for loops can also have an else clause in Python. The else clause executes after the loop finishes its execution as long as it has not been terminated by break, e.g.:

```
n = 0
while n != 3:
    print(n)
    n += 1
else:
    print(n, "else")
print()
for i in range(0, 3):
    print(i)
else:
    print(i, "else")
```

The range() function generates a sequence of numbers. It accepts integers and returns range objects. The syntax of range() looks as follows: range(start, stop, step), where:

- start is an optional parameter specifying the starting number of the sequence (0 by default)
- stop is an optional parameter specifying the end of the sequence generated (it is not included),
- and step is an optional parameter specifying the difference between the numbers in the sequence (1 by default.)

Example code:

```
for i in range(3):
    print(i, end=" ")  # Outputs: 0 1 2

for i in range(6, 1, -2):
    print(i, end=" ")  # Outputs: 6, 4, 2
```

Logical and BitWise operators

Python supports the following logical operators:

- and → if both operands are true, the condition is true, e.g., (True and True) is True,
- or → if any of the operands are true, the condition is true, e.g., (True or False) is True,
- not → returns false if the result is true, and returns true if the result is false, e.g., not True is False.

2. You can use bitwise operators to manipulate single bits of data. The following sample data:

- x = 15, which is 0000 1111 in binary,
- y = 16, which is 0001 0000 in binary.

will be used to illustrate the meaning of bitwise operators in Python. Analyze the examples below:

- & does a *bitwise and*, e.g., x & y = 0, which is 0000 0000 in binary,
- | does a *bitwise or*, e.g., x | y = 31, which is 0001 1111 in binary,
- ˜ does a *bitwise not*, e.g., ˜ x = 240*, which is 1111 0000 in binary,
- ^ does a *bitwise xor*, e.g., x ^ y = 31, which is 0001 1111 in binary,
- >> does a *bitwise right shift*, e.g., y >> 1 = 8, which is 0000 1000 in binary,
- << does a *bitwise left shift*, e.g., y << 3 = , which is 1000 0000 in binary,

* -16 (decimal from signed 2's complement) -- read more about the Two's complement operation.

Lists

1. The **list is a type of data** in Python used to **store multiple objects**. It is an **ordered and mutable collection** of comma-separated items between square brackets, e.g.:

```
my_list = [1, None, True, "I am a string", 256, 0]
```

2. Lists can be **indexed and updated**, e.g.:

```
my_list = [1, None, True, 'I am a string', 256, 0]
print(my_list[3])  # outputs: I am a string
print(my_list[-1])  # outputs: 0

my_list[1] = '?'
```

```
print(my_list)  # outputs: [1, '?', True, 'I am a string', 256, 0]

my_list.insert(0, "first")
my_list.append("last")
print(my_list)  # outputs: ['first', 1, '?', True, 'I am a string', 256, 0, 'last']
```

3. Lists can be **nested**, e.g.:

```
my_list = [1, 'a', ["list", 64, [0, 1], False]]
```

You will learn more about nesting in module 3.1.7 - for the time being, we just want you to be aware that something like this is possible, too.

4. List elements and lists can be **deleted**, e.g.:

```
my_list = [1, 2, 3, 4]
del my_list[2]
print(my_list)  # outputs: [1, 2, 4]
del my_list  # deletes the whole list
```

5. Lists can be **iterated** through using the for loop, e.g.:

```
my_list = ["white", "purple", "blue", "yellow", "green"]
for color in my_list:
    print(color)
```

6. The len() function may be used to **check the list's length**, e.g.:

```
my_list = ["white", "purple", "blue", "yellow", "green"]
print(len(my_list))  # outputs 5

del my_list[2]
print(len(my_list))  # outputs 4
```

A typical **function** invocation looks as follows: result = function(arg), while a typical **method** invocation looks like this:result = data.method(arg).

8. You can use the sort() method to sort elements of a list, e.g.:

```
lst = [5, 3, 1, 2, 4]
print(lst)
lst.sort()
print(lst)  # outputs: [1, 2, 3, 4, 5]
```

9. There is also a list method called reverse(), which you can use to reverse the list, e.g.:

```
lst = [5, 3, 1, 2, 4]
print(lst)
lst.reverse()
print(lst)  # outputs: [4, 2, 1, 3, 5]
```

10. If you have a list l1, then the following assignment: l2 = l1 does not make a copy of the l1 list, but makes the variables l1 and l2 **point to one and the same list in memory**. For example:

```
vehicles_one = ['car', 'bicycle', 'motor']
print(vehicles_one) # outputs: ['car', 'bicycle', 'motor']

vehicles_two = vehicles_one
```

del vehicles_one[0] # deletes 'car'
print(vehicles_two) # outputs: ['bicycle', 'motor']

11. If you want to copy a list or part of the list, you can do it by performing **slicing**:
colors = ['red', 'green', 'orange']

copy_whole_colors = colors[:] # copy the entire list
copy_part_colors = colors[0:2] # copy part of the list

12. You can use **negative indices** to perform slices, too. For example:
sample_list = ["A", "B", "C", "D", "E"]
new_list = sample_list[2:-1]
print(new_list) # outputs: ['C', 'D']

13. The start and end parameters are **optional** when performing a slice: list[start:end], e.g.:
my_list = [1, 2, 3, 4, 5]
slice_one = my_list[2:]
slice_two = my_list[:2]
slice_three = my_list[-2:]

print(slice_one) # outputs: [3, 4, 5]
print(slice_two) # outputs: [1, 2]
print(slice_three) # outputs: [4, 5]

14. You can **delete slices** using the del instruction:
my_list = [1, 2, 3, 4, 5]
del my_list[0:2]
print(my_list) # outputs: [3, 4, 5]

del my_list[:]
print(my_list) # deletes the list content, outputs: []

Other ways to delete elements from a list:
a. my_list.pop(2) #Remove element at second position
b. my_list.remove(21) # Remove element 21 from the list

15. You can test if some items **exist in a list or not** using the keywords in and not in, e.g.:
my_list = ["A", "B", 1, 2]

print("A" in my_list) # outputs: True
print("C" not in my_list) # outputs: True
print(2 not in my_list) # outputs: False

Multidimensional List(List Comprehension)

1. **List comprehension** allows you to create new lists from existing ones in a concise and elegant way. The syntax of a list comprehension looks as follows:

[expression for element in list if conditional]

which is actually an equivalent of the following code:

```
for element in list:
    if conditional:
        expression
```

Here's an example of a list comprehension – the code creates a five-element list filled with the first five natural numbers raised to the power of 3:

```
cubed = [num ** 3 for num in range(5)]
print(cubed)  # outputs: [0, 1, 8, 27, 64]
```

2. You can use **nested lists** in Python to create **matrices** (i.e., two-dimensional lists). For example:
# A four-column/four-row table – a two dimensional array (4x4)

```
table = [[":(", ":)", ":(", ":)"],
         [":)", ":(", ":)", ":)"],
         [":(", ":)", ":)", ":("],
         [":)", ":)", ":)", ":("]]

print(table)
print(table[0][0])  # outputs: ':('
print(table[0][3])  # outputs: ':)'
```

3. You can nest as many lists in lists as you want, thereby creating n-dimensional lists, e.g., three-, four- or even sixty-four-dimensional arrays. For example:

```
# Cube - a three-dimensional array (3x3x3)
cube = [[[':(', 'x', 'x'],
         [':)', 'x', 'x'],
         [':(', 'x', 'x']],

        [[':)', 'x', 'x'],
         [':(', 'x', 'x'],
         [':)', 'x', 'x']],

        [[':(', 'x', 'x'],
         [':)', 'x', 'x'],
         [':)', 'x', 'x']]]

print(cube)
print(cube[0][0][0])  # outputs: ':('
print(cube[2][2][0])  # outputs: ':)'
```

Functions

1. A function is a block of code that performs a specific task when the function is called (invoked). You can use functions to make your code reusable, better organized, and more readable. Functions can have parameters and return values.

2. There are at least four basic types of functions in Python:

built-in functions which are an integral part of Python (such as the print() function). You can see a complete list of Python built-in functions at https://docs.python.org/3/library/functions.html.

The ones that come **from pre-installed modules** (you'll learn about them in the Python Essentials 2 course)
user-defined functions which are written by users for users - you can write your own functions and use them freely in your code,
the lambda functions

3. You can define your own function using the def keyword and the following syntax:

```
def your_function(optional parameters):
   # the body of the function
```

You can define a function which doesn't take any arguments, e.g.:

```
def message():    # defining a function
   print("Hello")    # body of the function
message()    # calling the function
```

4. You can pass information to functions by using parameters. Your functions can have as many parameters as you need.

```
#An example of a one-parameter function:
def hi(name):
   print("Hi,", name)
hi("Greg")

#An example of a three-parameter function:

def address(street, city, postal_code):
   print("Your address is:", street, "St.,", city, postal_code)

s = input("Street: ")
p_c = input("Postal Code: ")
c = input("City: ")

address(s, c, p_c)
```

5. You can pass arguments to a function using the following techniques:
positional argument passing in which the order of arguments passed matters
Ex. 1

```
def subtra(a, b):
   print(a - b)
```

```
subtra(5, 2)    # outputs: 3
subtra(2, 5)    # outputs: -3
```

keyword (named) argument passing in which the order of arguments passed doesn't matter (Ex. 2), a mix of positional and keyword argument passing (Ex. 3).

Ex. 2

```
def subtra(a, b):
    print(a - b)

subtra(a=5, b=2)    # outputs: 3
subtra(b=2, a=5)    # outputs: 3
```

Ex. 3

```
def subtra(a, b):
    print(a - b)

subtra(5, b=2)    # outputs: 3
subtra(5, 2)    # outputs: 3
```

It's important to remember that positional arguments mustn't follow keyword arguments. That's why if you try to run the following snippet:

```
def subtra(a, b):
    print(a - b)

subtra(5, b=2)    # outputs: 3
subtra(a=5, 2)    # Syntax Error
```

6. You can use the keyword argument passing technique to pre-define a value for a given argument:

```
def name(first_name, last_name="Smith"):
    print(first_name, last_name)
name("Andy")    # outputs: Andy Smith
name("Betty", "Johnson")    # outputs: Betty Johnson (the keyword argument replaced by "Johnson")
```

7. You can use the return keyword to tell a function to return some value. The return statement exits the function, e.g.:

```
def multiply(a, b):
    return a * b
print(multiply(3, 4))    # outputs: 12

def multiply(a, b):
    return

print(multiply(3, 4))    # outputs: None
```

8. The result of a function can be easily assigned to a variable, e.g.:

```
def wishes():
    return "Happy Birthday!"

w = wishes()
print(w)    # outputs: Happy Birthday!
```

Look at the difference in output in the following two examples:

```
# Example 1
def wishes():
    print("My Wishes")
    return "Happy Birthday"

wishes()    # outputs: My Wishes
```

9. You can use a list as a function's argument, e.g.:

```
def hi_everybody(my_list):
    for name in my_list:
        print("Hi,", name)

hi_everybody(["Adam", "John", "Lucy"])
```

10. A list can be a function result, too, e.g.:

```
def create_list(n):
    my_list = []
    for i in range(n):
        my_list.append(i)
    return my_list

print(create_list(5))
```

Scope in Functions

1. A variable that exists outside a function has a scope inside the function body (Example 1) unless the function defines a variable of the same name (Example 2), e.g.:

```
Example 1:
var = 2
def mult_by_var(x):
    return x * var
print(mult_by_var(7))    # outputs: 14

Example 2:
def mult(x):
    var = 7
    return x * var

var = 3
print(mult(7))    # outputs: 49
```

2. A variable that exists inside a function has a scope inside the function body (Example 4), e.g.:
Example 4:

```
def adding(x):
    var = 7
    return x + var
print(adding(4))    # outputs: 11
print(var)    # NameError
```

```
# You can use the global keyword followed by a variable name to make the variable's scope global,
var = 2
print(var)    # outputs: 2

def return_var():
    global var
    var = 5
    return var

print(return_var())    # outputs: 5
print(var)    # outputs: 5
```

Recursion

1. A function can call other functions or even itself. When a function calls itself, this situation is known as **recursion**, and the function which calls itself and contains a specified termination condition (i.e., the base case - a condition which doesn't tell the function to make any further calls to that function) is called a **recursive** function.
2. You can use recursive functions in Python to write **clean, elegant code, and divide it into smaller, organized chunks**. On the other hand, you need to be very careful as it might be **easy to make a mistake and create a function which never terminates**. You also need to remember that **recursive calls consume a lot of memory**, and therefore may sometimes be inefficient.
When using recursion, you need to take all its advantages and disadvantages into consideration.
The factorial function is a classic example of how the concept of recursion can be put in practice:
# Recursive implementation of the factorial function.

```
def factorial(n):
    if n == 1:    # The base case (termination condition.)
        return 1
    else:
        return n * factorial(n - 1)
print(factorial(4)) # 4 * 3 * 2 * 1 = 24
```

Unpacking Arguments

```
def multiply(*args):
    print(args)
    total = 1
    for arg in args:
        total = total * arg
    return total
```

```
print(multiply(3, 5))
print(multiply(-1))

# The asterisk takes all the arguments and packs them into a tuple.
# The asterisk can be used to unpack sequences into arguments too!


def add(x, y):
    return x + y
nums = [3, 5]
print(add(*nums))  # instead of add(nums[0], nums[1])

# -- Uses with keyword arguments --
# Double asterisk packs or unpacks keyword arguments


def add(x, y):
    return x + y

nums = {"x": 15, "y": 25}

print(add(**nums))

# -- Forced named parameter --
def multiply(*args):
    total = 1
    for arg in args:
        total = total * arg

    return total


def apply(*args, operator):
    if operator == "*":
        return multiply(args)
    elif operator == "+":
        return sum(args)
    else:
        return "No valid operator provided to apply()."

print(apply(1, 3, 6, 7, operator="+"))
print(apply(1, 3, 5, "+"))  # Error
```

Unpacking keyword arguments

```
# -- Unpacking kwargs --
def named(**kwargs):
    print(kwargs)
named(name="Bob", age=25)
# named({"name": "Bob", "age": 25})  # Error, the dictionary is actually a positional argument.
# Unpack dict into arguments. This is OK, but slightly more confusing. Good when working with
variables though.
named(**{"name": "Bob", "age": 25})
# -- Unpacking and repacking --
def named(**kwargs):
    print(kwargs)
def print_nicely(**kwargs):
    named(**kwargs)  # Unpack the dictionary into keyword arguments.
    for arg, value in kwargs.items():
        print(f"{arg}: {value}")
print_nicely(name="Bob", age=25)
# -- Both args and kwargs --
def both(*args, **kwargs):
    print(args)
    print(kwargs)
both(1, 3, 5, name="Bob", age=25)
# This is normally used to accept an unlimited number of arguments and keyword arguments, such
that some of them can be passed onto other functions.
# You'll frequently see things like these in Python code:
"""
def post(url, data=None, json=None, **kwargs):
    return request('post', url, data=data, json=json, **kwargs)
"""
# While the implementation is irrelevant at this stage, what it allows is for the caller of `post()` to
pass arguments to `request()`.
# -- Error when unpacking --
def myfunction(**kwargs):
    print(kwargs)
myfunction(**"Bob")  # Error, must be mapping
myfunction(**None)  # Error
```

First Class Functions

```
# A first class function just means that functions can be passed as arguments to functions.
def calculate(*values, operator):
    return operator(*values)

def divide(dividend, divisor):
    if divisor != 0:
        return dividend / divisor
    else:
        return "You fool!"

# We pass the `divide` function as an argument
result = calculate(20, 4, operator=divide)
print(result)

def average(*values):
    return sum(values) / len(values)
```

```
result = calculate(10, 20, 30, 40, operator=average)
print(result)

# -- searching with first-class functions --
def search(sequence, expected, finder):
    for elem in sequence:
        if finder(elem) == expected:
            return elem
    raise RuntimeError(f"Could not find an element with {expected}")

friends = [
    {"name": "Rolf Smith", "age": 24},
    {"name": "Adam Wool", "age": 30},
    {"name": "Anne Pun", "age": 27},
]

def get_friend_name(friend):
    return friend["name"]

print(search(friends, "Bob Smith", get_friend_name))

# -- using lambdas since this can be simple enough --
def search(sequence, expected, finder):
    for elem in sequence:
        if finder(elem) == expected:
            return elem
    raise RuntimeError(f"Could not find an element with {expected}")
friends = [
    {"name": "Rolf Smith", "age": 24},
    {"name": "Adam Wool", "age": 30},
    {"name": "Anne Pun", "age": 27},
]

print(search(friends, "Bob Smith", lambda friend: friend["name"]))

# -- or as an extra, using built-in functions --
from operator import itemgetter

def search(sequence, expected, finder):
    for elem in sequence:
        if finder(elem) == expected:
            return elem
    raise RuntimeError(f"Could not find an element with {expected}")

friends = [
    {"name": "Rolf Smith", "age": 24},
    {"name": "Adam Wool", "age": 30},
    {"name": "Anne Pun", "age": 27},
]

print(search(friends, "Rolf Smith", itemgetter("name")))
```

Tuples

1. **Tuples** are ordered and unchangeable (immutable) collections of data. They can be thought of as immutable lists. They are written in round brackets:

```
my_tuple = (1, 2, True, "a string", (3, 4), [5, 6], None)
print(my_tuple)

my_list = [1, 2, True, "a string", (3, 4), [5, 6], None]
print(my_list)
```

Each tuple element may be of a different type (i.e., integers, strings, booleans, etc.). What is more, tuples can contain other tuples or lists (and the other way round).

2. You can create an empty tuple like this:

```
empty_tuple = ()
print(type(empty_tuple))    # outputs: <class 'tuple'>
```

3. A one-element tuple may be created as follows:

```
one_elem_tuple_1 = ("one", )    # Brackets and a comma.
one_elem_tuple_2 = "one",       # No brackets, just a comma.
```

If you remove the comma, you will tell Python to create a **variable**, not a tuple:

```
my_tuple_1 = 1,
print(type(my_tuple_1))    # outputs: <class 'tuple'>

my_tuple_2 = 1             # This is not a tuple.
print(type(my_tuple_2))    # outputs: <class 'int'>
```

4. You can access tuple elements by indexing them:

```
my_tuple = (1, 2.0, "string", [3, 4], (5, ), True)
print(my_tuple[3])    # outputs: [3, 4]
```

5. Tuples are **immutable**, which means you cannot change their elements (you cannot append tuples, or modify, or remove tuple elements). The following snippet will cause an exception:

```
my_tuple = (1, 2.0, "string", [3, 4], (5, ), True)
my_tuple[2] = "guitar"    # The TypeError exception will be raised.
```

However, you can delete a tuple as a whole:

```
my_tuple = 1, 2, 3,
del my_tuple
print(my_tuple)    # NameError: name 'my_tuple' is not defined
```

6. You can loop through a tuple elements (Example 1), check if a specific element is (not)present in a tuple (Example 2), use the len() function to check how many elements there are in a tuple (Example 3), or even join/multiply tuples (Example 4):

```
# Example 1
tuple_1 = (1, 2, 3)
for elem in tuple_1:
    print(elem)

# Example 2
tuple_2 = (1, 2, 3, 4)
print(5 in tuple_2)
print(5 not in tuple_2)


# Example 4
tuple_4 = tuple_1 + tuple_2
tuple_5 = tuple_3 * 2
print(tuple_4)
print(tuple_5)
```

EXTRA

You can also create a tuple using a Python built-in function called tuple(). This is particularly useful when you want to convert a certain iterable (e.g., a list, range, string, etc.) to a tuple:

```
my_tuple = tuple((1, 2, "string"))
print(my_tuple)

my_list = [2, 4, 6]
tup = tuple(my_list)
print(tup)    # outputs: (2, 4, 6)
print(type(tup))    # outputs: <class 'tuple'>
```

By the same fashion, when you want to convert an iterable to a list, you can use a Python built-in function called list():

```
tup = 1, 2, 3,
my_list = list(tup)
print(type(my_list))    # outputs: <class 'list'>
```

Destructuring Variables

```
x, y = 5, 11
# x, y = (5, 11)
# -- Destructuring in for loops --
student_attendance = {"Rolf": 96, "Bob": 80, "Anne": 100}

for student, attendance in student_attendance.items():
    print(f"{student}: {attendance}")

# -- Another example --
people = [("Bob", 42, "Mechanic"), ("James", 24, "Artist"), ("Harry", 32, "Lecturer")]
for name, age, profession in people:
    print(f"Name: {name}, Age: {age}, Profession: {profession}")

# -- Much better than this! --

for person in people:
  print(f"Name: {person[0]}, Age: {person[1]}, Profession: {person[2]}")


# -- Ignoring values with underscore --
person = ("Bob", 42, "Mechanic")
name, _, profession = person
print(name, profession)  # Bob Mechanic

# -- Collecting values --
head, *tail = [1, 2, 3, 4, 5]

print(head)  # 1
print(tail)  # [2, 3, 4, 5]

*head, tail = [1, 2, 3, 4, 5]
print(head)  # [1, 2, 3, 4]
print(tail)  # 5
```

Dictionary

1. Dictionaries are unordered*****, changeable (mutable), and indexed collections of data. (*****In Python 3.6x dictionaries have become ordered by default. Keys can be strings, numbers. While values can be lists, another dictionary, tuples, lists, etc as well.

Each dictionary is a set of *key: value* pairs. You can create it by using the following syntax:

```
my_dictionary = {
  key1: value1,
  key2: value2,
  key3: value3
  }
```

2. If you want to access a dictionary item, you can do so by making a reference to its key inside a pair of square brackets (ex. 1) or by using the get() method (ex. 2):

```
pol_eng_dictionary = {
  "kwiat": "flower",
  "woda": "water",
  "gleba": "soil"
  }

item_1 = pol_eng_dictionary["gleba"]    # ex. 1
print(item_1)    # outputs: soil
item_2 = pol_eng_dictionary.get("woda")
print(item_2)    # outputs: water
```

Using get will return None if key doesn’t exist. Using square brackets will throw error if key doesn’t exist.

3. If you want to change the value associated with a specific key, you can do so by referring to the item's key name in the following way:

```
pol_eng_dictionary = {
   "zamek": "castle",
   "woda": "water",
   "gleba": "soil"
   }

pol_eng_dictionary["zamek"] = "lock"
item = pol_eng_dictionary["zamek"]
print(item)  # outputs: lock
```

4. To add or remove a key (and the associated value), use the following syntax:

```
phonebook = {}    # an empty dictionary
phonebook["Adam"] = 3456783958    # create/add a key-value pair
print(phonebook)    # outputs: {'Adam': 3456783958}

del phonebook["Adam"]
print(phonebook)    # outputs: {}
```

You can also insert an item to a dictionary by using the update() method, and remove the last element by using the popitem() method, e.g.:

```
pol_eng_dictionary = {"kwiat": "flower"}
pol_eng_dictionary.update({"gleba": "soil"})
print(pol_eng_dictionary)    # outputs: {'kwiat': 'flower', 'gleba': 'soil'}

pol_eng_dictionary.popitem()
print(pol_eng_dictionary)    # outputs: {'kwiat': 'flower'}
```

5. You can use the for loop to loop through a dictionary,Using

- dict.keys()
- dict.values()
- dict.items()

e.g.:

```
pol_eng_dictionary = {
   "zamek": "castle", "woda": "water", "gleba": "soil"
   }

for item in pol_eng_dictionary:
   print(item)
# outputs: zamek#         woda#         gleba
```

6. If you want to loop through a dictionary's keys and values, you can use the items() method, e.g.:

```
for key, value in pol_eng_dictionary.items():
   print("Pol/Eng ->", key, ":", value)
```

7. To check if a given key exists in a dictionary, you can use the in keyword:

```
if "zamek" in pol_eng_dictionary:
    print("Yes")
else:
    print("No")
```

8. You can use the del keyword to remove a specific item, or delete a dictionary. To remove all the dictionary's items, you need to use the clear() method:

```
print(len(pol_eng_dictionary))    # outputs: 3
del pol_eng_dictionary["zamek"]    # remove an item
print(len(pol_eng_dictionary))    # outputs: 2
pol_eng_dictionary.clear()   # removes all the items
print(len(pol_eng_dictionary))    # outputs: 0
del pol_eng_dictionary    # removes the dictionary
```

9. To copy a dictionary, use the copy() method:

```
copy_dictionary = pol_eng_dictionary.copy()
```

Sets

Collection of unique elements

a = {} #Empty
a = set()
a.add(5)
Cannot add list or dicts. Unordered, cannot change values.
a.remove(5) # Removes 5 if present or error thrown
a.pop() # Removes random element
a.clear()
a.union(a2)
a.intersect(a2)
Note: For a set 20 = 20.0

Exceptions

1. In Python, there is a distinction between two kinds of errors:

- **syntax errors** (parsing errors), which occur when the parser comes across a statement that is incorrect. For example:

Trying to execute the following line:

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

will cause a *SyntaxError*, and result in the following (or similar) message being displayed in the console:

```
  File "main.py", line 1
    print("Hello, World!)
                        ^
SyntaxError: EOL while scanning string literal
```

Pay attention to the arrow – it indicates the place where the Python parser has run into trouble. In our case, it's the missing double quote. Did you notice it?

- **exceptions**, which occur even when a statement/expression is syntactically correct; these are the errors that are detected during execution when your code results in an error which is not *uncoditionally fatal*. For example:

Trying to execute the following line:
print(1/0)

will cause a *ZeroDivisionError* exception, and result in the following (or similar) message being displayed in the console:

```
Traceback (most recent call last):
  File "main.py", line 1, in
    print(1/0)
ZeroDivisionError: division by zero
```

Pay attention to the last line of the error message – it actually tells you what happened. There are many different types of exceptions, such as *ZeroDivisionError*, *NameError*, *TypeError*, and many more; and this part of the message informs you of what type of exception has been raised. The preceding lines show you the context in which the exception has occured.

2. You can "catch" and handle exceptions in Python by using the *try-except* block. So, if you have a suspicion that any particular snippet may raise an exception, you can write the code that will gracefully handle it, and will not interrupt the program. Look at the example:

```
while True:
    try:
        number = int(input("Enter an integer number: "))
        print(number/2)
        break
    except:
        print("Warning: the value entered is not a valid number. Try again...")
```

The code above asks the user for input until they enter a valid integer number. If the user enters a value that cannot be converted to an int, the program will print Warning: the value entered is not a valid number. Try again..., and ask the user to enter a number again. What happens in such a case?

1. The program enters the *while* loop.
2. The try block/clause is executed. The user enters a wrong value, for example: hello!.
3. An exception occurs, and the rest of the try clause is skipped. The program jumps to the except block, executes it, and then continues running after the *try-except* block.

If the user enters a correct value and no exception occurs, the subsequent instructions in the *try* block are executed.

3. You can handle multiple exceptions in your code block. Look at the following examples:

```
while True:
    try:
        number = int(input("Enter an int number: "))
        print(5/number)
        break
    except ValueError:
        print("Wrong value.")
    except ZeroDivisionError:
        print("Sorry. I cannot divide by zero.")
    except:
        print("I don't know what to do...")
```

You can use multiple *except* blocks within one *try* statement, and specify particular exception names. If one of the except branches is executed, the other branches will be skipped. Remember: you can specify a particular built-in exception only once. Also, don't forget that the **default** (or generic) exception, that is the one with no name specified, should be placed **at the bottom of the branch** (use the more specific exceptions first, and the more general last).
You can also specify and handle multiple built-in exceptions within a single *except* clause:

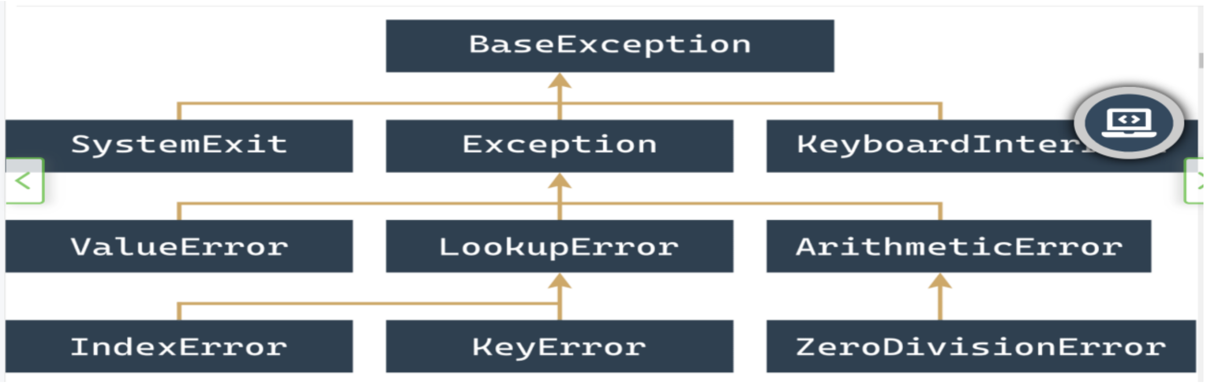
```
while True:
    try:
        number = int(input("Enter an int number: "))
        print(5/number)
        break
    except (ValueError, ZeroDivisionError):
        print("Wrong value or No division by zero rule broken.")
    except:
        print("Sorry, something went wrong...")
```

At most, one of the except branches is executed – none of the branches is performed when the raised exception doesn't match any of the specified exceptions.

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

4. Some of the most useful Python built-in exceptions are: *ZeroDivisionError*, *ValueError*, *TypeError*, *AttributeError*, and *SyntaxError*. One more exception that, in our opinion, deserves your attention is the *KeyboardInterrupt* exception, which is raised when the user hits the interrupt key (CTRL-C or Delete). Run the code above and hit the key combination to see what happens.
To learn more about the Python built-in exceptions, consult the official Python documentation.

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



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

```
try:
   # Risky code.
except IndexError:
   # Taking care of mistreated lists
except LookupError:
   # Dealing with other erroneous lookups
```

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

```
try:
   # Risky code.
except LookupError:
   # Dealing with erroneous lookups
except IndexError:
    You'll never get here
```

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

7. 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.

8. Last but not least, you should remember about testing and debugging your code. Use such debugging techniques as *print* debugging; if possible – ask someone to read your code and help you to find bugs in it or to improve it; try to isolate the fragment of code that is problematic and susceptible to errors: **test your functions** by applying predictable argument values, and try to **handle** the situations when someone enters wrong values; **comment out** the parts of the code that obscure the issue. Finally, take breaks and come back to your code after some time with a fresh pair of eyes.

9. ZeroDivisionError falling into ArithmeticError branch

```
try:
   y = 1 / 0
except ArithmeticError:
   print("Oooppsss...")
```

ZeroDivisionError is a specific case of ArithmeticError.
In this example, the ZeroDivisionError exception falls into the branch for ArithmeticError since ZeroDivisionError is more specific.

10. Example: Handling multiple exceptions in the same way

```
try:
    y = 1 / 0
except (ZeroDivisionError, ArithmeticError):
    print("Arithmetic problem!")
```

Multiple exceptions (ZeroDivisionError and ArithmeticError) are handled in the same way using a tuple in the except clause.

11. Example: Putting more general exceptions before concrete ones

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print("Zero division!")
except ArithmeticError:
    print("Arithmetic problem!")
```

The order of exception branches matters. Here, the more general ArithmeticError branch is listed after the specific ZeroDivisionError branch.
As a result, the ZeroDivisionError branch becomes unreachable and the ArithmeticError branch handles the exception.

12. Example: Handling exceptions inside a function

```
def bad_fun(n):
    return 1 / n
try:
    bad_fun(0)
except ArithmeticError:
    print("What happened? An exception was raised!")
```

Exceptions raised inside a function can be handled within the function itself.
In this example, the exception raised inside the bad_fun() function is caught and handled within the function.

13. Example: Propagation of exception to the invoker

```
def bad_fun(n):
    return 1 / n

try:
    bad_fun(0)
except ArithmeticError:
    print("What happened? An exception was raised!")
```

If an exception is raised inside a function and not handled within the function, it propagates to the invoker or higher levels to be handled.

14. Example: Raising a specified exception

```
raise ZeroDivisionError("Oops, division by zero!")
```

The raise statement is used to raise a specified exception with a custom error message.
In this example, a ZeroDivisionError exception is raised with the message "Oops, division by zero!".

15. Example: Simulating raising and handling an exception

```
try:
    raise ZeroDivisionError("Oops, division by zero!")
except ZeroDivisionError as e:
    print("Handled exception:", str(e))
```

The raise statement can be used to simulate raising an exception for testing or handling purposes.
In this example, a ZeroDivisionError exception is raised and then caught in the except block, where it is handled.

16. Example: Re-raising the same exception

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print("I did it again!")
    raise
```

The raise statement without an exception name can be used inside an except block to re-raise the same exception.
In this example, the ZeroDivisionError exception is first caught and then re-raised, propagating it further.

17. Example: Using assert to check an expression

```
x = 5
assert x >= 0.0
```

The assert statement checks an expression and raises an AssertionError if the expression evaluates to false or None.
In this example, the assertion checks if x is greater than or equal to 0.0. If the assertion fails, an AssertionError is raised.

18. Some Common Exceptions

- ArithmeticError: An exception for arithmetic operation errors.
- AssertionError: Raised when an assert statement fails.
- BaseException: The most general exception, including all other exceptions.
- IndexError: Raised when accessing a non-existent element in a sequence.
- KeyboardInterrupt: Raised when the user interrupts the program execution.
- LookupError: An abstract exception for invalid references to collections.
- MemoryError: Raised when there is insufficient memory to perform an operation.
- OverflowError: Raised when a number is too large to be stored.
- ImportError: Raised when an import operation fails.
- KeyError: Raised when accessing a non-existent element in a collection.

Exceptions as Objects

The try-except block can have an additional branch called else, which is executed only when no exception is raised.

The else branch must be located after the last except branch.

```
try: # Code that may raise an exception
        result = 1 / 2
        print("Everything went fine")
except ZeroDivisionError:
        print("Division failed")
else:
        print(result)
```

Output:

Everything went fine 0.5

- The try-except block can also include a finally block, which is always executed, regardless of whether an exception occurred or was handled. Finally block can exist without else as well.

Example:

```
try: # Code that may raise an exception
        result = 1 / 0
        print("Everything went fine")
except ZeroDivisionError:
        print("Division failed")
finally:
        print("It's time to say goodbye")
```

Division failed It's time to say goodbye

Exceptions are Classes:

- Exceptions in Python are implemented as classes.
- When an exception is raised, an object of the corresponding exception class is instantiated.
- The exception object carries useful information about the exception.
- All built-in Python exceptions form a hierarchy of classes.

Example: Prints all exceptions in hierarchial format:

```
def print_exception_tree(exception_class, indent=""):
  print(indent + exception_class.__name__)
  subclasses = exception_class.__subclasses__()
  for subclass in subclasses:
    print_exception_tree(subclass, indent + "    ")

print_exception_tree(BaseException)
```

Detailed Anatomy of Exceptions:

The BaseException class introduces the args property, which contains the arguments passed to the exception's constructor.

```
def print_exception_args(exception):
    print(exception)
    print(exception.args)

# Case 1: No arguments
try:
    raise Exception
except Exception as e:
    print_exception_args(e)

# Case 2: One argument
try:
    raise Exception("Error occurred")
except Exception as e:
    print_exception_args(e)

# Case 3: Two arguments
try:
    raise Exception("Invalid value", 42)
except Exception as e:
    print_exception_args(e)
```

Output:

```
Exception
()
Exception: Error occurred
('Error occurred',)
Exception: ('Invalid value', 42)
(('Invalid value', 42),)
```

How to Create Your Own Exception:

Python allows creating custom exceptions by defining new classes that inherit from existing exception classes.

Example:

```
class MyZeroDivisionError(ZeroDivisionError):
    pass

def do_the_division(x):
    if x == 0:
        raise MyZeroDivisionError("My custom zero division error")
    else:
        return 1 / x
```

```
try:
    print(do_the_division(2))
    print(do_the_division(0))
except MyZeroDivisionError:
    print("My custom zero division error occurred")
except ZeroDivisionError:
    print("Regular zero division error occurred")
```

```
0.5
My custom zero division error occurred
```

Modules

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

1. If you want to import a module as a whole, you can do it using the import module_name statement. You are allowed to import more than one module at once using a comma-separated list. For example:

```
import mod1
import mod2, mod3, mod4
```

although the latter form is not recommended due to stylistic reasons, and it's better and prettier to express the same intention in more a verbose and explicit form, such as:

```
import mod2
import mod3
import mod4
```

2. If a module is imported in the above manner and you want to access any of its entities, you need to prefix the entity's name using dot notation. For example:

```
import my_module
result = my_module.my_function(my_module.my_data)
```

The snippet makes use of two entities coming from the my_module module: a function named my_function() and a variable named my_data. Both names **must be prefixed by** my_module. None of the imported entity names conflicts with the identical names existing in your code's namespace.

3. You are allowed not only to import a module as a whole, but to import only individual entities from it. In this case, the imported entities **must not be** prefixed when used.

For example:

```
from module import my_function, my_data
result = my_function(my_data)
```

The above way - despite its attractiveness - is not recommended because of the danger of causing conflicts with names derived from importing the code's namespace.

4. The most general form of the above statement allows you to import **all entities** offered by a module:

```
from my_module import *
result = my_function(my_data)
```

Note: this import's variant is not recommended due to the same reasons as previously (the threat of a naming conflict is even more dangerous here).

5. You can change the name of the imported entity "on the fly" by using the as phrase of the import. For example:

```
from module import my_function as fun, my_data as dat
 result = fun(dat)
```

Common Modules

A function named dir() can show you a list of the entities contained inside an imported module. For example:

```
import os
dir(os)
```

prints out the list of all the os module's facilities you can use in your code.

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

3. The random module groups more than 60 entities designed to help you use pseudo-random numbers. Don't forget the prefix "random", as there is no such thing as a real random number when it comes to generating them using the computer's algorithms.

4. The platform module contains about 70 functions which let you dive into the underlaying layers of the OS and hardware. Using them allows you to get to know more about the environment in which your code is executed.

5. **Python Module Index** (https://docs.python.org/3/py-modindex.html is a community-driven directory of modules available in the Python universe. If you want to find a module fitting your needs, start your search there.

Packages

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. 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.

3. 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.

4. 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.

5. 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.

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

Ex:

```
my_package/
   __init__.py
   module1.py
   module2.py
```

```
from sys import path
path.append('.. \ \my_package)
```

```
import my_package.module1
import my_package.module2

my_package.module1.function1()
result = my_package.module2.function2()
```

Pip

1. A **repository** (or **repo** for short) designed to collect and share free Python code 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** (***p****ip* ***i****nstalls* ***p****ackages* 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

4. The list of the main **pip** activities looks as follows:

- 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* – searches through PyPI directories in order to find packages whose names contain *anystring*;
- 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 install *name* – version – Installs specific version of package
- pip uninstall *name* – uninstalls a previously installed package.

Strings

1. Computers store characters as numbers. There is more than one possible way of encoding characters, but only some of them gained worldwide popularity and are commonly used in IT: these are ASCII (used mainly to encode the Latin alphabet and some of its derivates) and UNICODE (able to encode virtually all alphabets being used by humans).

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

3. UNICODE uses different ways of encoding when it comes to storing the characters using files or computer memory: two of them are UCS-4 and UTF-8 (the latter is the most common as it wastes less memory space).

Nature of Strings

1. Python strings are **immutable sequences** and can be indexed, sliced, and iterated like any other sequence, as well as being subject to the in and not in operators. There are two kinds of strings in Python:

- **one-line** strings, which cannot cross line boundaries – we denote them using either apostrophes ('string') or quotes ("string")
- **multi-line** strings, which occupy more than one line of source code, delimited by trigraphs:

- **'''**
- **string**
- **'''**
-

or
"""
string
"""

2. The length of a string is determined by the len() function. The escape character (\) is not counted. For example:
print(*len*("\n\n"))

outputs 2.
3. Strings can be **concatenated** using the + operator, and **replicated** using the * operator. For example:
asterisk = ''*
plus = "+"
*decoration = (asterisk + plus) * 4 + asterisk*

print(decoration)
*outputs *+*+*+*+*.*

4. The pair of functions chr() and ord() can be used to create a character using its codepoint, and to determine a codepoint corresponding to a character. Both of the following expressions are always true:
chr(ord(character)) == character
ord(chr(codepoint)) == codepoint

5. Some other functions that can be applied to strings are:
- list() – creates a list consisting of all the string's characters;
- max() – finds the character with the maximal codepoint;
- min() – finds the character with the minimal codepoint.

6. The method named index() finds the index of a given substring inside the string.

Methods of String

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(can specify what character to use to make string of that 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;(can specify what character to strip)
- 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?

3. 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**.

4. 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*

5. A number can be converted to a string using the str() function.
6. 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).

String Formatting

In Python, you can use f-strings for concise string formatting.

1. Using variables: Variable can be defined before or after the string

```
name = "John"
age = 25
message = f"My name is {name} and I am {age} years old."
print(message)
```

OR

```
message = "My name is {name} and I am {age} years old."
print(message.format(name = "John", age = 25))
```

# You can also apply formatting options directly within the curly braces.

```
pi = 3.14159
formatted_pi = f"The value of pi is: {pi:.2f}"
print(formatted_pi)
```

OR

```
formatted_pi = “The value of pi is: {:.2f}"
print(formatted_pi.format(26.3))
```

Object-Oriented Programming Fundamentals

- Object-Oriented Programming (OOP) is a programming paradigm that represents real-world entities as objects to create reusable and modular code.
- Contrasted with procedural programming, OOP is especially advantageous for large and complex projects with multiple developers.
- Python is a versatile language suitable for both object-oriented and procedural programming.

Key Concepts of OOP:

1. Classes and Objects:
    - Classes are blueprints for creating objects, defining their properties and behaviors.

- Objects are instances of classes, representing individual entities with distinct characteristics.

2. Encapsulation:
    - Encapsulation hides the internal workings of objects from external access, protecting data integrity.
    - It allows the object to control how its data is accessed and modified through public and private methods.
    - In python private variables defined with two leading underscores __
3. Inheritance:
    - Inheritance allows a class (subclass) to inherit properties and behaviors from another class (superclass).
    - Subclasses can extend and modify the functionalities of the superclass, promoting code reuse.
4. Polymorphism:
    - Polymorphism enables objects to take on multiple forms and behave differently based on the context.
    - It allows the use of a common interface to work with different types of objects.

Benefits of OOP:

1. Reusability:
    - OOP promotes modular code, allowing developers to reuse classes and objects in different parts of the program.
2. Maintainability:
    - The modular nature of OOP makes code easier to manage, debug, and update, enhancing maintainability.
3. Scalability:
    - OOP supports managing large projects by breaking them down into smaller, manageable classes and objects.
4. Collaboration:
    - OOP fosters teamwork as developers can work on different parts of the project independently.

Example:

```
 --Base Clasee
class Stack:
   def __init__(self): # Self is mandatory argument in class methods
      self.__stk = []  #_stk is private

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

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

#Child Class
class CountingStack(Stack):
   def __init__(self): #Constructor Class
```

```
        Stack.__init__(self)#Calling parent constructor(Mandatory to specify)
        self.__counter = 0

    def get_counter(self): #getter method
        return self.__counter

    def pop(self):
        self.__counter += 1
        return Stack.pop(self)

stk = CountingStack() –Object creation
for i in range(100):
    stk.push(i)
    stk.pop()
print(stk.get_counter())
```

Instance and Class Variables

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.

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 objname._ClassName__PrivatePropertyName.

2. 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.

If class and instance variable have same name, instance variable takes precedence.
3. A function named hasattr() can be used to determine if any object/class contains a specified property.

For example:

```
class Sample:
    gamma = 0 # Class variable.
    def __init__(self):

        self.alpha = 1 # Instance variable.
        self.__delta = 3 # Private instance variable.

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

The code outputs: {'alpha': 1, '_Sample__delta': 3, 'beta': 2}

Methods

1. A method is a function embedded inside a class. The first (or only) parameter of each method is usually named self(mandatory), 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.

Example

```
class Sample:
    def __init__(self):
        self.name = Sample.__name__
    def myself(self):
        print("My name is " + self.name + " living in a " + Sample.__module__)

obj = Sample()
obj.myself()

print(Sample.__name__)
print(Sample.__module__)
print(Sample. __bases__ )# Will return Object if no superclass
```

4. Investigating a Class
Example:

```
class MyClass:
    pass

obj = MyClass()
obj.a = 1
obj.b = 2
obj.i = 3
obj.ireal = 3.5
obj.integer = 4
obj.z = 5


def incIntsI(obj):
    for name in obj.__dict__.keys():
        if name.startswith('i'):
            val = getattr(obj, name)
            if isinstance(val, int):
                setattr(obj, name, val + 1)


print(obj.__dict__)
```

```
incIntsI(obj)
print(obj.__dict__)
```

Above def incInts gets all the integer attributes and increases the value by 1.
Output:

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

Inheritance

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

```
class Mouse:
   def __init__(self, name):
      self.my_name = name
   def __str__(self):
      return self.my_name
the_mouse = Mouse('mickey')
print(the_mouse) # Prints "mickey".
```

2. A function named issubclass(Class_1, Class_2) is able to determine if Class_1 is a **subclass** of Class_2. For example:

```
class Mouse:
   pass
class LabMouse(Mouse):
   pass
print(issubclass(Mouse, LabMouse), issubclass(LabMouse, Mouse)) # Prints "False True
```

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

```
class Mouse:
   pass
 class LabMouse(Mouse):
   pass
mickey = Mouse()
print(isinstance(mickey, Mouse), isinstance(mickey, LabMouse)) # Prints "True False".
```

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

```
class Mouse:
   pass
mickey = Mouse()
minnie = Mouse()
cloned_mickey = mickey
print(mickey is minnie, mickey is cloned_mickey) # Prints "False True".
```

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

```
class Mouse:
    def __str__(self):
        return "Mouse”
class LabMouse(Mouse):
    def __str__(self):
        return "Laboratory " + super().__str__()

doctor_mouse = LabMouse();

print(doctor_mouse) # Prints "Laboratory Mouse".
```

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

```
class Mouse:
    Population = 0
    def __init__(self, name):
      self.name = name

    def __str__(self):
        return "Hi, my name is " + self.name

class LabMouse(Mouse):
    pass

professor_mouse = LabMouse("Professor Mouser")

print(professor_mouse, Mouse.Population) # Prints "Hi, my name is Professor Mouser 1"
```

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

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

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

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

```
  def __str__(self):
  return "My name is " + self.name

class AncientMouse(Mouse):
  def __str__(self):
  return "Meum nomen est " + self.name

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

Types of Inheritance

In Python, there are several types of inheritance that can be used to create class hierarchies and reuse code. The main types of inheritance in Python are:

Single Inheritance: In single inheritance, a subclass inherits from a single superclass. It forms a simple parent-child relationship between classes. The subclass inherits all the attributes and methods of the superclass. This is the most common and basic form of inheritance in Python.

Multiple Inheritance: Multiple inheritance allows a subclass to inherit from multiple superclasses. The subclass can inherit attributes and methods from multiple parent classes. It enables code reuse by combining features from different classes. However, it can lead to the "diamond problem" which we'll explain shortly.

Multilevel Inheritance: Multilevel inheritance involves creating a hierarchy of classes where each class inherits from another class. It forms a parent-child-grandchild relationship between classes. Each subclass inherits the attributes and methods from its immediate superclass, forming a chain of inheritance.

Hierarchical Inheritance: Hierarchical inheritance occurs when multiple subclasses inherit from a single superclass. It forms a tree-like structure with a single superclass and multiple subclasses. Each subclass inherits the attributes and methods of the superclass but may have additional unique features.

Diamond Problem

Now, let's discuss the "diamond problem" in the context of multiple inheritance. The diamond problem is a challenge that arises when a class inherits from two or more classes that have a common superclass. It leads to ambiguity in method resolution.

```
class A:
  def test(self):
    print("A")

class B(A):
  def test(self):
    print("B")

class C(A):
  def test(self):
    print("C")
```

```
class D(B, C):
    pass

d = D()
d.test()
```

In this example, class D inherits from classes B and C, which both inherit from class A. When the test() method is called on an instance of D, there is ambiguity about which version of the method to invoke: the one from B or the one from C.

To resolve this ambiguity, Python uses a method resolution order (MRO) algorithm, known as C3 linearization. It determines the order in which the base classes are searched for a method or attribute. The MRO algorithm ensures that each class is visited only once and maintains the order of the inheritance hierarchy.

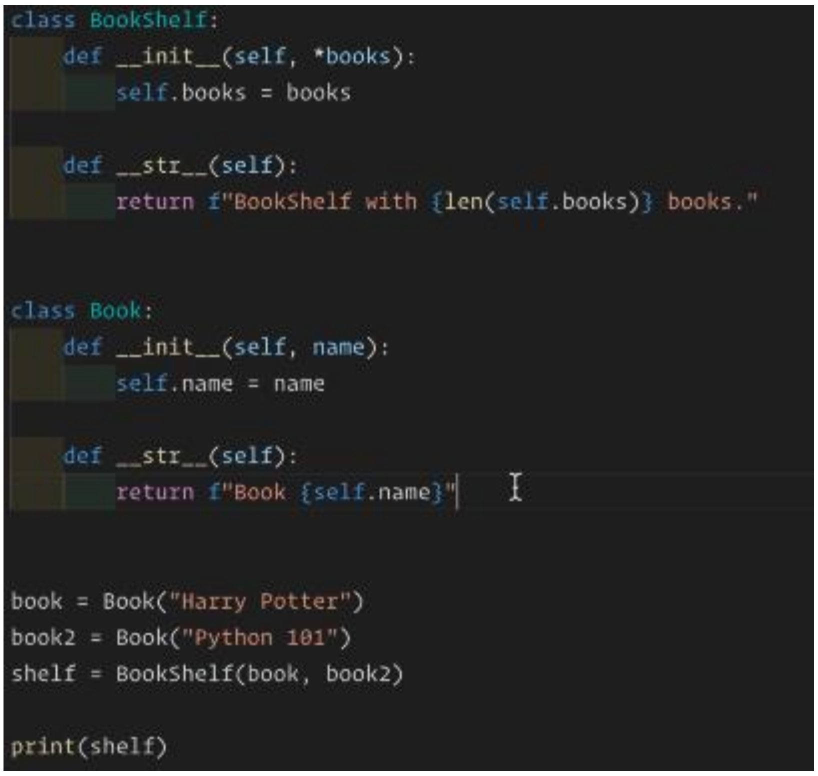
To find the MRO of a class, you can use the mro() method or access the __mro__ attribute. For example:
print(D.mro()) # Output: [<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]

In case of the diamond problem, the MRO algorithm follows a depth-first left-to-right traversal. In the example above, the MRO of class **D** is **[D, B, C, A, object]**. It means that when **test()** is called on **D**, it will first look for the method in **B**, then **C**, then **A**, and finally in the base object class.
By understanding the MRO and using it to design class hierarchies, you can avoid and resolve the diamond problem in Python's multiple inheritance.

Class Composition

Class Composition

- Composition is a counterpart to inheritance so you build out classes that use other classes
- Composition allows your classes to be simpler and reduces the complexity of your code overall

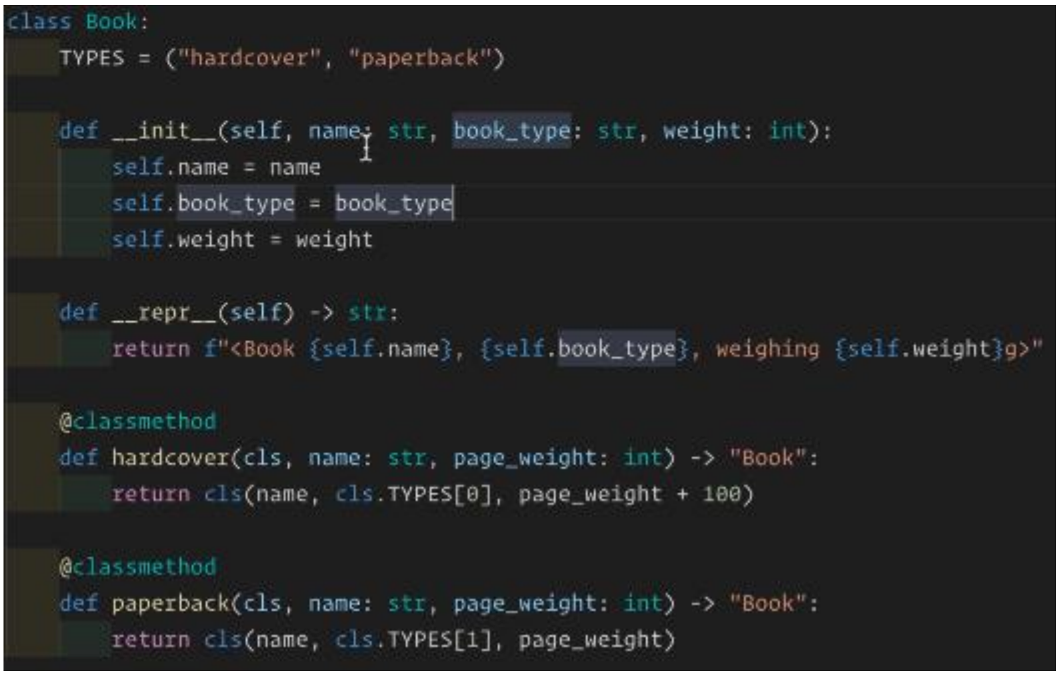


```python
class BookShelf:
    def __init__(self, *books):
        self.books = books

    def __str__(self):
        return f"BookShelf with {len(self.books)} books."


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

    def __str__(self):
        return f"Book {self.name}"


book = Book("Harry Potter")
book2 = Book("Python 101")
shelf = BookShelf(book, book2)

print(shelf)
```

- Inheritance means that a book is a bookshelf (issues with this kind of application for inheritance due to plain logic and the technical fact that you do not use any of the parameters in the inherited class), composition means that a bookshelf has many books (more logical approach).

Type Hinting

Type Hinting (For Python 3.5 and later)



```python
class Book:
    TYPES = ("hardcover", "paperback")

    def __init__(self, name: str, book_type: str, weight: int):
        self.name = name
        self.book_type = book_type
        self.weight = weight

    def __repr__(self) -> str:
        return f"<Book {self.name}, {self.book_type}, weighing {self.weight}g>"

    @classmethod
    def hardcover(cls, name: str, page_weight: int) -> "Book":
        return cls(name, cls.TYPES[0], page_weight + 100)

    @classmethod
    def paperback(cls, name: str, page_weight: int) -> "Book":
        return cls(name, cls.TYPES[1], page_weight)
```

- *from typing import List*

- Type hinting helps you to get told if you pass in the wrong argument formats (examples below)

  def __repr__(self) → str

  def hardcover(cls, name:str, page_weight: int) → "Book"

Decorators

Simple decorators

# When we ran `secure_function`, we checked the user's access level. Because at that point the user was not an admin, the function did not `return func`. Therefore `get_admin_password` is set to `None`.

# We want to delay overwriting until we run the function

```
def get_admin_password():
    return "1234"

def make_secure(func):
    def secure_function():
        if user["access_level"] == "admin":
            return func()

    return secure_function

get_admin_password = make_secure(
    get_admin_password
)  # `get_admin_password` is now `secure_func` from above

user = {"username": "jose", "access_level": "guest"}
print(get_admin_password())  # Now we check access level

user = {"username": "bob", "access_level": "admin"}
print(get_admin_password())  # Now we check access level

# -- More information or error handling --
def get_admin_password():
    return "1234"

def make_secure(func):
    def secure_function():
        if user["access_level"] == "admin":
            return func()
        else:
            return f"No admin permissions for {user['username']}."

    return secure_function
```

```
get_admin_password = make_secure(
    get_admin_password
)  # `get_admin_password` is now `secure_func` from above

user = {"username": "jose", "access_level": "guest"}
print(get_admin_password())  # Now we check access level

user = {"username": "bob", "access_level": "admin"}
print(get_admin_password())  # Now we check access level
```

Using @

```
user = {"username": "jose", "access_level": "guest"}

def make_secure(func):
    def secure_function():
        if user["access_level"] == "admin":
            return func()
        else:
            return f"No admin permissions for {user['username']}."
    return secure_function


@make_secure
def get_admin_password():
    return "1234"
# -- keeping function name and docstring —

import functools
user = {"username": "jose", "access_level": "guest"}


def make_secure(func):
    @functools.wraps(func)
    def secure_function():
        if user["access_level"] == "admin":
            return func()
        else:
            return f"No admin permissions for {user['username']}."

    return secure_function


@make_secure
def get_admin_password():
    return "1234"
```

Decorators with parameters

```
import functools

user = {"username": "anna", "access_level": "user"}

def make_secure(func):
    @functools.wraps(func)
    def secure_function(*args, **kwargs):
        if user["access_level"] == "admin":
            return func(*args, **kwargs)
        else:
            return f"No admin permissions for {user['username']}."
    return secure_function

@make_secure
def get_admin_password():
    return "admin: 1234"


@make_secure
def get_dashboard_password():
    return "user: user_password"


# What if we wanted some passwords to be available to "user" and others to "admin" ?
user = {"username": "anna", "access_level": "user"}

def make_secure(access_level):
    def decorator(func):
        @functools.wraps(func)
        def secure_function(*args, **kwargs):
            if user["access_level"] == access_level:
                return func(*args, **kwargs)
            else:
                return f"No {access_level} permissions for {user['username']}."
        return secure_function
    return decorator

@make_secure(
    "admin"
)  # This runs the make_secure function, which returns decorator. Essentially the same to doing
`@decorator`, which is what we had before.
def get_admin_password():
    return "admin: 1234"

@make_secure("user")
def get_dashboard_password():
    return "user: user_password"

print(get_admin_password())
print(get_dashboard_password())

user = {"username": "anna", "access_level": "admin"}

print(get_admin_password())
print(get_dashboard_password())
```

Generators

A Python generator is a specialized code that can produce a series of values and control the iteration process.

Generators are often called iterators, and they differ from regular functions in that they return a series of values instead of a single well-defined value.

Distinction Between Functions and Generators

- A function returns one well-defined value, invoked only once.
- A generator returns a series of values and can be implicitly invoked multiple times.

The Iterator Protocol

- The iterator protocol defines how an object should behave to work with the context of the **for** and **in** statements.
- An object that conforms to the iterator protocol is called an iterator.
- An iterator must provide two methods: **__iter__()** and **__next__()**.
    - **__iter__()** returns the iterator object itself and is invoked once to start the iteration.
    - **__next__()** is responsible for returning the next value of the desired series and is invoked by the **for/in** statements to iterate.

```
class Fib:
   def __init__(self, n):
      self.__n = n
      self.__i = 0
      self.__p1 = 0
      self.__p2 = 1

   def __iter__(self):
      print("__iter__")
      return self

   def __next__(self):
      print("__next__")
      if self.__i < self.__n:
         if self.__i in [0, 1]:
            result = 1
         else:
            result = self.__p1 + self.__p2
            self.__p1, self.__p2 = self.__p2, result
         self.__i += 1
         return result
      else:
            raise StopIteration

fibonacci = Fib(10)
for num in fibonacci:
    print(num)
```

The yield Statement

- The **yield** statement is a powerful mechanism that turns a function into a generator.
- It provides a value just like **return**, but it doesn't lose the state of the function, allowing it to be resumed from where it left off.

Example:

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

# Example usage:
for v in fun(5):
    print(v)

output 0 1 2 3 4
```

List Comprehension with generators

- Generators can be used within list comprehensions for more compact and elegant code.

  Ex:

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


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

The list() Function with Generators

- The **list()** function can transform a series of subsequent generator invocations into an actual list.

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

Using Generators with the in Operator

- Generators can be used with the **in** operator in loops to print specific values.

```
•  for i in range(20):
•      if i in powers_of_2(4):
•          print(i)
```

List Comprehension in Detail

Used to make lists elegantly

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

Can use conditions as well:

the_list = [1 if x % 2 == 0 else 0 for x in range(10)]

List comprehensions vs. generators

```
the_list = [1 if x % 2 == 0 else 0 for x in range(10)]
the_generator = (1 if x % 2 == 0 else 0 for x in range(10))
```

The brackets make a comprehension, the parentheses make a generator.

Generator output is not a list and you cant directly apply list methods or treat like list.

Lambda Function

- The lambda function is an anonymous function without a name.
- It is used to simplify code and make it clearer and easier to understand.
- The syntax of a lambda function is: lambda parameters: expression.
- Lambdas can be used as anonymous parts of code to evaluate a result.
- They are useful when defining functions that are used only once.

```
# Lambda functions
two = lambda: 2
sqr = lambda x: x * x
pwr = lambda x, y: x ** y

# Using lambda functions
print(two())  # Output: 2
print(sqr(4))  # Output: 16
print(pwr(2, 3))  # Output: 8
```

Lambdas and the map() function

The map() function applies a function to all elements of a list and returns an iterator.
Lambdas can be used as the function argument for map().

```
# Using lambdas with map()
list_1 = [0, 1, 2, 3, 4]
list_2 = list(map(lambda x: 2 ** x, list_1))
print(list_2)  # Output: [1, 2, 4, 8, 16]
```

Lambdas and the filter() function

The filter() function filters elements from a list based on a given condition and returns an iterator.
Lambdas can be used as the condition for filtering in filter().

```
        # Using lambdas with filter()
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even_numbers)  # Output: [2, 4, 6, 8, 10]
```

Lambdas and Reduce function

reduce() should take two arguments and return a single value. It is applied to the first two elements of the iterable, then to the result and the next element, and so on, until the iterable is exhausted.

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
# Compute the sum of the numbers using reduce and a lambda function
sum_result = reduce(lambda x, y: x + y, numbers)
print(sum_result)  # Output: 15
```

Closures

Closures are functions that remember and retain the values of variables even if the context in which they were created no longer exists.

Closures can be created using nested functions.

```
# Creating closures
def outer(par):
   loc = par
   def inner():
      return loc
   return inner
var = 1
fun = outer(var)
print(fun())  # Output: 1
```

Files

Python provides a consistent set of objects and functions to access and process files effectively.

File names

- Different operating systems handle file names differently.
- Windows and Unix/Linux systems have different naming conventions and separators.
- Unix/Linux file names are case-sensitive, while Windows file names are case-insensitive.
- Python's use of the backslash () as an escape character can cause issues in Windows file paths.
- Python can convert slashes to backslashes for Windows file paths automatically.

Examples:

```
try:
   file = open("example.txt", "r")
   content = file.read()
   print(content)
   file.close()
except IOError as e:
   print("Error:", e)
```

File streams

- Files are accessed through abstract entities called streams or handles.
- Streams allow operations on files through a set of functions/methods.
- Streams are connected (opened) to files and disconnected (closed) when finished.
- Opening a stream can fail due to reasons such as non-existent files or permission issues.

File handles

- Files in Python are represented by objects of specific classes.
- Streams are associated with file objects created by the open() function.
- Streams can have different behaviors and operations based on their classes.
- BufferIOBase and TextIOBase are commonly used classes for file streams.
- 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.

  .
- Text mode is the default mode for opening files, can append ‘t’(optional). Binary mode is specified by appending 'b' to the mode string. Text mode processes files as lines, while binary mode operates on byte sequences.

Opening the stream for the first time

- Opening a file for reading is done using the open() function.
- The file name and mode are specified as arguments.
- Exceptions are raised if the file cannot be opened.

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

For example:

```
for line in open("file", "rt"):
    print(line, end='')
```

Pre-opened streams

- Pre-defined streams (stdin, stdout, stderr) don't require explicit opening.
- They are accessible through the sys module (e.g., sys.stdin).

Closing streams

- Closing a stream is the final operation performed on it.
- The close() method is called on the stream object.
- Closing flushes the buffers and frees system resources.
- Streams like stdin, stdout, stderr don't require closing.

Diagnosing stream problems

- IOError objects can provide information about stream errors.
- The errno property of an IOError object contains error codes.
- The errno module provides symbolic constants for common error codes.
- The strerror() function in the os module can be used to get error descriptions.

Reading Text File

Read plain simple text file.

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;

Ex:

```
try:
    stream = open('text.txt', 'rt')
    char = stream.read(1)
    while char:
        print(char, end='')
        char = stream.read(1)
    stream.close()
except IOError as e:
    print("Error:", e)
```

Similarly can use other methods.

Writing Text Files

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;

Ex:

```
try:
    stream = open('newtext.txt', 'w')
    for i in range(1, 11):
        line = 'line #' + str(i) + '\n'
        stream.write(line)
    stream.close()
except IOError as e:
    print("Error:", e)
```

Binary Files

When working with binary files or amorphous data, Python provides the bytearray class to store bytes. Bytearrays can be created explicitly with a specified size and filled with values.

Binary files can be read into a bytearray using the readinto() method, which fills a previously created bytearray with values from the file.

The read() method can also be used to read binary files and returns a bytes object.

```
try:
    stream = open('binary.bin', 'rb')
    data = bytearray(10)
    bytes_read = stream.readinto(data)
    for byte in data:
        print(hex(byte))
    stream.close()
except IOError as e:
    print("Error:", e)
```

OS Module

The os module in Python allows interaction with the operating system.

It provides functions that perform operations available on Unix and/or Windows systems, similar to the commands available on those systems.

- Example: The **mkdir** function in the os module allows creating directories, similar to the **mkdir** command in Unix and Windows.
- The os module enables operations such as:
    - Getting information about the operating system
    - Managing processes
    - Operating on I/O streams using file descriptors
- The **uname** function in the os module provides information about the operating system, including system name, machine name, release, version, and machine hardware identifier.

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).

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)

Examples:

```
import os
print(os.uname())
print(os.name) #For OS Name
os.mkdir('my_first_directory') #Make directory
```

```
print(os.listdir())  #Lists files and directories in current path
os.makedirs('my_first_directory/my_second_directory') #Make recursive dirs.
os.chdir('my_first_directory') #Change directory
print(os.getcwd())#Get current working directory
os.rmdir('my_first_directory')#Remove dir
os.removedirs('my_first_directory/my_second_directory')
```

The system function in the os module executes a command passed as a string and returns the result depending on the system

```
result = os.system('mkdir my_first_directory')
```

DateTime

The datetime module in Python provides classes for working with dates and times.

Date and time are widely used in various applications, such as event logging, tracking changes in the database, data validation, and storing important information.

Date Class

The date class represents a date consisting of the year, month, and day. It has attributes like year, month, and day, which are read-only.

The date class provides methods like today() to get the current local date and fromtimestamp() to create a date object from a timestamp.

The fromisoformat() method can be used to create a date object from a string in the ISO 8601 format.

The replace() method allows replacing specific components of a date object, such as the year, month, or day.

The weekday() method returns the day of the week as an integer (0 for Monday, 6 for Sunday), while the isoweekday() method follows the ISO 8601 specification (1 for Monday, 7 for Sunday).

Examples:

```
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
print("Today:", date.today()) # Displays: Today: 2020-09-29
date_iso = date.fromisoformat('2023-07-07')
date_original = date(2023, 7, 7)
date_replaced = date_original.replace(year=2024, month=8, day=12)
date_object = date(2023, 7, 7)
print(date_object.weekday())  # Output: 5 (Saturday)
print(date_object.isoweekday())  # Output: 6 (Saturday)
```

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)
```

Time Class

- The time class represents time without a date. It has attributes like hour, minute, second, and microsecond.
- The module also includes functions like sleep() to pause program execution and ctime() to convert a timestamp to a string representing the date and time.
- The time class constructor accepts parameters like hour, minute, second, microsecond, tzinfo, and fold.
- The tzinfo parameter is associated with time zones, while fold is associated with wall times.
- The time module provides functions like gmtime() and localtime() to get struct_time objects representing UTC time and local time, respectively.
- The asctime() function converts a struct_time object to a string, and mktime() converts a struct_time object or tuple to the number of seconds since the Unix epoch.

Examples:

```
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
time.sleep(10)#Sleeps for 10 seconds

timestamp = time.time()
formatted_time = time.ctime(timestamp)
print(formatted_time)  # Output: Wed Jul  7 15:30:00 2023

utc_time = time.gmtime() # Get UTC time
print(utc_time)
# Output: time.struct_time(tm_year=2023, tm_mon=7, tm_mday=7, tm_hour=15,
tm_min=30, tm_sec=0, tm_wday=4, tm_yday=188, tm_isdst=0)
```

Datetime

- The datetime class combines date and time information. Its constructor accepts parameters like year, month, day, hour, minute, second, microsecond, tzinfo, and fold.

- The datetime class provides methods like today(), now(), and utcnow() to get the current date and time.
- The timestamp() method can be used to get the number of seconds since the Unix epoch from a datetime object.
- The strftime() method in datetime classes allows formatting date and time as a string using directives.
- Directives are strings starting with % and represent specific components like year, month, day, hour, minute, second, etc.
- The time module also includes a strftime() function for formatting time.
  %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.
- The strptime() method in datetime classes allows parsing a string representing a date and time into a datetime object using a specified format.
- The timedelta class represents a duration or difference between two dates or times.
- Timedelta objects can be created by subtracting two date or datetime objects or by using the timedelta constructor with arguments like days, seconds, microseconds, etc.
- Timedelta objects support operations like addition, subtraction, multiplication with integers, and more.

Examples:

```
from datetime import datetime

dt = datetime(2020, 9, 29, 13, 51)
print("Datetime:", dt) # Displays: Datetime: 2020-09-29 13:51:00
now = datetime.now()
formatted_date = now.strftime("%Y/%m/%d")
formatted_datetime = now.strftime("%Y/%B/%d %H:%M:%S")
print(formatted_date)  # Output: 2023/07/07
print(formatted_datetime)  # Output: 2023/July/07 15:30:00
date_string = "2023/07/07 14:30:00"
date_object = datetime.strptime(date_string, "%Y/%m/%d %H:%M:%S")
print(date_object)  # Output: 2023-07-07 14:30:00

delta1 = timedelta(days=16, hours=2)
delta2 = timedelta(weeks=4, days=2, hours=4)
date = date(2023, 7, 7)
datetime = datetime(2023, 7, 7, 18, 53)

multiplied_delta = delta1 * 2
new_date = date + delta1
new_datetime = datetime + delta2

print(multiplied_delta)  # Output: 32 days, 4:00:00 #Output of delta is always this format
#default
print(new_date)  # Output: 2023-07-23
print(new_datetime)  # Output: 2023-08-10 22:53:00
```

Calendar

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.

```
calendar.setfirstweekday(calendar.SUNDAY)
```

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

```
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:

```
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:

```
# Check if a year is a leap year
```

```
is_leap = calendar.isleap(2020)
print(is_leap)

# Get the number of leap years between 2010 and 2020
leap_years = calendar.leapdays(2010, 2020)
```

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.:

```
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.

9. In addition to functions, the calendar module provides several classes for creating calendars:

- calendar.Calendar: Provides methods to prepare calendar data for formatting.
- calendar.TextCalendar: Used to create regular text calendars.
- calendar.HTMLCalendar: Used to create HTML calendars.
- calendar.LocalTextCalendar: A subclass of calendar.TextCalendar that takes the locale parameter to return appropriate months and weekday names.
- calendar.LocalHTMLCalendar: A subclass of calendar.HTMLCalendar that takes the locale parameter to return appropriate months and weekday names.

The **Calendar** class provides the **itermonthdates** method, which returns an iterator for all days in a specified month and year. It includes days from the previous month and the following month to complete the weeks.

```
for iter in c.itermonthdays(2019, 11):
   print(iter, end=" ")
#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
```

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.