Python Workshop

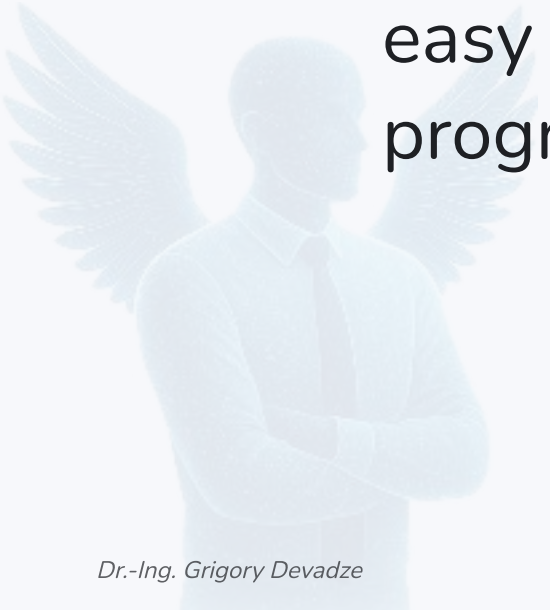
Dr.-Ing. Grigory Devadze

Python

- Python is an interpreted, powerful, and versatile programming language.
- Code or source code: the sequence of instructions in a program, written by a programmer to perform specific tasks.
- Syntax: the set of allowed structures and commands that can be used in a specific programming language. Python's syntax is designed to be especially readable and simple.

- Output: the messages produced by the program for the user, which can include text, numbers, or other information.
- Console: the text field or terminal window where output is displayed and where the user interacts with the program.
- Python programs can be run interactively (one command at a time) or as scripts (a file with a sequence of commands).

- Python is commonly used for web development, data analysis, artificial intelligence, scientific computing, automation, and more.
- The large standard library and active community make it easy to find tools and resources for almost any programming task.



Quick overview of editors

- There are many editors and IDEs (Integrated Development Environments) for Python, depending on preference and use case.
- Popular editors:
  - VS Code: Very popular, free, many extensions for Python, debugging, Git integration.
  - PyCharm: Powerful IDE for Python, Community (free) and Professional (commercial).

- Popular editors:
  - Jupyter Notebook: Ideal for data science, interactive notebooks, visualization and documentation.
  - Thonny: Simple IDE for beginners, especially suitable for teaching.
  - Sublime Text, Atom: Lightweight editors with Python support.
  - Vim, Emacs: Powerful, customizable editors for advanced users.

- Many editors provide:
  - Syntax highlighting
  - Autocompletion
  - Debugging tools
  - Integration with version control (e.g. Git)
  - Extensions/plugins for additional functionality

The help function

- Python provides a built-in help function to retrieve information about modules, functions, classes, and objects directly in the interpreter.
- The `help()` function can be applied to any Python object to display its documentation.
- Especially useful when working interactively or for quick lookup of syntax and parameters.

The help function - example code

```
1 help(print)
2 help(str)
3 import math
4 help(math.sqrt)
```

```
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.
```

The help function

- In the interactive interpreter you can also call `help()` with no argument to start a help menu.
- Many objects also have a special `__doc__` attribute that contains the docstring:

The help function - show docstring contents

```
1 print(print.__doc__)
```

```
Prints the values to a stream, or to sys.stdout by default.

  sep
    string inserted between values, default a space.
  end
    string appended after the last value, default a newline.
  file
    a file-like object (stream); defaults to the current sys.stdout.
  flush
    whether to forcibly flush the stream.
```

- Tip: In Jupyter notebooks, you can append a question mark after a function name, e.g. `print?`, to display help.

Compiling and interpreting

- Many languages require you to compile (translate) your program so the computer can understand it.
- Python is interpreted directly into machine instructions instead.

Difference vs. Java / C#.NET / C++ / PHP

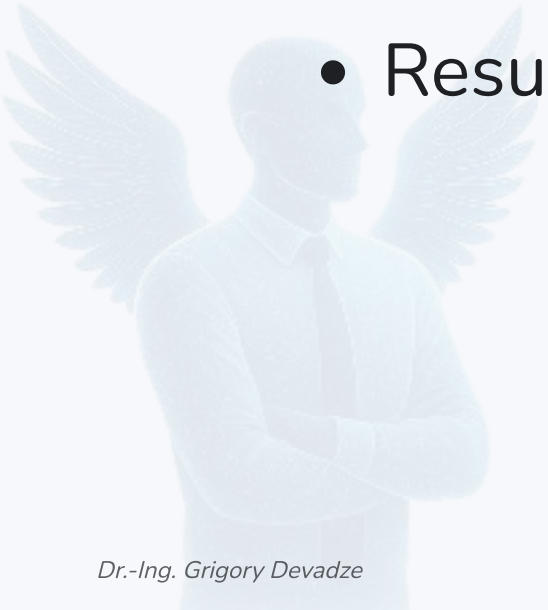
- Python is an interpreted language, while Java, C#.NET, and C++ are compiled.
- PHP: interpreted, but with different syntax and semantics.
- Python does not require explicit type declarations (dynamically typed), while Java, C#, and C++ are statically typed.
- Python uses indentation for block structure, not braces as in Java, C#, C++, or PHP.

Difference vs. Java / C#.NET / C++ / PHP

- Python code is usually shorter and more readable, because many language constructs and the standard library are powerful and easy to use.
- Python supports multiple programming paradigms: procedural, object-oriented, and functional.
- Memory management (garbage collection) in Python is automatic, similar to Java and C#, but unlike C++ (manual).
- Python is platform-independent and is often used for scripting, automation, data science, and web development.

The Python interpreter

- The interpreter provides an interactive environment to experiment with the language.
- Results of expressions are printed to the screen.



```
1 3 + 7
```

```
10
```

```
1 3 < 15
```

```
True
```

```
1 'print me'
```

```
'print me'
```

```
1 print('print me')
```

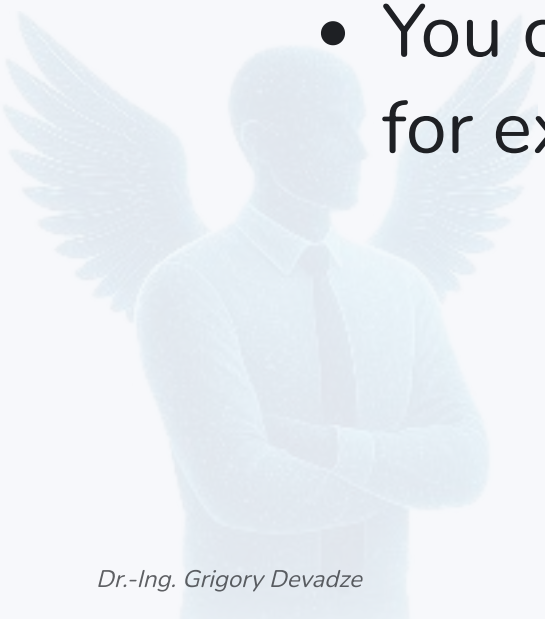
```
print me
```

The Python interpreter

- You can use the Python interpreter in two main ways:
  - Interactive mode: run `python` or `python3` in your terminal and enter commands one by one.
  - Script mode: save your code in a `.py` file and run it with `python filename.py`.

The Python interpreter

- The interpreter evaluates expressions immediately and shows their results.
- You can use the interpreter as a calculator, for testing, or for exploring Python functions.



The Python interpreter

```
$ python3
Python 3.x ...
Type "help", "copyright", "credits" or "license" ...
>>> 2 * 5
10
>>> print("Hello, Python!")
Hello, Python!
>>> exit()
```

- The `>>>` prompt indicates that the interpreter is ready.
- Use `exit()` or press `Ctrl+D` (on Unix/macOS) or `Ctrl+Z` (on Windows) to leave the interpreter.

Expressions and operators

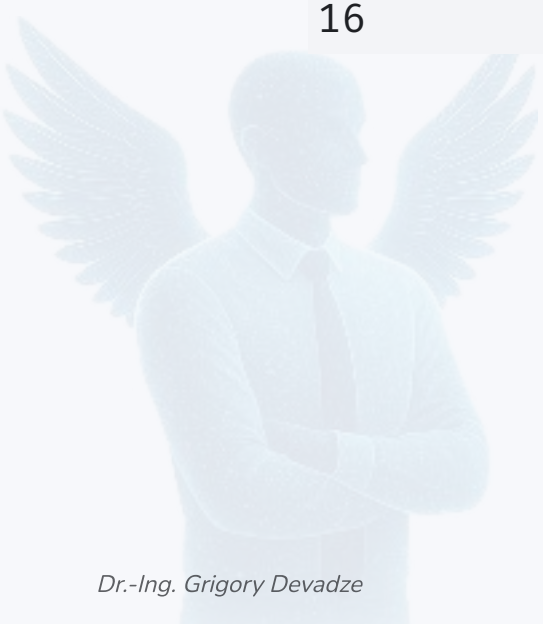
- Expression: a data value or a sequence of operations to compute a value.
- Arithmetic operators: `+, -, *, /, %, **`
- Precedence: `* / % **` have higher precedence than `+` and `-`.
- Parentheses can be used to enforce a specific evaluation order.

```
1 1 + 4 * 3
```

```
13
```

```
1 (1 + 3) * 4
```

```
16
```



Integer division and modulo

- When dividing integers with `/`, the result in Python 3 is always a float.
- The `%` operator computes the remainder of integer division.

```
1 35 / 5
```

```
7.0
```

```
1 84 / 10
```

```
8.4
```

```
1 156 / 100
```

```
1.56
```

```
1 43 % 5
```

```
3
```

Floating-point numbers

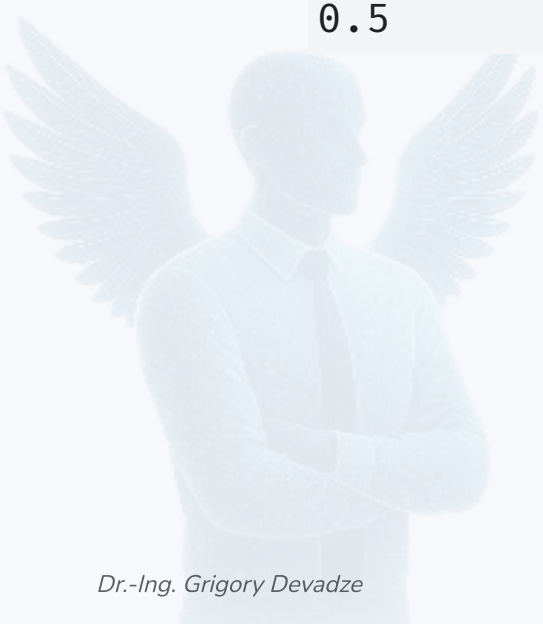
- Python can also work with floating-point numbers (floats).
- The `/` operator returns an exact result (Python 3).
- When integers and floats are mixed, the result is a float.

```
15.0 / 2.0
```

```
7.5
```

```
1 / 2.0
```

```
0.5
```



Mathematical functions

- `abs(x)`: absolute value
- `ceil(x)`: smallest integer >= x
- `floor(x)`: largest integer <= x
- `cos(x)`, `sin(x)`, `tan(x)`: trigonometric functions (x in radians)
- `sqrt(x)`: square root
- `log(x)`: natural logarithm (base e)

Mathematical functions

- `log10(x)`: logarithm base 10
- `pow(x, y)`: x to the power of y
- `max(x, y)`, `min(x, y)`: maximum/minimum of two values
- `round(x)`: nearest integer
- `e`, `pi`: mathematical constants

Mathematical functions - examples

```
 1  from math import *
 2  abs(-5)       # 5
 3  ceil(2.3)     # 3.0
 4  cos(0)        # 1.0
 5  sqrt(16)      # 4.0
 6  floor(2.9)    # 2.0
 7  log(10)       # 2.30258...
 8  log10(100)    # 2.0
 9  pow(2, 3)     # 8.0
10  max(5, 9)     # 9
11  min(5, 9)     # 5
12  round(2.718) # 3
13  e             # 2.718...
14  pi            # 3.141...
```

Variables and assignment

- Variable: a named storage location that can hold a value.
- Assignment statement: stores a value in a variable.
- Variables in Python are dynamically typed, meaning you do not have to declare the type before assignment.

- You can assign any types to a variable (number, string, list, etc.), and the type can change during program execution.
- Variable names should start with a letter or underscore and can contain letters, numbers, and underscores.

Variables and assignment - examples

```
1  x = 5
2  gpa = 3.14
3  name = "Alice"
4  x = x + 4   # 9
5  gpa = gpa * 2   # 6.28
6  x = "hello"     # x is now a string
```

```
1  a = b = c = 0
```

- You can assign the same value to multiple variables at once.

```
1 x, y, z = 1, 2, 3
```

- You can also assign multiple values at once (tuple unpacking).

Output with `print`

- `print`: outputs text to the console.
- The `print` function can output multiple items separated by commas (inserts spaces).
- You can use formatted strings (f-strings, available from Python 3.6+) for more readable and flexible output.
- You can also use the older %-format or `str.format()` for string interpolation.

print - examples

```python
print("Hello, world!")
age = 45
print("You have", 65 - age, "years until retirement")
# With f-strings (Python 3.6+)
print(f"You have {65 - age} years until retirement")
# With str.format()
print("You have {} years until retirement".format(65 - age))
# With %-formatting
print("You have %d years until retirement" % (65 - age))
```

```
Hello, world!
You have 20 years until retirement
You have 20 years until retirement
You have 20 years until retirement
You have 20 years until retirement
```

Input

- `input`: reads a line of user input from the console and returns it as a string. You can convert it to another type if needed.
- The prompt string (if provided) is shown before the input.
- Use `int()`, `float()`, or other conversion functions to process numeric input.

input - examples

```
1 name = input("What is your name? ")
2 print("Hello,", name)
3
4 age = int(input("How old are you? "))
5 print("Your age is", age)
6 print("You have", 65 - age, "years until retirement")
```

For loops

```
for variable in sequence:
    statements
```

- `for` loop: repeats a group of statements over a set of values.
- The `for` loop in Python is used to iterate over sequences (like lists, tuples, strings), ranges, or other iterable objects.
- The variable takes each value from the sequence in turn, and the indented block is executed for each value.

For loops - example with range

```
1 for x in range(1, 6):
2     print(x, "squared is", x * x)
```

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

For loops over lists and strings

- You can use `for` loops to iterate over lists, tuples, strings, dictionaries, and more.

Iterate over a list and a string

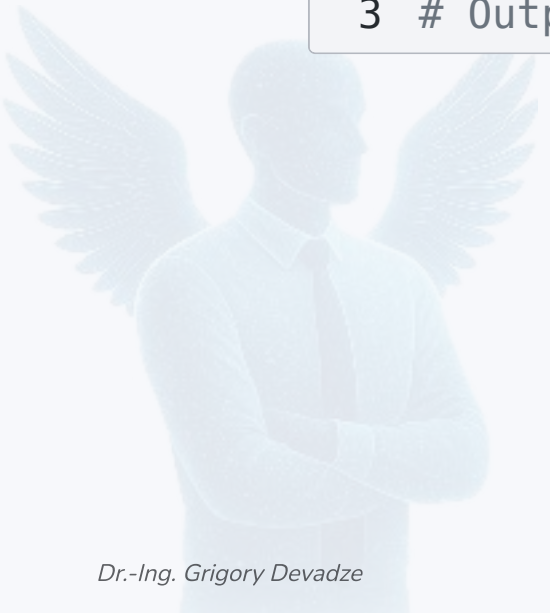
```python
# Iterate over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)

# Iterate over a string
for char in "hello":
    print(char)
```

For loops and `range`

- The `range()` function is commonly used to generate a sequence of numbers for a loop.
- `range(start, stop, step)` generates numbers from `start` (inclusive) to `stop` (exclusive), with an optional step size.

For loops - counting down

```
1 for i in range(10, 0, -2):
2     print(i)
3 # Output: 10 8 6 4 2
```



For + `enumerate`

- Use `enumerate()` to get both the index and the value while iterating over a sequence.

For + enumerate - example

```python
colors = ["red", "green", "blue"]
for idx, color in enumerate(colors):
    print(idx, color)
# Output:
# 0 red
# 1 green
# 2 blue
```

For + `break/continue`

- Use `break` to exit the loop early, and `continue` to jump to the next iteration.

For + break / continue - examples

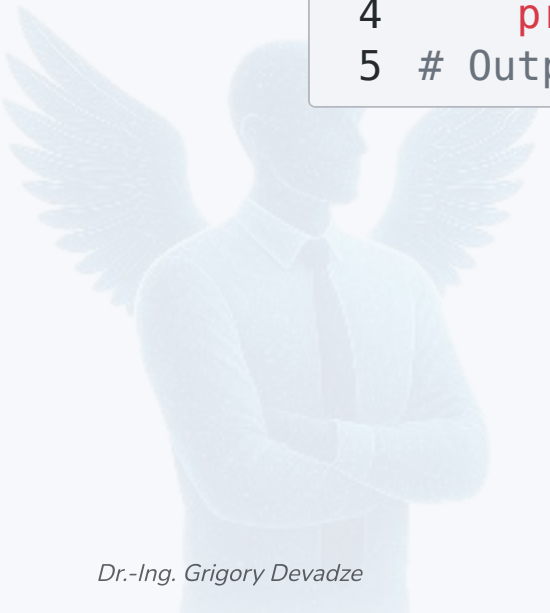
```python
for n in range(1, 10):
    if n == 5:
        break
    print(n)
# Output: 1 2 3 4

for n in range(1, 6):
    if n == 3:
        continue
    print(n)
# Output: 1 2 4 5
```

For with `else`

- You can use else with a `for` loop; the `else` block runs if the loop ends without `break`.

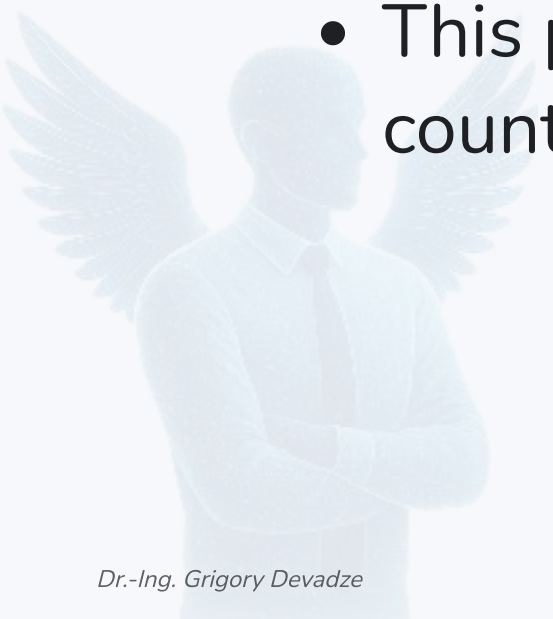
For + else - example

```
1 for n in range(3):
2     print(n)
3 else:
4     print("Loop finished!")
5 # Output: 0 1 2 Loop finished!
```



Accumulating loops

- Some loops compute a value step by step, initialized outside the loop.
- This pattern is useful for summing, multiplying, counting, or other aggregations of values.



Accumulating loops - summation

```
1 # Compute the sum of squares of the first 10 numbers
2 total = 0
3 for i in range(1, 11):
4     total = total + (i * i)
5 print("Sum of the first 10 squares is", total)
```

Accumulating loops - counting

```python
# Count how many even numbers are in a list
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
count = 0
for n in numbers:
    if n % 2 == 0:
        count += 1
print("Number of even values:", count)
```

Accumulating loops - product

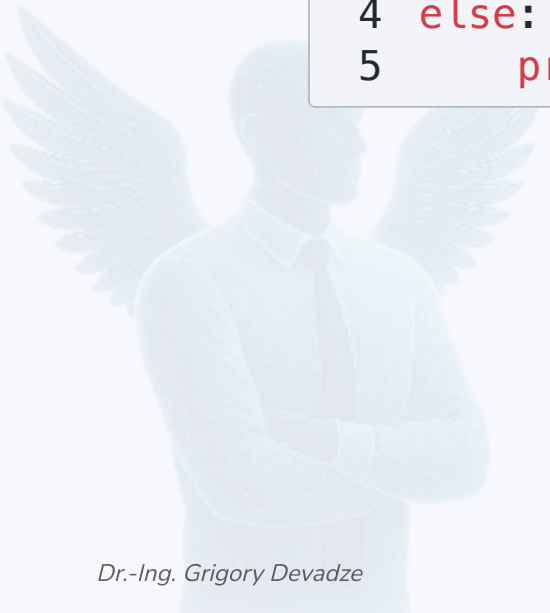
```
1 # Compute the product of a sequence (factorial of 5)
2 product = 1
3 for i in range(1, 6):
4     product *= i
5 print("5! =", product)
```

If and if/else statements

- `if` statement: executes a group of statements only if a condition is true.
- `if/else` statement: executes one block when the condition is true, otherwise another block.
- `elif` (“else if”) allows multiple conditions to be checked in sequence.
- The condition in an `if` statement is an expression that evaluates to True or False.
- Indentation is used to define the blocks that belong to each branch.

If/else - example

```
1 gpa = 1.4
2 if gpa > 2.0:
3     print("Welcome to Mars University!")
4 else:
5     print("Your application was rejected.")
```



If/else with `elif`

```
1 score = 85
2 if score >= 90:
3     print("Grade: A")
4 elif score >= 80:
5     print("Grade: B")
6 elif score >= 70:
7     print("Grade: C")
8 else:
9     print("Grade: D or F")
```

Additional `if` uses

```
1  name = "Alice"
2  if name:
3      print("Name is not empty")
4  if len(name) > 3:
5      print("Name is longer than 3 characters")
```

```
Name is not empty
Name is longer than 3 characters
```

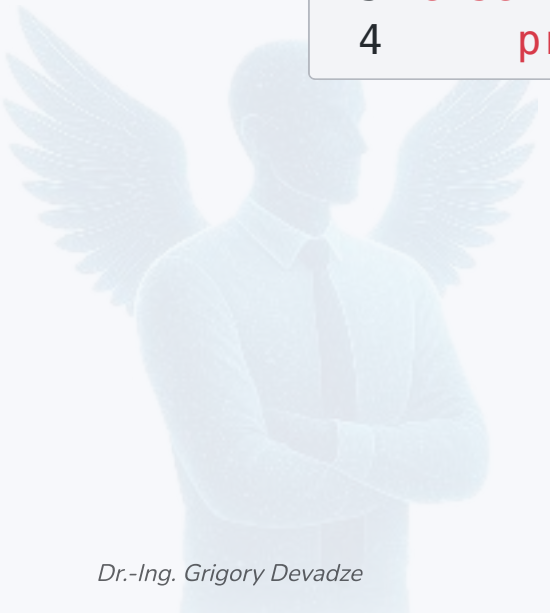
Nested `if` statements

```
1  age = 20
2  if age >= 18:
3      print("You are an adult.")
4      if age >= 21:
5          print("You may drink alcohol in the US.")
6      else:
7          print("You may not drink alcohol in the US.")
8  else:
9      print("You are a minor.")
```

```
You are an adult.
You may not drink alcohol in the US.
```

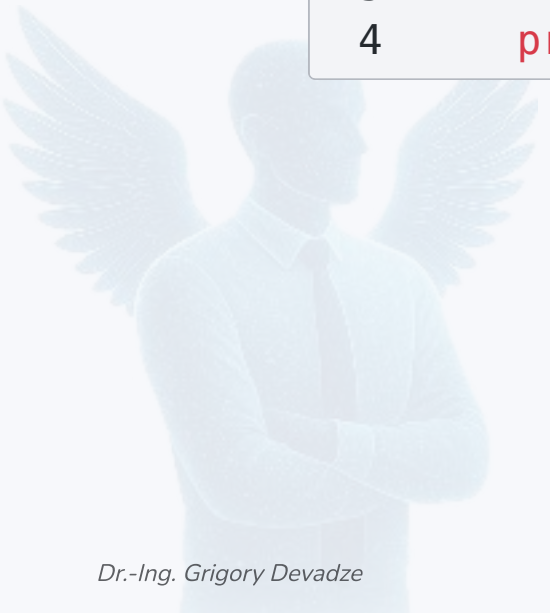
`pass` as a placeholder

```
1 if False:
2     pass  # Do nothing
3 else:
4     print("This is always printed.")
```



Boolean expressions with `and`, `or`, `not`

```
1  x = 5
2  y = 10
3  if x < 10 and y > 5:
4      print("Both conditions are true")
```

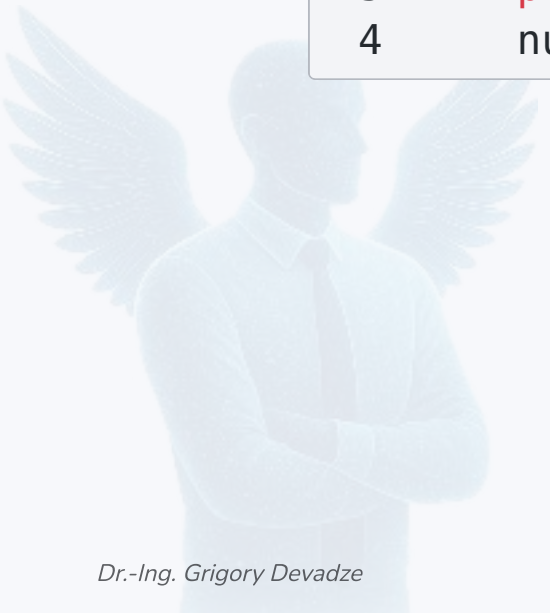


While loops

- `while` loop: executes a group of statements as long as a condition is true.
- The `while` loop is useful when you do not know in advance how many times an action needs to be repeated.
- The loop runs as long as the condition is true; if the condition is false at the start, the loop body never executes.
- Make sure to update variables in the loop so that the condition eventually becomes false - otherwise you get an infinite loop.

While - example

```
1 number = 1
2 while number < 200:
3     print(number)
4     number = number * 2
```

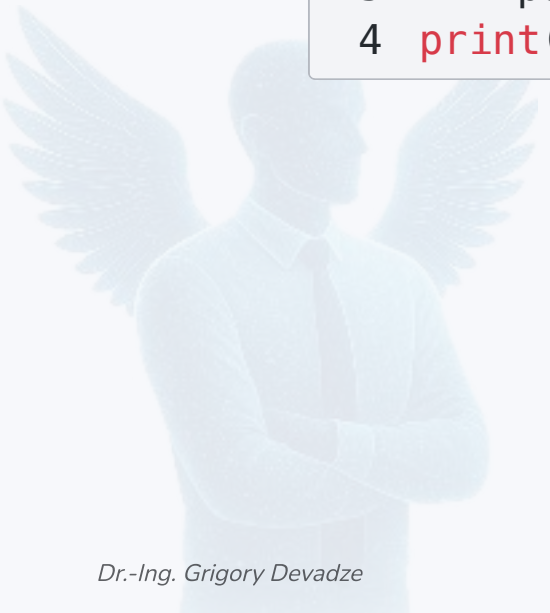


While - summing until stop

```
 1  total = 0
 2  count = 0
 3  print("0 to stop")
 4  num = int(input("Enter a number: "))
 5  while num != 0:
 6      total += num
 7      count += 1
 8      print("0 to stop")
 9      num = int(input("Enter a number: "))
10  print("Count:", count)
11  print("Sum:", total)
```

While - password check

```
1 password = ""
2 while password != "secret":
3     password = input("Enter the password: ")
4 print("Access granted!")
```

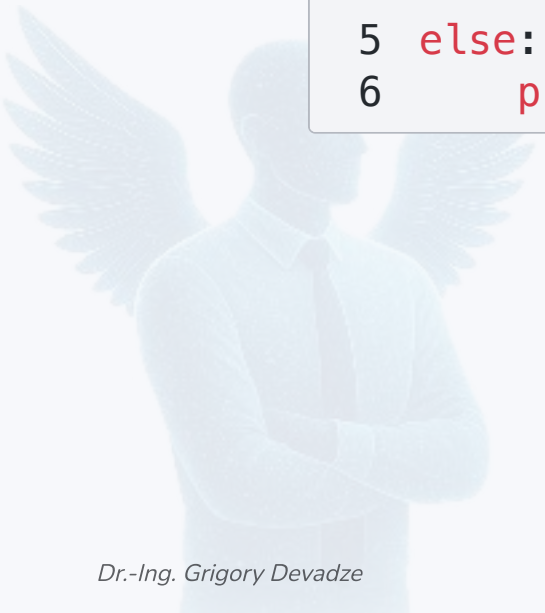


while, break and continue

```
1  i = 0
2  while True:
3      i += 1
4      if i == 5:
5          continue  # 5 is skipped
6      if i > 10:
7          break
8      print(i)
```

While with `else`

```
1 n = 3
2 while n > 0:
3     print(n)
4     n -= 1
5 else:
6     print("Loop finished!")
```



Logic and comparisons

- Comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`
  - `==` : equal, `!=` not equal, `<` less than, `>` greater than, `<=` less than or equal, `>=` greater than or equal
- Logical operators: `and`, `or`, `not`
  - `and`: True if both sides are True
  - `or`: True if at least one side is True
  - `not`: negates the truth value

Logic and comparisons

- These operators are used to build boolean expressions that are essential for controlling program flow with `if`, `while`, and other statements.

Logic - examples

```
 1 x,y = 5,8
 2 # Comparison operators
 3 print(x == 5)    # True
 4 print(y != 10)   # True
 5 print(x < y)     # True
 6 print(x >= 0)    # True
 7 # Logical operators
 8 if x > 0 and y < 10:
 9     print("x is positive and y is less than 10")
10 if x < 0 or y < 10:
11     print("At least one variable is less than 10")
12 if not (x == 0):
13     print("x is not zero")
```

```
True
True
True
True
x is positive and y is less than 10
At least one variable is less than 10
x is not zero
```

Logic - complex expressions and list filtering

```python
a = 3
b = 7
c = 10
if (a < b and b < c) or (a == 3 and c == 10):
    print("Complex condition is True")

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = [n for n in numbers if n % 2 == 0]
print("Even numbers:", even_numbers)
```

```
Complex condition is True
Even numbers: [2, 4, 6, 8, 10]
```

Logic - truthiness of empty/non-empty lists

```
1 if []:
2     print("This will not be printed")
3 if [1, 2, 3]:
4     print("This will be printed")
```

```
This will be printed
```

Lists

- Lists are ordered collections of data, mutable (can be changed after creation) and can contain elements of different types.
- Lists are one of the most commonly used data structures in Python.
- Lists are defined with square brackets `[]`, and elements are separated by commas.

Lists

- Lists can be indexed, sliced, iterated, and modified.
- Lists can contain any objects, including other lists (nested lists).

Lists - basic operations

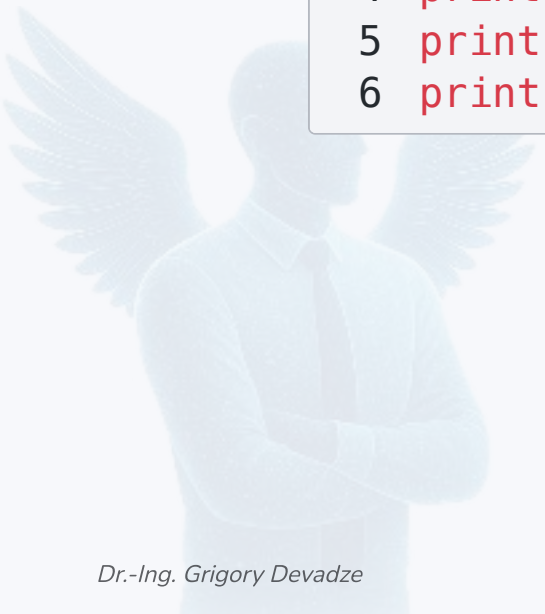
```
1 x = [1, 'hello', (3 + 2j)]
2 x[2]      # (3+2j)
3 x[0:2]    # [1, 'hello']
4 x.append(12)  # Add 12 at the end
5 print(x)      # [1, 'hello', (3+2j), 12]
```

Lists: indexing, slicing

- Indexing starts at 0. Negative indices count from the end.
- Slicing: `list[start:stop]` returns a new list from index start up to, but not including, stop.

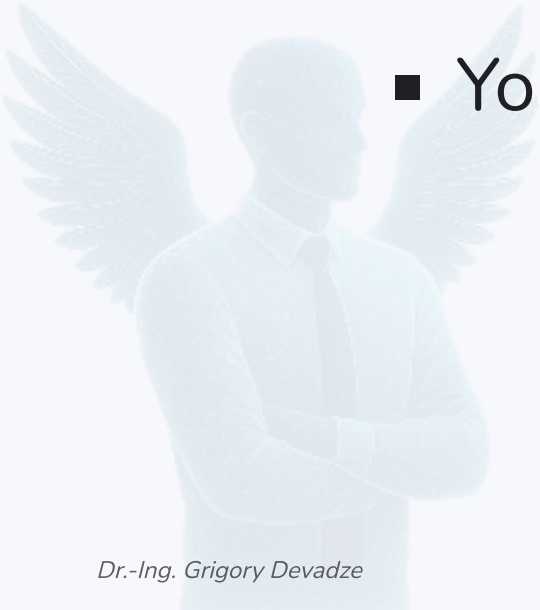
Lists - indexing and slicing

```
1 numbers = [10, 20, 30, 40, 50]
2 print(numbers[0])    # 10
3 print(numbers[-1])   # 50
4 print(numbers[1:4])  # [20, 30, 40]
5 print(numbers[:3])   # [10, 20, 30]
6 print(numbers[3:])   # [40, 50]
```



Lists: mutability

- Lists are mutable:
  - You can change elements via assignment.
  - You can add, remove, or modify elements.



Lists - mutable, methods

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "blueberry"   # Change an element
fruits.append("date")     # Add at the end
fruits.insert(1, "kiwi")  # Insert at index 1
print(fruits)             # ['apple', 'kiwi', 'blueberry', 'cherry', 'date'
fruits.remove("kiwi")     # Remove by value
item = fruits.pop()       # Remove and return last item
print(item)               # 'date'
print(fruits)             # ['apple', 'blueberry', 'cherry']
```

Lists - important methods

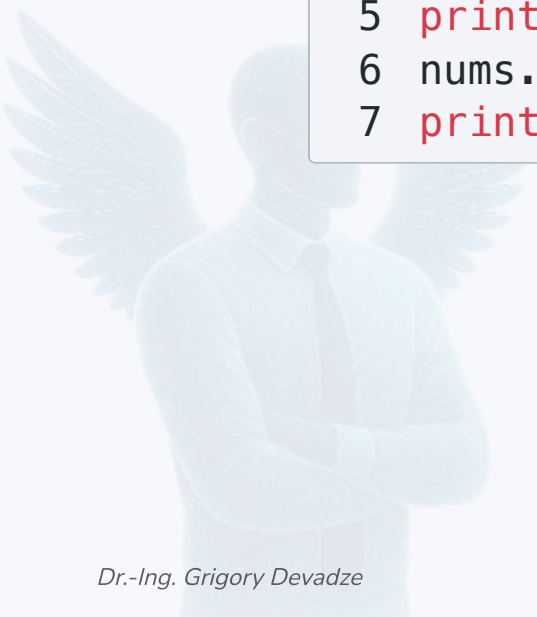
- `len(list)`: number of elements
- `list.append(x)`: adds x at the end
- `list.insert(i, x)`: inserts x at index i
- `list.remove(x)`: removes the first occurrence of x
- `list.pop([i])`: removes and returns the element at index i (default: last)
- `list.index(x)`: returns the index of the first occurrence of x

Lists - important methods

- `list.count(x)`: counts occurrences of x
- `list.sort()`: sorts the list in-place
- `list.reverse()`: reverses the order
- `list.copy()`: returns a shallow copy
- `list.clear()`: removes all elements

Lists - sorting, reversing, copy

```
1 nums = [3, 1, 4, 1, 5, 9, 2]
2 print(len(nums))      # 7
3 print(nums.count(1))  # 2
4 nums.sort()
5 print(nums)  # [1, 1, 2, 3, 4, 5, 9]
6 nums.reverse()
7 print(nums)  # [9, 5, 4, 3, 2, 1, 1]
```



Lists - list comprehension

```python
squares = [x * x for x in range(1, 6)]
print(squares)  # [1, 4, 9, 16, 25]

even_numbers = [n for n in range(10) if n % 2 == 0]
print(even_numbers)    # [0, 2, 4, 6, 8]
```

Lists - nesting, copying

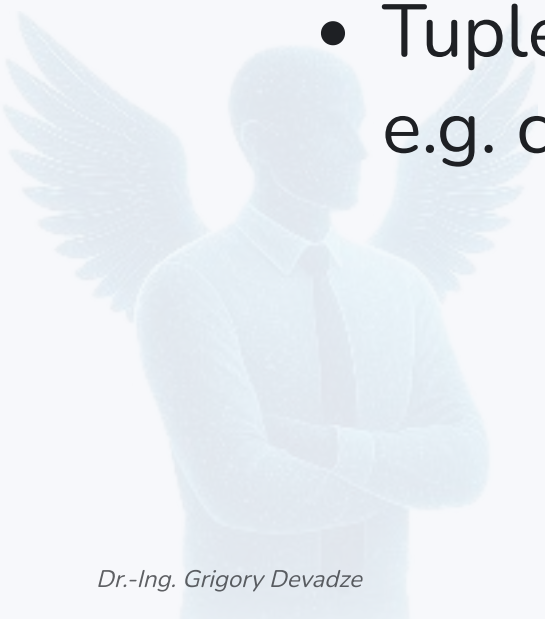
```python
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print(matrix[1][2])  # 6

import copy
a = [[1, 2], [3, 4]]
b = a.copy()          # Shallow copy
c = copy.deepcopy(a) # Deep copy
a[0][0] = 99
print(b)  # [[99, 2], [3, 4]]
print(c)  # [[1, 2], [3, 4]]
```

Tuples

- Tuples are immutable versions of lists.
- After creation, the elements of a tuple cannot be changed, added, or removed.
- Tuples are defined with parentheses `( )`, and elements are separated by commas.

Tuples

- Tuples can contain elements of different types, including other tuples or lists.
- Tuples are often used for fixed collections of values, e.g. coordinates, RGB colors, or function return values.



Tuples - examples

```python
x = (1, 2, 3)
y = (2,)
z = ("apple", 3.14, [1, 2, 3])
print(x[1])      # 2
print(x[:2])     # (1, 2)
a, b, c = x
print(a, b, c)   # 1 2 3
t = (1, (2, 3), 4)
print(t[1][0])   # 2
```

Tuples - error on modification

```
1 x = (1, 2, 3)
2 # x[0] = 10  # TypeError!
```

Tuples - as dictionary keys, function return values

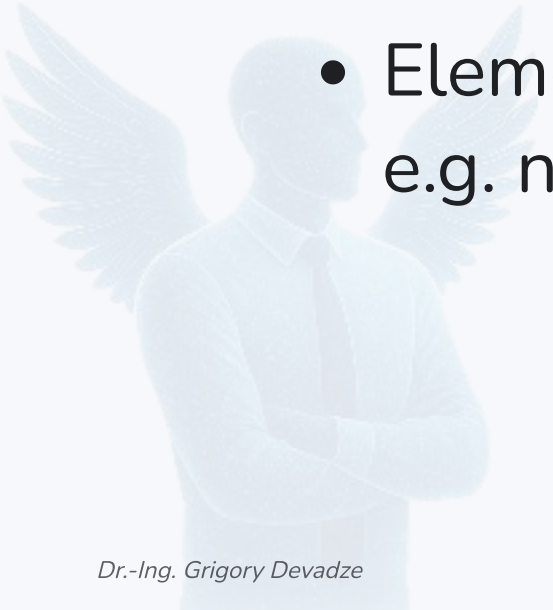
```
locations = { (52.5, 13.4): "Berlin", (48.8, 2.3): "Paris" }
print(locations[(52.5, 13.4)])  # Berlin

def min_max(numbers):
    return min(numbers), max(numbers)
lo, hi = min_max([1, 2, 3, 4])
print(lo, hi)  # 1 4

lst = [1, 2, 3]
tup = tuple(lst)
lst2 = list(tup)
```

Sets

- Sets are unordered collections of unique elements (no duplicates).
- Sets are mutable: you can add or remove elements after creation.
- Sets are defined with the `set()` constructor or curly braces `{}` (for empty sets, you must use `set()`).

Sets

- Sets are useful for membership checks, removing duplicates from sequences, and for mathematical set operations.
- Elements of a set must be immutable (hashable), e.g. numbers, strings, tuples (no lists or other sets).



Sets - examples and operations

```python
setA = set(["a", "b", "c", "d"])
setB = set(["c", "d", "e", "f"])
setC = {"x", "y", "z"}  # Set literal
empty_set = set()
print("a" in setA)      # True
print("e" in setA)      # False

print(setA - setB)  # {'a', 'b'}
print(setA | setB)  # {'a', 'b', 'c', 'd', 'e', 'f'}
print(setA & setB)  # {'c', 'd'}
print(setA ^ setB)  # {'a', 'b', 'e', 'f'}

setA.add("z")
setA.remove("a")
setA.discard("not_in_set")  # No error
print(setA)

for item in setA:
```

Dictionaries

- Dictionaries are mutable collections of key-value pairs.
- Keys must be unique and immutable (e.g. strings, numbers, tuples); values can be of any type.
- Dictionaries are defined with curly braces `{}` and key:value pairs, or with the `dict()` constructor.
- Dictionaries are often used for fast lookups, configurations, counting, and more.

Dictionary - examples

```python
d = {'one': 1, 'two': 2, 'three': 3}
print(d['three'])  # 3
d['two'] = 99      # Change the value for key 'two'
d['four'] = 4      # Add a new key-value pair
print(d)
# {'one': 1, 'two': 99, 'three': 3, 'four': 4}

print(d.get('five', 'not found'))  # not found

del d['one']
value = d.pop('three')
print(d)      # {'two': 99, 'four': 4}
print(value)  # 3

if 'two' in d:
    print("Key 'two' exists!")
```

```
3
{'one': 1, 'two': 99, 'three': 3, 'four': 4}
not found
{'two': 99, 'four': 4}
3
Key 'two' exists!
```

Functions

- Functions are defined with the `def` keyword and can have parameters and return values.
- Functions help structure code, avoid repetition, and make programs more readable and maintainable.
- The function body is indented, and a function can use the `return` keyword to return a value.

Functions

- Parameters are variables in the function definition; arguments are the values you pass to the function.
- Functions can have default parameter values, variable argument lists, and keyword arguments.
- Functions can return multiple values as a tuple.
- Docstrings (triple-quoted strings) can be used to document functions.

Function - find maximum

```python
def maximum(x, y):
    """Return the maximum of x and y."""
    if x < y:
        return y
    else:
        return x

def greeting(name="World"):
    print(f"Hello, {name}!")

greeting()
greeting("Markus")
```

```
Hello, World!
Hello, Markus!
```

Multiple return values, *args, **kwargs

```
 1
 2 def min_max(numbers):
 3     """Return the minimum and maximum of a list."""
 4     return min(numbers), max(numbers)
 5
 6 def add_all(*args):
 7     """Return the sum of all arguments."""
 8     return sum(args)
 9
10 def print_info(**kwargs):
11     for key, value in kwargs.items():
12         print(f"{key}: {value}")
```

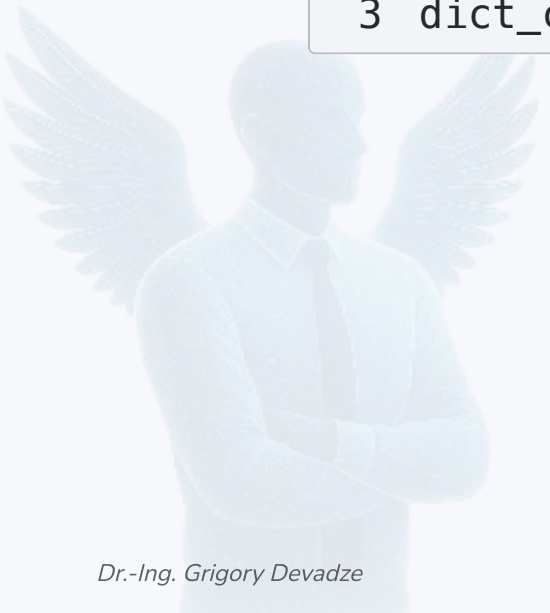
Higher-order functions (`lambda`)

```python
square = lambda x: x * x
print(square(5))
def double(x): return 2 * x
lst = list(range(10))
print(list(map(double, lst)))    # [0, 2, 4, ..., 18]
print(list(map(lambda x: x ** 2, lst)))
print(list(filter(lambda x: x > 5, lst)))
from functools import reduce
print(reduce(lambda x, y: x + y, lst))  # 45
```

```
25
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[6, 7, 8, 9]
45
```

List vs. dictionary comprehension

```
1 squares = [x ** 2 for x in lst]
2 even_numbers = [x for x in lst if x % 2 == 0]
3 dict_comp = {x: x*x for x in range(5)}
```

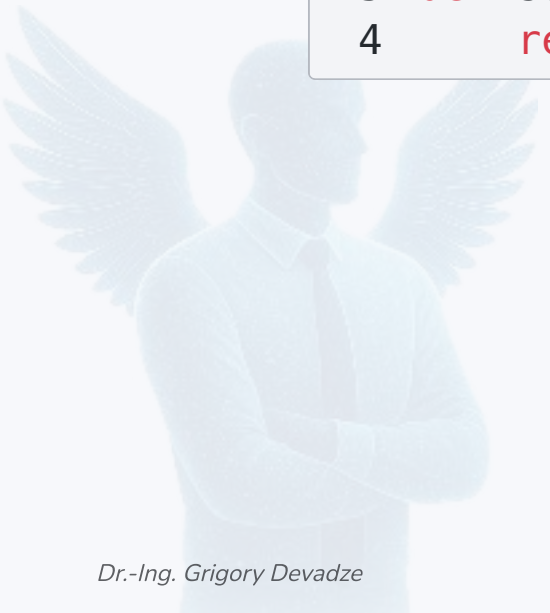


Typing and type hints in Python

- Python is dynamically typed: the type of variables is determined at runtime.
- With *type hints* (type annotations), you can optionally specify the type of variables, lists, and function parameters.
- Type hints are hints, but are not checked by Python itself at runtime.
- For static checks, tools like `mypy` can be used.

Type hints - examples

```python
from typing import List

def squares(values: List[int]) -> List[int]:
    return [x ** 2 for x in values]
```



Type hints - Python 3.9+ shorthand

```
1 def double(values: list[int]) -> list[int]:
2     return [2 * x for x in values]
```

Type hints - type checking and runtime

```
1 def foo(a: int) -> int:
2     return a + 4
3
4 foo('seven')  # mypy: error, Python at runtime: accepts it!
```

- Python does not check type annotations at runtime.
- Static tools like mypy can find errors.

Runtime type checking

```
1 x = [1, 2, 3]
2 if isinstance(x, list):
3     print("x is a list!")
```

- You can check types at runtime with `isinstance(obj, type)`:

Classes and objects

- A class defines what objects look like and which functions operate on them.
- An object is an instance of a class with its own data and behavior.
- The first parameter of each method in a class is `self`, which refers to the instance itself.
- Classes can have attributes (fields) and methods (functions inside the class).
- The method `__init__` is the constructor and is called when a new instance is created.

Simple class - example

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance_to_origin(self):
        return (self.x**2 + self.y**2) ** 0.5

p = Point(3, 4)
print(p.distance_to_origin())
```

```
5.0
```

Object-oriented features

- Methods are called with dot notation: `object.method(arguments)`.
- The `__str__` method defines how the object is printed as a string.
- Python does not enforce access restrictions (like private/protected in other languages), but a leading underscore (e.g. `_x`) is a convention for “internal”.

Object-oriented features

- All classes inherit from `object` by default.
- Use `type(obj)` to check the type of an object, and `isinstance(obj, ClassName)` to check membership in a class.

Class with str and special methods

```python
class Fraction:
    def __init__(self, z, n):
        self.z = z
        self.n = n
    def __str__(self):
        return f"{self.z}/{self.n}"
    def __add__(self, other):
        return Fraction(self.z * other.n + other.z * self.n,
        self.n * other.n)
    def __eq__(self, other):
        return self.z * other.n == other.z * self.n
b1 = Fraction(1, 2)
b2 = Fraction(1, 3)
print(b1 + b2)
print(b1 == b2)
```

```
5/6
False
```

Static methods and dynamism

```
1  class MathUtils:
2      @staticmethod
3      def midpoint(x1, y1, x2, y2):
4          return ((x1 + x2) / 2, (y1 + y2) / 2)
5
6  print(MathUtils.midpoint(0, 0, 4, 4))
```

```
(2.0, 2.0)
```

Inheritance

- Inheritance allows a class (subclass) to take on attributes and methods of another class (superclass).
- This enables code reuse, extension, and specialization of behavior.
- Syntax: `class SubclassName(SuperclassName):`
- The subclass can override or extend methods and attributes of the superclass.
- You can call superclass methods with `super()`.

Inheritance

- Multiple inheritance: `class C(A, B):`
- Use `issubclass(Subclass, Superclass)` to check whether a class inherits from another.
- Use `isinstance(obj, ClassName)` to check whether an object is an instance of a class or its subclasses.

Inheritance - example

```
1 class Animal:
2     def speak(self):
3         print("An animal makes a sound")
4 class Dog(Animal):
5     def speak(self):
6         print("Woof!")
7 h = Dog()
8 h.speak()  # Woof!
```

```
Woof!
```

Multiple inheritance/MRO

```python
class A:
    def foo(self):
        print("A.foo")
class B:
    def bar(self):
        print("B.bar")
class C(A, B):
    pass

c = C()
c.foo()   # A.foo
c.bar()   # B.bar
print(C.mro())
```

```
A.foo
B.bar
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class
'object'>]
```