Introduction to Python

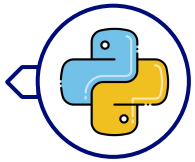
-

PALISSON Antoine

Table of **Contents**

Table of **Contents**

01



Introduction

Introduction to Python

Definition

Python is a high-level interpreted programming language created by Guido van Rossum and first released in 1991.

Python's syntax is designed to be straightforward and human-readable as well as more flexible than other programming language, making it an ideal language for beginners. However, Python will generally run more slowly than compiled programming language because of its interpreted nature.

*A **programming language** is like a set of instructions/rules that humans can use to tell a computer what to do. It is a way to communicate with the computer, using a special language that both you and the computer can understand.*

*A **compiled language** translates the source code all at once by a compiler into a language that the computer can understand, whereas an **interpreted language** translates the source code on the fly, line by line, as the program runs using an interpreter.*

Source : Wikipedia

Guido Van Rossum

General-Purpose Language

General-purpose languages, like **Python**, are crafted to handle a wide range of applications across diverse domains. In contrast, **domain-specific languages** (DSLs) such as **SQL** and **HTML** are honed for particular tasks, like database queries and web content structuring, respectively.

This specialization allows DSLs to offer a steeper learning curve, letting individuals grasp their intricacies quickly. However, while you could mold Python to suit various needs, from web development to scientific computing, a DSL remains confined to its core purpose; one wouldn't, for instance, turn to SQL to craft a mobile application.

Versatility

Python's clear syntax and extensive standard libraries make it adaptable for a wide range of tasks without being specialized for any single one.

Extensibility

Python can be combined with other languages (e.g., C, C++, Java) to enhance its capabilities. This ensures that Python can harness the strengths of specialized languages when needed.

Readability

Emphasizing simplicity, Python's syntax promotes clear coding practices. This readability fosters easier collaboration among developers from diverse backgrounds.

Prototyping

Python's interpreted nature and high-level constructs enable quick iteration and prototyping, which is invaluable in research and development phases.

Applications

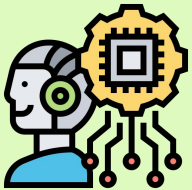
Web Development

Data Science

Software Development

Game Development

Artificial Intelligence



Automation

Python's Readability

Python

```python
import math

radius = 5
area = math.pi * radius**2

print("The area of the circle is", area)
```

C++

```cpp
#include <iostream>
#include <cmath>

int main() {
  double radius = 5;
  double area = M_PI * std::pow(radius, 2);

  std::cout << "The area of the circle is " << area << std::endl;
  return 0;
}
```

Java

```java
public class CircleArea {
  public static void main(String[] args) {
    double radius = 5;
    double area = Math.PI * Math.pow(radius, 2);

    System.out.println("The area of the circle is " + area);
  }
}
```

Python’s Philosophy

Python's philosophy is underpinned by a set of guiding principles known as the “Zen of Python”. Written by Tim Peters in 2004, these aphorisms encapsulate the spirit and design considerations behind the Python language.

The Zen of Python consists of 19 aphorisms. However, the 20th one is intentionally left blank by Tim Peters as an inside joke, emphasizing that there's always one more principle or something more to add.

So, while there are technically 20 slots, there are 19 written guiding principles in the Zen of Python.

Beautiful is better than ugly.

Explicit is better than implicit.

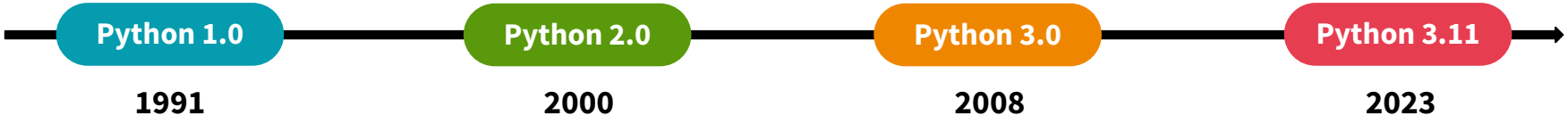
Simple is better than complex. Complex is better than complicated.

There should be one, and preferably only one, obvious way to do it

In the face of ambiguity, refuse the temptation to guess

and more …

Python's Versions



*Python 1 marked the beginning of a language designed for **readability**, **simplicity**, and **ease of use**, aiming to strike a balance between low-level languages (like C) and shell scripting.*

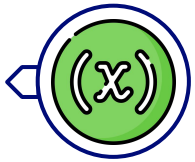
*Python 2 made programming in Python **smoother** and **more efficient**. It vastly expanded and **enriched its toolkit**, making a wider range of tasks achievable right out of the box.*

*Python 3 aimed to **correct inconsistencies and design flaws**, even if it meant breaking backward compatibility. It eliminated outdated ways of doing things, **promoting more modern and intuitive methods**.*

*Python 3's subsequent versions have consistently refined the language. This period has seen Python embrace modern programming paradigms, **expand its library**, **enhance performance**, and **adapt to user needs and global tech trends**.*

***PEPs** (Python Enhancement Proposals) are design documents that provide information or describe a new feature for Python or its processes and environment. They propose major changes or enhancements to the language. You can oversee the next major changes for Python by looking at the Open PEPs, the Accepted PEPs and the Finished PEPs.*

02

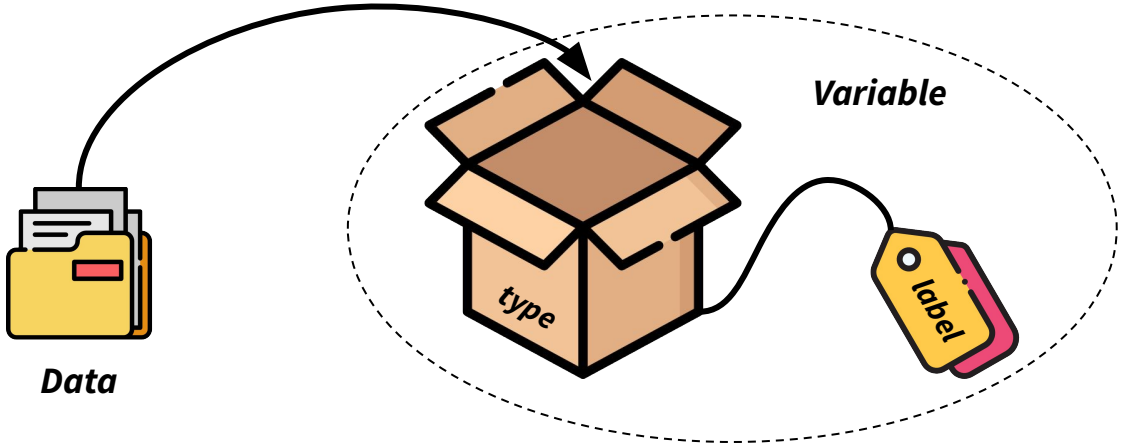


Variables and Types

Introduction to Python

Variables

In **programming**, a **variable** is like a labeled box that can hold a specific value at a time. You can put something inside this container, take it out, or replace it with something else, and the variable's name always refers to the current content.



For example, you can have a variable called "age" that stores the number 25. Later in your program, you can use this variable to perform calculations or make decisions.

While programming variables store specific values and can be changed during the execution of a program, **mathematical variables** are more abstract and represent an undefined or changeable quantity within a mathematical expression or equation.

Variable Names

Python has some rules and best practices when it comes to naming these labels or variables:

Start with a Letter or Underscore

Variable names **must begin with a letter** (a-z, A-Z) **or an underscore** (_). *For example, "name" and "_private_var" are valid, but "1name" is not.*

After the first character, variable names can consist of letters, numbers, and underscores.

No Spaces

Variable names **cannot have spaces**. So, user name would be incorrect. Instead, use underscores to separate words, like user_name.

Case Sensitive

Python differentiates between uppercase and lowercase letters, so "**U**sername" and "**u**sername" are considered distinct variables.

Don't Use Reserved Words

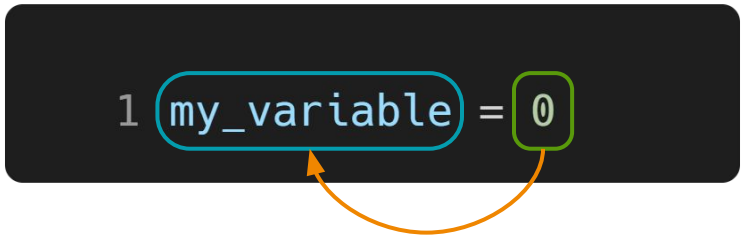
Python has specific words, known as keywords, reserved for its own functions, such as "if", "else", "list" and "print". You should not use these as variable names, as they have special meanings in the language.

Declaring a Variable

Declaring a variable in programming is setting a name, on the box. It tells the computer what to call a particular piece of information. Later on, a value can be stored in the variable, changed, or used in other parts of the program.

Unlike some other programming languages, in Python, you don't need a special command or keyword to declare a variable. You just start using it! You declare a variable by simply giving it a name and then assigning a value to it using the **equal sign (=)**.

Imagine you have a set of empty boxes, and you want to use one to store your favorite toy. Before putting the toy in the box, you put a label on the box with a name like "MyToy." Now, everyone knows that this particular box contains your favorite toy and can refer to it by that name.



The value 0 has been assigned to the variable named “my_variable” using the equal sign (=).

Variable Types

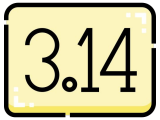
In **programming**, a **type** refers to the category of data that tells the computer how to handle that data. There is 4 main types of variables:

Integers

Represents whole numbers without any decimal points.

```
1 my_integer = 3
```

Floating Points



Numerical values that include decimals

```
1 my_float = 3.14159
```

Strings

A sequence of characters, essentially text.

```
1 my_string = "apple"
```

Booleans

Can hold only two values: True or False. They're like light switches, either ON (True) or OFF (False).

```
1 my_bool_1 = True
2 my_bool_2 = False
```

Dynamic Typing

Dynamic typing refers to a language's ability to determine the data type of a variable at runtime, rather than requiring the programmer to specify it beforehand. In simpler terms, with dynamic typing, the type of data a variable holds can change over time, and the language will adapt on-the-fly. Python figures out whether your variable is, for example, a number, text, or something else, all by itself.

Runtime ???

This is the time when you run or execute your program. It's different from "compile-time", which is when some languages (not Python, since it's an interpreted language) transform code into a format that a computer can understand and then execute.

```
1 age = 25
2 age = "twenty-five"
```

The age variable has been assigned to two different types of values:

- *At line 1 - to 25, an integer.*
- *At line 2 - then to “twenty-five”, a string.*

Built-in Functions in Python

Built-in functions are fundamental tools in Python that you can use without having to do any setup. Think of them as basic commands that Python understands out of the box. They are always available whenever you're using Python, so you don't have to "turn them on" or "get them" from somewhere else.

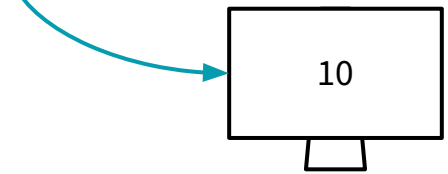
*To use a built-in function, you simply **call its name followed by parentheses ()**. Inside these parentheses, you can place the data or value you want the function to act on. This is known as passing an argument to the function.*

For example, in print("Hello!"), "Hello!" is the argument we're passing to the print function.

The **print built-in** function displays text or data to the screen.

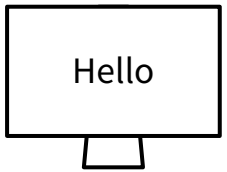
It can print a variable value

```
1 my_var = 10
2 print(my_var)
```



Or it can directly print a value

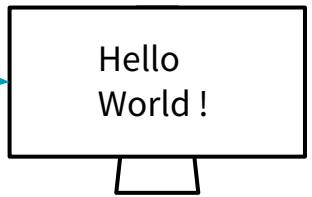
```
1 print("Hello")
```



Built-in Functions in Python - Print

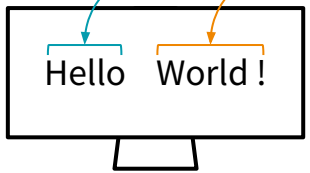
Print add a line break after its output by default.

```
1 var_1 = "Hello"
2 var_2 = "World !"
3 print(var_1)
4 print(var_2)
```



You can add multiple arguments in the print function by separating by commas.

```
1 var_1 = "Hello"
2 var_2 = "World!"
3 print(var_1, var_2)
```



Print always add a space between multiple variables.

```
print(var_1, var_2, var_3, …)
```

Type Function

The **type()** function is used to find out what kind of data something is. In Python, everything is an object, and objects have types. Knowing the type of data you're dealing with is crucial, as certain operations might only be valid for specific data types.

```
1 my_var = 5
2 var_type = type(my_var)
3 print(var_type)
```

<class 'int'>

```
1 my_var = "hello"
2 var_type = type(my_var)
3 print(var_type)
```

<class 'str'>

```
1 my_var = 3.14
2 var_type = type(my_var)
3 print(var_type)
```

<class 'float'>

Objects ???

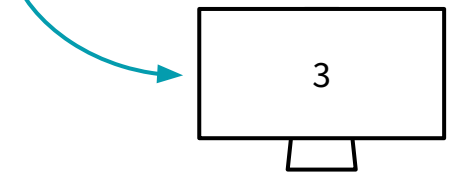
*Imagine you're in a toy store with many different types of toys: teddy bears, toy cars, and so on. Each of these toys can be seen as an **object**. Now, each toy, or object, has **attributes** and can perform certain **actions**. For example, a teddy bear might have attributes like its color or size, and an action where it plays a tune when you press its paw. Almost everything in Python can be thought of as an object, just like our toys. As you delve deeper into Python, you'll see that this idea of "everything is an object" is quite powerful.*

Conversion Functions

int(), **float()** and **str()** are conversion functions. They're used to explicitly convert one data type to another. Sometimes, you might have data in one format, but you need it in another. These functions allow you to make that switch.

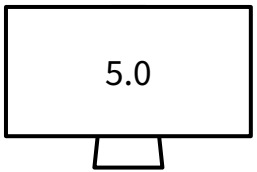
int() converts a value to an integer, which is a whole number.

```
1 my_var = 3.14
2 var_int = int(my_var)
3 print(var_int)
```



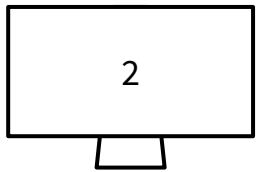
float() converts a value to a floating-point number, which can have decimals.

```
1 my_var = 5
2 var_flt = float(my_var)
3 print(var_flt)
```



str() converts a value to a string (text).

```
1 my_var = 2
2 var_str = str(my_var)
3 print(var_str)
```



It looks like an integer but it's a "2" and not a 2.

Input Function

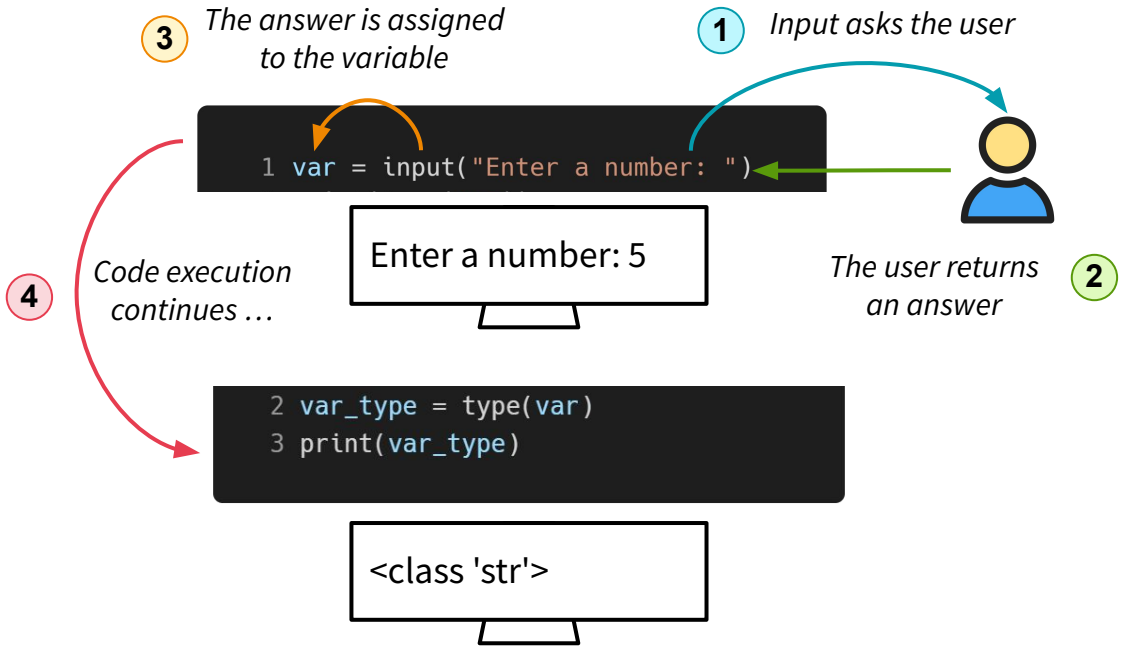
The **input()** function is a built-in function in Python that allows users to take input from the keyboard and returns it as a string. It can take an optional string argument, which is displayed to the user. This serves as a prompt to guide the user on what to input.

General Mistakes

Since input() always returns a string, beginners often forget to convert this string to the desired data type, leading to errors.

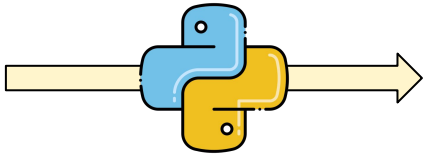
The input() function doesn't have built-in validation the check the type of the user's entry.

Execution of the program halts until the user provides input and hits the "Enter" key.



Numerical Operators

Addition	+
Subtraction	-
Multiplication	*
Exponentiation	**
Division	/
Floor Division	//
Modulus	%



```
1 my_var_1 = 5
2 my_var_2 = 10
3 print(my_var_1 + my_var_2)
```

15

```
1 my_var_1 = 5
2 my_var_2 = 10
3 print(my_var_1 * my_var_2)
```

50

The floor division is the entire part of a floating-point number. The modulus is the remainder of the division.

Variables from Operations - Numbers

The value of variable can be a calculation

```
1 my_var_1 = 5 ** 2
2 print(my_var_1)
```

25

A variable can be used in a calculation to set the value of another variable

```
1 my_var_1 = 5
2 my_var_2 = my_var_1 - 20
3 print(my_var_2)
```

-15

```
1 my_var_1 = 12
2 my_var_1 = my_var_1 % 5
3 print(my_var_1)
```

2

Variables from Operations - Strings

In Python, strings are among the most versatile data types, and they come with a plethora of operations that you can perform on them. Among all the numerical operators, only the addition and the multiplication works:

*In Python strings, the **backslash (\)** is known as the escape character. It's used to represent certain special characters within strings, allowing you to include characters that might be problematic or difficult to include otherwise. When the backslash is followed by certain characters, it represents a special character.*

***\n** - Represents a newline, which moves the cursor to the next line.*
***\t** - Represents a tab, which adds a tab space.*

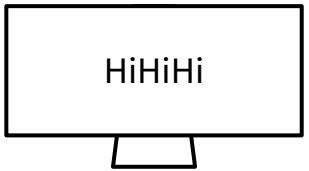
and so on

The addition combines two or more strings into one.

```
1 var_1 = "Hello"
2 var_2 = "World !"
3 var = var_1 + " " + var_2
4 print(var)
```

The multiplication repeats a string a specified number of times.

```
1 var = "Hi"
2 var = var * 3
3 print(var)
```



Multiple & Chained Assignments

In Python, you have the flexibility to assign values to multiple variables simultaneously or to assign the same value to several variables in a single line. This is known as **multiple assignments** and **chained assignments**.

It helps in writing more concise code, especially when initializing several variables to the same value or when extracting multiple values from a data structure (see chapter 05). In certain situations, using multiple or chained assignments can make the code easier to read and understand.

However, it's essential to use these features judiciously. Overusing them, especially with complex data structures, can sometimes make the code less readable. ***Always prioritize clarity over cleverness!***

Multiple assignments

```
1 x, y, z = 5, 10, 15
```

```
1 x = 5
2 y = 10
3 z = 15
```

Chained assignments

```
1 x = y = z = 5
```

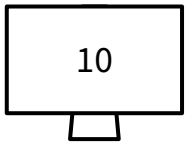
```
1 x = 5
2 y = 5
3 z = 5
```

Math Function for Numbers

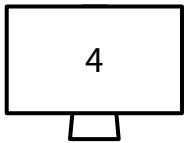
abs(number**)**	Return the **absolute value** of a number
round(number, n**)**	**Round** a number to n decimals
min(nb1, nb2, …**)**	**Find the minimum** of a number serie
max(nb1, nb2, …**)**	**Find the maximum** of a number serie

```
1 var = 3.1459
2 round_var = round(var, 2)
3 print(round_var)
```


```
1 var = -10
2 abs_var = abs(var)
3 print(abs_var)
```




```
1 var1, var2, var3 = 4, 5 ,8
2 min_var = min(var1, var2, var3)
3 print(min_var)
```



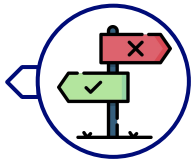
Exercises

Now, it's time to **practice**!

Complete the following **exercise sheets**:

- 1 - (EN) Exercises - Variables & Types

03



Conditional Logic

Introduction to Python

Conditional Logic

If you've ever pondered questions like "If it rains, should I bring an umbrella?" or "If the movie ticket is less than $10, I'll go see it," then you've already used **conditional logic** in your everyday life. In programming, this logic allows computer programs to make decisions, much like you do in the real world.

Conditional

The word "conditional" relates to conditions or scenarios that might occur.

Logic

"Logic" refers to the process of reasoning used to solve problems or make decisions.

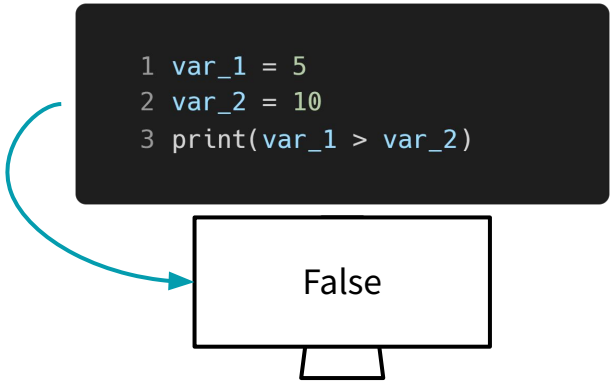
Therefore, **Conditional Logic** refers to the reasoning process programs use based on certain conditions.

Condition

When you hear the word **Condition** in the context of programming, think of it as a simple question your program asks. The answer to this question isn't a long sentence, but a straightforward "Yes" or "No" - or in Python terms, **True** or **False** : booleans.

In Python, conditions are usually made up of two objects and a comparator. This is the tool you use to compare the two values. Some of the common comparators in Python are:

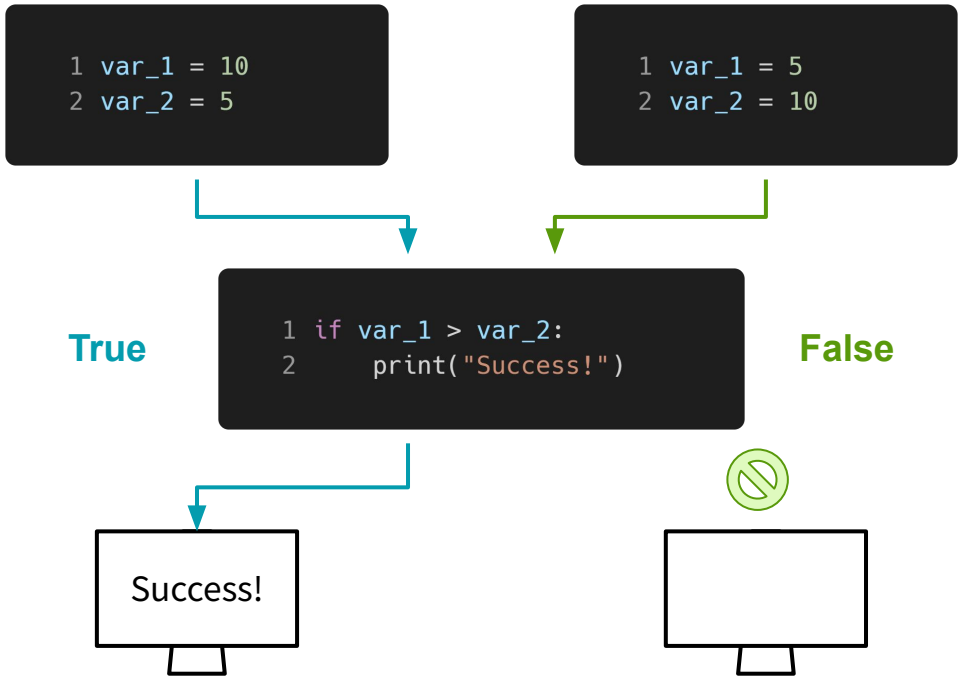
Are the two values **equal** ?	==
Are the two values **not equal** ?	!=
Is the 1st value **greater than** the 2nd ?	>
Is the 1st value **lower than** the 2nd ?	<
Is the 1st value **greater than or equal** to the 2nd ?	>=
Is the 1st value **lower than or equal** to the 2nd ?	<=



Logical Test – if Statement

The **if statement** tells the computer, "if a particular condition is true, then do something.". In essence, a condition provides a True or False answer. The if statement then uses this answer, allowing us to run particular actions when the condition is True.

Imagine you're standing in front of a locked door. To open this door, you need the right key. If your key fits the lock, the door will open. If not, the door remains locked. This is very similar to how conditions and if statements work together in Python.



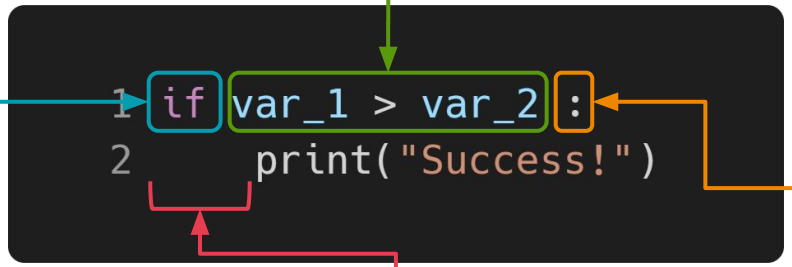
Logical Test – if Statement Syntax

if Keyword

*This is where it all begins. The if keyword in Python is used to test a specific condition. It should **ALWAYS** be in lowercase.*

Condition

This is a logical test that follows the if keyword. The condition is an expression that evaluates to either True or False.



Colon (:)

*The colon is a **critical aspect of Python syntax**. It indicates the end of the if statement's condition and signifies the beginning of the block of code that will be executed if the condition is True.*

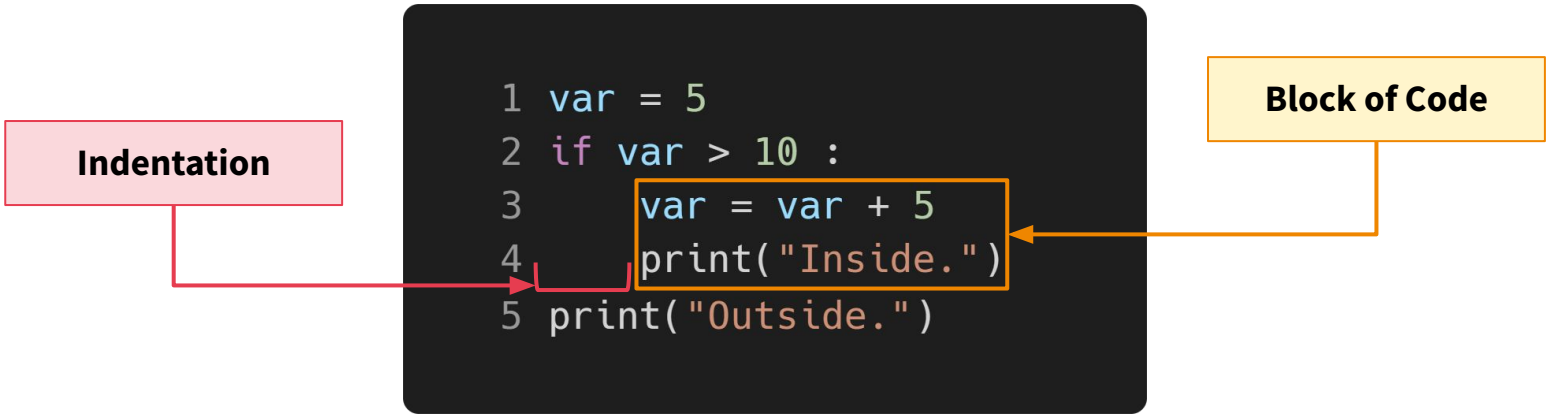
Indentation

*In Python, unlike many other programming languages that use braces { } or other markers to define **blocks of code**, indentation is used. The indented block of code under the if statement represents what the program should do if the condition is true. If the condition is false, the indented code is skipped.*

Indentation & Blocks of Code

Indentation refers to the spaces at the beginning of a line of code. In Python, indentation isn't just for aesthetics; it has a vital syntactical role. It delineates what we call **blocks of code**.

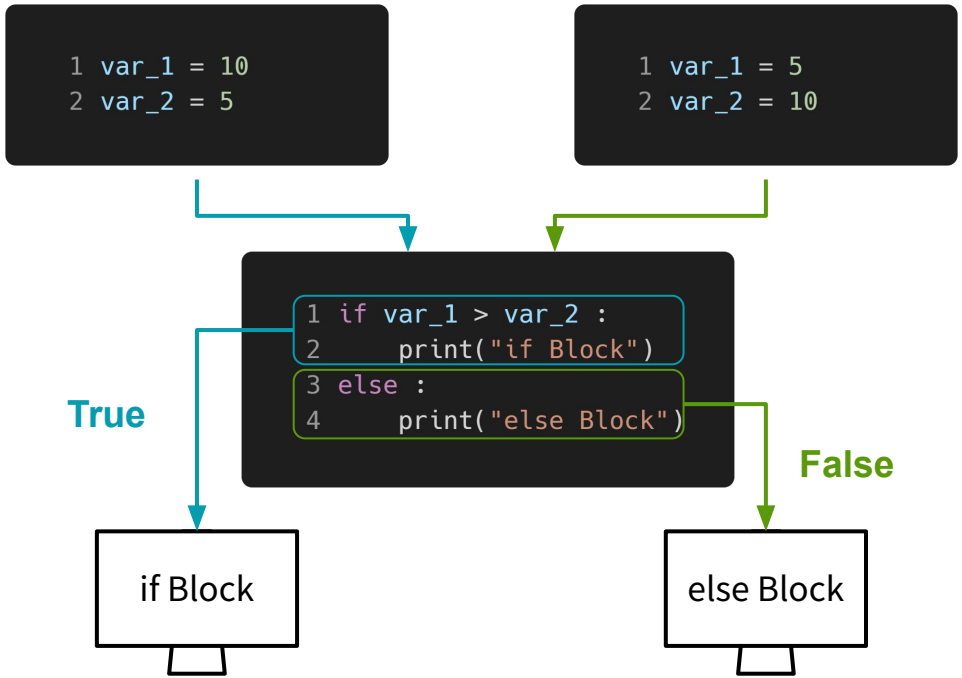
A **block of code** is a group of instructions that belong together as a unit. When a particular condition is met, such as in an if statement, everything in its associated block of code will execute.



Logical Test – else Statement

While the if statement defines what happens when a condition is met, the **else statement** captures the alternative action for when it's not. It's Python's way of handling both possibilities: the "if this is true" scenario and the "if it's not" scenario.

Imagine the scenario with the door once again. You try your key in the lock. If the key fits, the door opens. But what happens if the key doesn't fit? The door remains closed, right? Imagine now that you have a door that ONLY opens when you've tried the locked door. This is the alternative action: the else statement.



Logical Test – if/else Syntax

if Keyword

*This is where it all begins. The if keyword in Python is used to test a specific condition. It should **ALWAYS** be lowercase.*

Colon (:)

It indicates the end of both the if and the else statement conditions and signifies the beginning of the block of code that will be executed given the output of the condition.

```
1 if var_1 > var_2 :
2     print("if Block")
3 else :
4     print("else Block")
```

Don't forget to indent both the if and the else blocks.

else Keyword

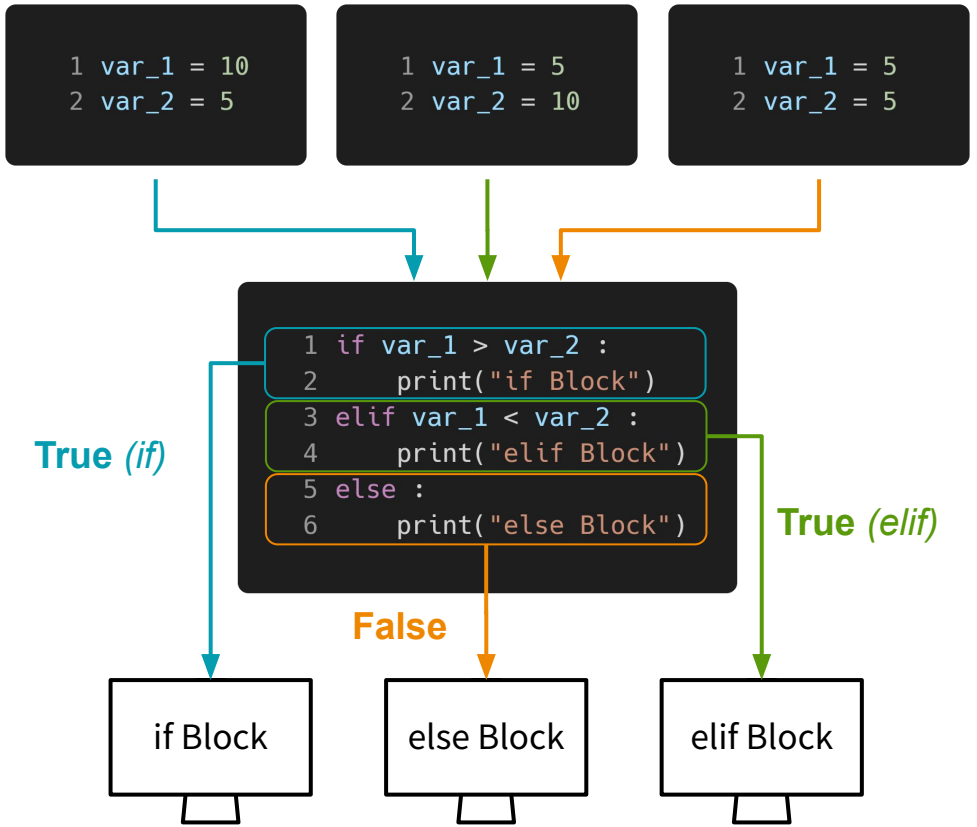
*After the block of code for the if statement, we have the else keyword. It introduces the alternative block of code that should run if the condition in the if statement is false. It should **ALWAYS** be lowercase.*

Logical Test – elif Statement

The **elif statement** is an abbreviation of "else if." It allows us to check multiple conditions sequentially, adding another layer of decision-making capability

Imagine you're standing in front of multiple doors. You have a key, and you want to find out which door it opens. You start with the first door and try your key. If it fits, the door opens. But if it doesn't, instead of concluding that none of the doors will open, you move to the next door and try your key there.

The elif statement represents this process of moving from one door to the next, checking each in turn until you find the one that your key opens.



Logical Test – Stacked elif

Multiple elif statements can be chained. There's no limit.

```
1 color = input("What's your favorite color ?")
2 if color == "blue" :
3     print("Blue is the color of the sky.")
4 elif color == "green":
5     print("Green is the color of the trees.")
6 elif color == "yellow":
7     print("Yellow is the color of the sun.")
8 elif color == "red":
9     print("Red is the color of passion.")
```

The else statement is not mandatory. However, a conditional logic should ALWAYS start with an if statement.

Logical & Identity Operators

Logical and **identity** operators allow you to form complex conditions by combining simple ones. This can be extremely useful when you need your program to make decisions based on multiple criteria or when you want to check the relationships between various data objects.

Logical Operators	Returns True if both conditions being tested are true.	and
	Returns True if at least one of the conditions being tested is true.	or
	Returns True if only one condition being tested is true.	^ *(xor)*
	Negates the result of the condition it precedes.	not
Identity Operator	Returns True if both variables point to the same data object.	is

Logical & Identity Operators

Let C1 and C2 be two distinctive conditions.

AND	C2 True	C2 False
C1 True	True	False
C1 False	False	False

XOR	C2 True	C2 False
C1 True	False	True
C1 False	True	False

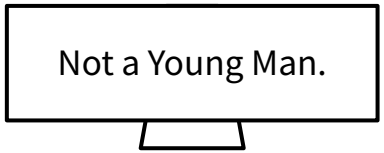
OR	C2 True	C2 False
C1 True	True	True
C1 False	True	False

NOT	
C1 True	False
C1 False	True

Multiple Conditions

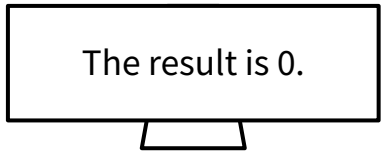
AND

```
1 age = 30
2 gender = "F"
3
4 if age < 18 and gender == "M":
5     print("Young Man.")
6 else:
7     print("Not a Young Man.")
```



OR

```
1 price = 0
2 number = 25
3
4 if price == 0 or number == 0:
5     print("The result is 0.")
6 else:
7     print(price*number)
```



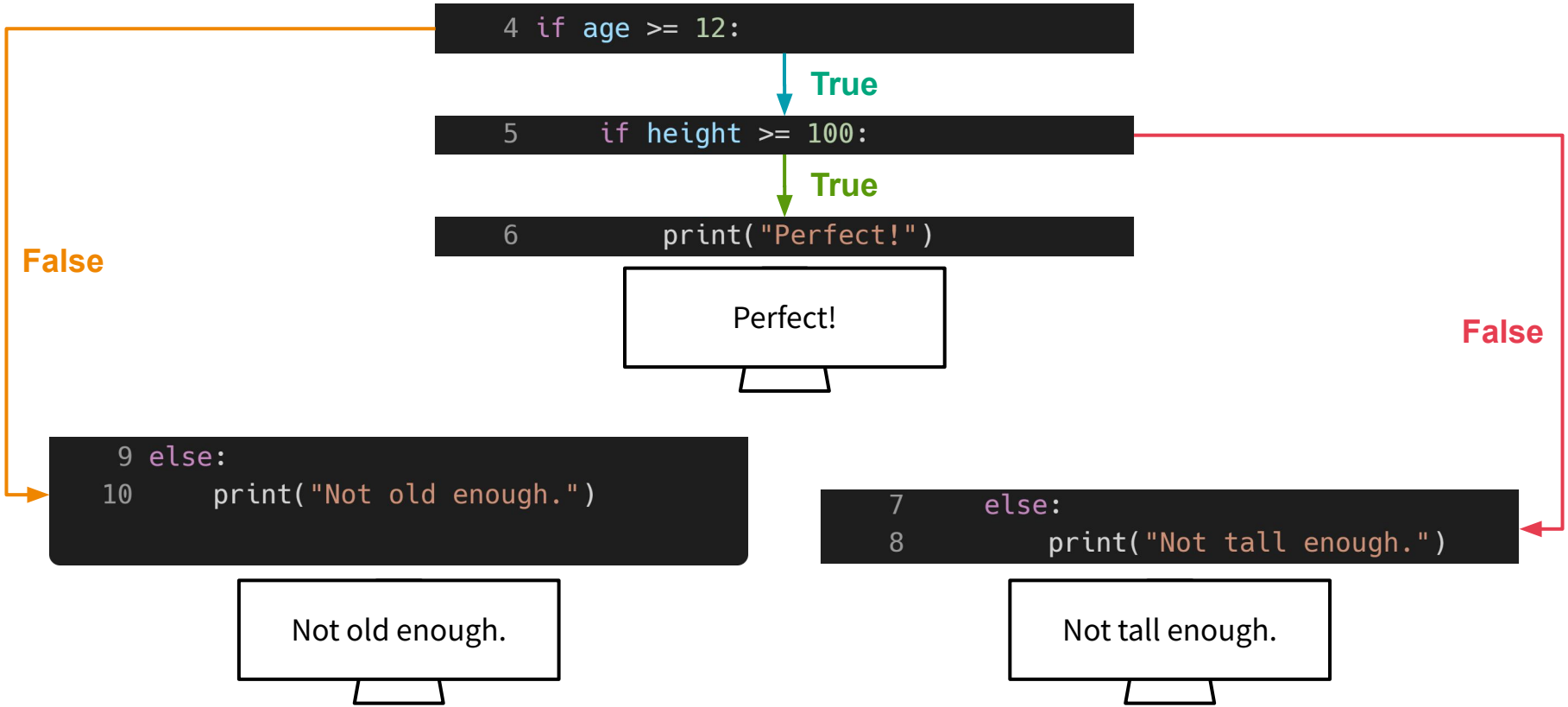
Nested Conditional Logic

Nested conditions in Python involve placing one or more conditional statements inside another conditional statement. When Python encounters a nested if statement, it evaluates conditions from the outside in.

Each if statement keeps its indentation. Hence two indentation spaces when one if is nested in another.

```
1  age = 14
2  height = 120
3
4  if age >= 12:
5      if height >= 100:
6          print("Perfect!")
7      else:
8          print("Not tall enough.")
9  else:
10     print("Not old enough.")
```

Nested Conditional Logic



Multiple Conditions or Nested Structures ?

Multiple Conditions

When conditions are independent - If the conditions you're checking don't depend on the outcome of another condition, use multiple conditions.

For clarity with and & or - If you can combine conditions with and or or and the result is still easy to understand, then go for it.

To reduce nesting - Reducing the level of nesting can improve code readability. If you can capture the logic with multiple conditions instead of nested ones, and it's clearer, choose that.

Nested Structure

When conditions are dependent - Use nested conditions if the evaluation of one condition depends on the result of another condition.

To emphasize sequence or hierarchy - If there's a logical or natural sequence to the conditions you're checking, nested structures can emphasize this sequence.

Regardless of the approach you choose, prioritize making your code as clear as possible.

As a rule of thumb, if you're nesting deeper than 3 levels, it might be a good idea to consider restructuring your conditions.

Exercises

Now, it's time to **practice**!

Complete the following **exercise sheets** in order:

- 2 - (EN) Exercises - Conditional Logic p.I
- 3 - (EN) Exercises - Conditional Logic p.II

04

Data Structures

Introduction to Python

Introduction

A **data structure** is a specific means of organizing and storing data in a computer such that it can be accessed and modified efficiently. Think of data structures as storage containers with specific shapes and functionalities that accommodate different kinds of data in various ways. The choice of a particular data structure often depends on the nature of the application and the operations required.

Efficiency

Proper data structures can optimize the performance of computer programs, making operations like searching, sorting, adding, and deleting items more efficient.

Reusability

Data structures are reusable, which means they can be implemented in one application and shared among multiple applications.

Memory Utilization

Effective use of data structures can lead to efficient memory usage, ensuring that data storage doesn't waste memory space.

Data Structure Types

While it's not the topic of this course, there are two main types of data structures:

- **Linear** - these are data structures where data elements are stored in a sequence, one after the other. You can think of them as being in a straight line, like people standing in a queue. We can find two linear data structure subtypes:
  - **Static Linear** - The size is fixed. Once you decide the size, you can't change it.
  - **Dynamic Linear** - The size can change. You can add or remove elements as needed.
- **Non-linear Data Structures** - These data structures don't follow a straight line. Instead, they branch out in multiple directions. You can think of them as being maps, networks or branches of a tree.

In general, Python provides a lot of flexibility and ease with its built-in data structures, which contributes to its popularity for various applications. Indeed, most built-in data structures in Python are dynamic (not all).

Python Built-In Data Structures

Lists

Ordered collection of items. Lists are mutable, meaning you can modify their content.

Think of them as a line of people holding hands. Each person is a piece of data, and we can easily add or remove people from the line.

Tuples

Similar to lists but immutable. Once data is set inside a tuple, it cannot be changed.

They are like lists, but once we make one, we can't change it.

Sets

Unordered collection of unique items. Useful for removing duplicates or testing membership.

It's like a bag where we put different fruits, but we can't have two of the same fruit.

Dictionaries

Key-Value paired collection. Think of it like an actual dictionary where for each word (the key), there's a definition (the value).

Imagine a real dictionary. For each word, there's a meaning. In Python, for each key, there's a value.

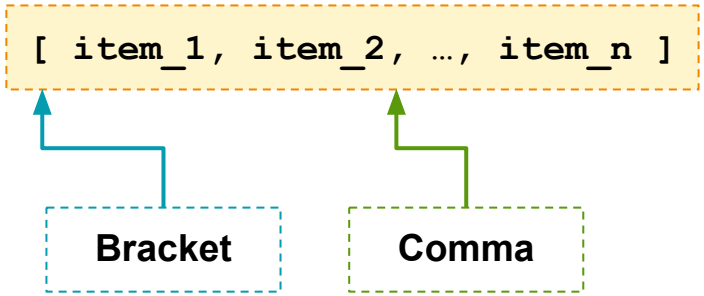
Lists

Lists are one of Python's most fundamental data structures. They can hold and mix a variety of data types, including integers, floats, strings, booleans, and more. Lists maintain the order of their elements and can contain repeated values. Finally, they are mutable, meaning their elements can be modified and each of them can be accessed using its index position.

An empty list is like a vacant container awaiting items.

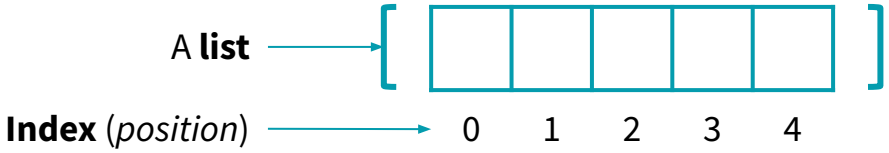
```
1 mylist = []
2 mylist = [0]
3 mylist = [1.1, 1.2, 1.3, 1.4]
4 mylist = [True, "abc", 3.14, 10]
```

Lists are created by placing all the items inside square **brackets [ ]**, separated by **commas**.



List Indexing

Each item can be accessed using its index position, starting from 0 for the first item.




```
1 mylist = ["a", "b", "c", "d", "e"]
```

```
1 print(mylist[3])
```

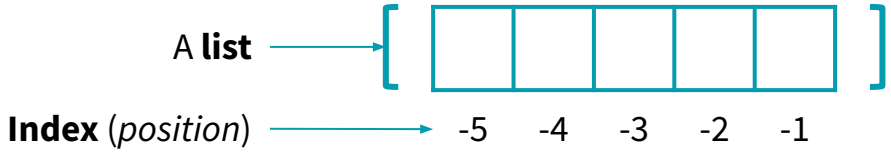
d

```
1 print(mylist[0])
```

a

List Reverse Indexing

Lists can be indexed from the end using negative numbers, with -1 being the last item.




```
1 mylist = ["a", "b", "c", "d", "e"]
```

```
1 print(mylist[-1])
```

e

```
1 print(mylist[-3])
```

c

List Slicing

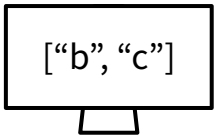
You can retrieve a portion of the list by using **slicing**.

In Python, when you slice a list using the format ***`[start:stop]`*** *:*

- ***start*** *- The index from which the slicing begins (**inclusive**).*
- ***stop*** *- The index where the slicing ends (**exclusive**).*

The concept of the exclusive stop is a general rule for slicing in Python. It's consistent across various data types that support slicing, not just lists.

```
1 mylist = ["a", "b", "c", "d", "e"]
2 print(mylist[1:3])
```



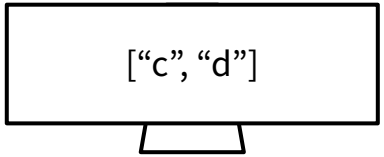
List Slicing

Slicing in Python is versatile and allows for various techniques to extract portions of a sequence.

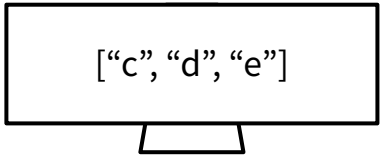
`mylist[b:e]`	Get all the list elements **from the b-th one to the e-th one excluded**
`mylist[-b:-e]`	Same as above in **reverse indexing** (-b should be lower than -e)
`mylist[b:e:s]`	Same as the first one but with a **step of s**
`mylist[b:e:-s]`	Same as above but with a **reverse step of s**
`mylist[b:]`	Get all the list elements **from the b-th one to the end**
`mylist[:e]`	Get all the list elements **from the start to the e-th one excluded**
`mylist[::-1]`	**Reverse all the list elements**

List Slicing

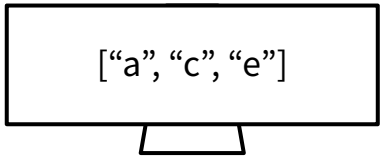
```
1 mylist = ["a", "b", "c", "d", "e"]
2 print(mylist[-3:-1])
```



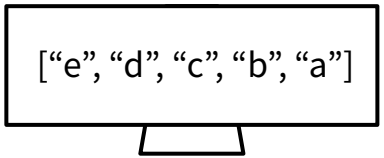

```
1 mylist = ["a", "b", "c", "d", "e"]
2 print(mylist[2:])
```




```
1 mylist = ["a", "b", "c", "d", "e"]
2 print(mylist[0:5:2])
```




```
1 mylist = ["a", "b", "c", "d", "e"]
2 print(mylist[::-1])
```



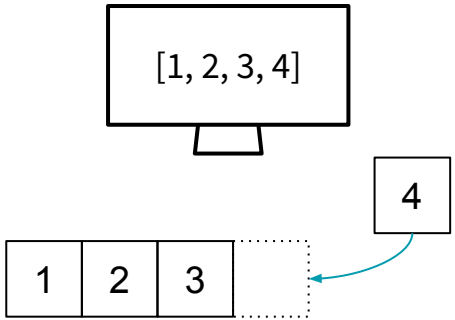
List Mutability

Lists are **mutable**, meaning you can modify an existing list by adding, removing, or changing items.

It's possible to add or change more than one item at once using another list.

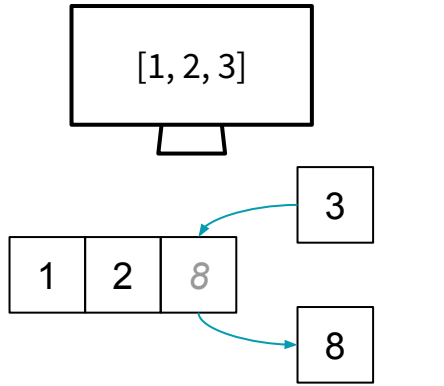
Adding an element

```
1 mylist = [1, 2, 3]
2 mylist = mylist + [4]
3 print(mylist)
```



Changing an element

```
1 mylist = [1, 2, 8]
2 mylist[2] = 3
3 print(mylist)
```



List Methods

Lists come with a variety of **built-in methods** for manipulation:

Adding methods	`.append(item)`	Adds an item to the end **in place**.
	`.insert(index, item)`	Inserts an item at a given index **in place**.
	`.extend(list)`	Adds the contents of another list to the end **in place**.
Removing methods	`.remove(item)`	Removes the item from the list **in place**.
	`.pop(index)`	Removes and returns the item at the given index.
	`.clear()`	Removes **all** the item from the list **in place**.
	`.sort()`	Sorts the list **in place**.
	`.reverse()`	Reverses the list **in place**.

List Adding Methods

In Place Methods

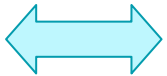
Some list methods modify the list directly and do not return a new list. This behavior is often referred to as "in-place" operations.

Methods like append(), insert(), extend(), remove(), and sort() all work this way.

For instance, when you use the insert() method to add an element to a list, it modifies the original list, and the method returns None.

```
1 mylist = [1, 2, 3]
2 mylist = mylist + [4]
3 print(mylist)
```

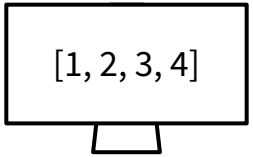
.append(item)



```
1 mylist = [1, 2, 3]
2 mylist.append(4)
3 print(mylist)
```

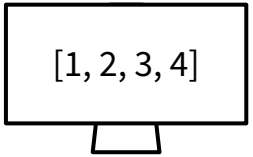
.insert(index, item)

```
1 mylist = [1, 2, 4]
2 mylist.insert(2, 3)
3 print(mylist)
```



.extend(container)

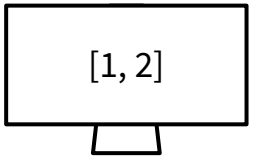
```
1 mylist = [1, 2]
2 mylist.extend([3, 4])
3 print(mylist)
```



List Removing Methods

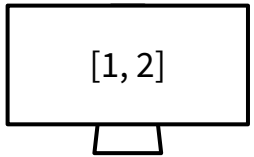
.remove(item)

```
1 mylist = [1, 2, 3]
2 mylist.remove(3)
3 print(mylist)
```



.pop(index)

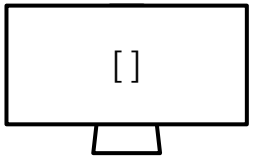
```
1 mylist = [1,2,3]
2 item = mylist.pop(2)
3 print(mylist)
```



The pop method is also in place but it returns the removed items.

.clear()

```
1 mylist = [1, 2, 3]
2 mylist.clear()
3 print(mylist)
```

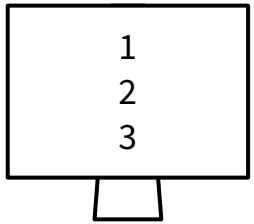


List Unpacking

List unpacking, also known as iterable unpacking, is a convenient feature in Python that allows you to extract multiple items from a list and assign them to variables in a single line of code. This can make your code cleaner and more readable.

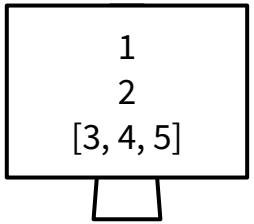
“Regular” Unpacking

```
1 numbers = [1, 2, 3]
2 a, b, c = numbers
3 print(a)
4 print(b)
5 print(c)
```



Advanced Unpacking

```
1 numbers = [1, 2, 3, 4, 5]
2 a, b, *c = numbers
3 print(a)
4 print(b)
5 print(c)
```



*Using the **Asterisk** * for Catch-All Unpacking.*

This allows you to capture multiple elements into a single variable.

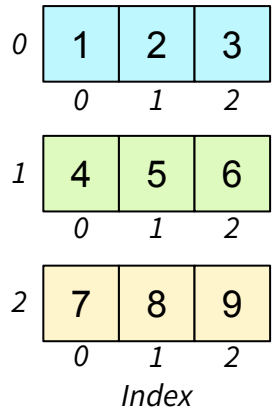
It's particularly useful when you're not sure about the number of elements in the list, or you're only interested in a part of the list.

Nested Lists

Nested lists in Python refer to lists that contain other lists as their elements. Essentially, it's a list inside another list. Nested lists are commonly used to represent **matrices**, create multi-dimensional arrays, or structure hierarchical data.

A **matrix**
made of a nested list

```
mylist = [[1,2,3],
          [4,5,6],
          [7,8,9]]
```




```
mylist = [[1,2,3],
          [4,5,6],
          [7,8,9]]

print(mylist[0])
print(mylist[0][1])
```

```
[1, 2, 3]
2
```

To access elements in a nested list, you'd use multiple indexing.

You can modify, add, or remove elements from nested lists just like with regular lists.

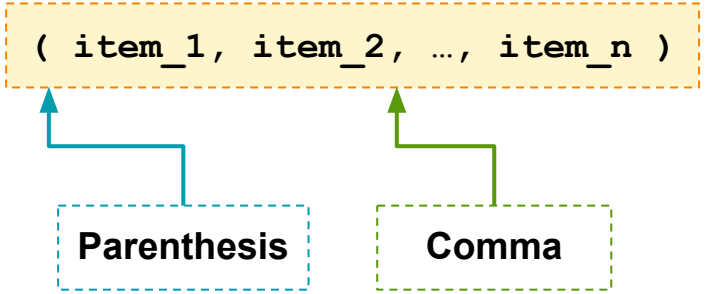
Tuples

Tuples are the simplest Python data structures, often seen as immutable siblings to lists. They can encapsulate various data types, such as integers, floats, strings, booleans, and more. Tuples preserve the sequence of their elements and can encompass duplicated values. However, a key distinction is their immutability, meaning once they're created, their content cannot be altered. Each item can still be accessed using its index position.

A tuple with a single value MUST include a comma: `(value,)`.

```
1 mytup = (0,)
2 mytup = (1.1, 1.2, 1.3, 1.4)
3 mytup = ('abc', 1, 3.14, True)
```

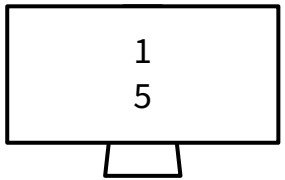
Tuples are created by placing all the items inside **parentheses ( )**, separated by **commas**.



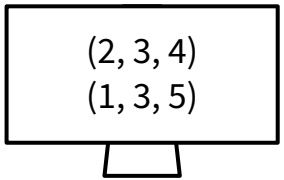
Tuple Indexing & Slicing

Tuples support both indexing and slicing operations, similar to lists. The syntax and behavior are almost identical between the two data structures in this regard.

```
1 mytup = (1, 2, 3, 4, 5)
2 print(mytup[0])
3 print(mytup[-1])
```




```
1 mytup = (1, 2, 3, 4, 5)
2 print(mytup[1:4])
3 print(mytup[::2])
```



Tuple Immutability

Immutability refers to the property of an object that prevents it from being modified after it's created. Once a tuple is created, this means, you cannot change, add, or remove its elements. There are immutable, but it's crucial to note that while tuples themselves are immutable, they can contain mutable objects. For instance, a tuple can have a list as one of its items, and the contents of that list can be changed.

```
1 mytup = (1, [2, 3], 4)
2 mytup[1][0] = 10
3 print(mytup)
```

(1, [10, 3], 4)

```
1 mytup = (1, 2, 3, 8)
2 mytup[3] = 4
3 print(mytup)
```

TypeError

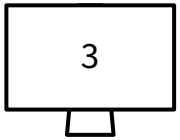
Tuple Methods

Tuples have fewer methods compared to lists, primarily due to their immutability. The two primary methods for tuples are:

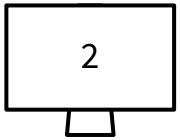
- ***.count(item)*** *- Returns the number of times a specified value appears in the tuple.*
- ***.index(item)*** *- Finds the first occurrence of a specified value in the tuple and returns its position.*

These two functions ***can also be used on lists and strings.***

```
1 mytup = (1, 2, 3, 2, 4, 2)
2 cnt = mytup.count(2)
3 print(cnt)
```




```
1 mytup = (1, 2, 3, 2, 4, 2)
2 idx = mytup.index(3)
3 print(idx)
```



Tuple vs List

Lists

Performance

May carry a bit of extra overhead due to their mutability and the variety of operations they support.

Use Case

Lists are great:

- when you need a collection that might need to change in size or in content.
- if you'll be performing frequent operations or random element modifications.
- for collections of homogeneous items.

Tuples

Performance

Generally, tuple operations can be slightly faster than list operations, especially in scenarios where the data doesn't need to be changed.

Use Case

Tuples are great :

- for representing things that shouldn't be altered, like days of the week or dates on a calendar.
- if you're defining a constant set of values, a tuple can be slightly more efficient.
- if you want to ensure data remains consistent throughout its lifecycle.

in Operator

The **in operator** in Python is used to check if a value exists in a given sequence. In the context of lists/tuples and conditional logic, the in operator can be incredibly useful to determine if a list/tuple contains a specific element.

```
1 mylist = [1, 2, 3, 4, 5]
2 if 3 in mylist:
3     print("3 is in the list!")
```

3 is in the list!

```
1 mytup = (1, 2, 3, 4, 5)
2 if 3 in mytup:
3     print("3 is in the tuple!")
```

3 is in the tuple!

Checking for the presence of an element in a list/tuple using the in operator is can be, for large ones, very slow.

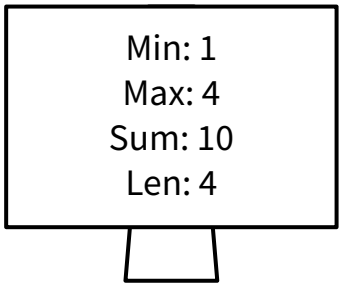
Math & Utility built-in functions

Here are other built-in functions that operate on iterables like **lists**, **tuples, sets** or **strings**. Yes, strings can be considered as iterables in Python as you will see in Chapter 05.

min(iterable)	Find the **min** of the iterable
max(iterable)	Find the **max** of the iterable
sum(iterable)	**Sum** the elements of the iterable
len(iterable)	Find the **number of elements** inside a iterable

list(iterable)	**Convert** a data structure **into a list**
tuple(iterable)	**Convert** a data structure **into a tuple**

```
1 mytup = (1, 2, 3, 4)
2 print("Min: ", min(mytup))
3 print("Max: ", max(mytup))
4 print("Sum: ", sum(mytup))
5 print("Len: ", len(mytup))
```



Remember, these functions can be used on Lists too !

Copy of a List or Tuple

Copying a sequence in Python can be achieved using various methods. Depending on the sequence type, you might choose one method over another.

The basic beginner mistake when copying a list in Python is directly assigning one list to another, which leads to both variables pointing to the same memory location.

This means that changes made to one list are reflected in the other, because both variables reference the same underlying list object, not two distinct lists.

```
1 mylist = [1, 2, 3, 4]
2 mylist_copy = mylist
```

Using Slicing

```
1 mylist = [1, 2, 3, 4]
2 mylist_copy = mylist[:]
3
4 mytup = (1, 2, 3, 4)
5 mytup_copy = mytup[:]
```

Using Built-in Constructors

```
1 mylist = [1, 2, 3, 4]
2 mylist_copy = list(mylist)
3
4 mytup = (1, 2, 3, 4)
5 mytup_copy = tuple(mytup)
```

Using the copy Method

Lists only

```
1 mylist = [1, 2, 3, 4]
2 mylist_copy = mylist.copy()
```

Exercises

Now, it's time to **practice**!

Complete the following **exercise sheet**:

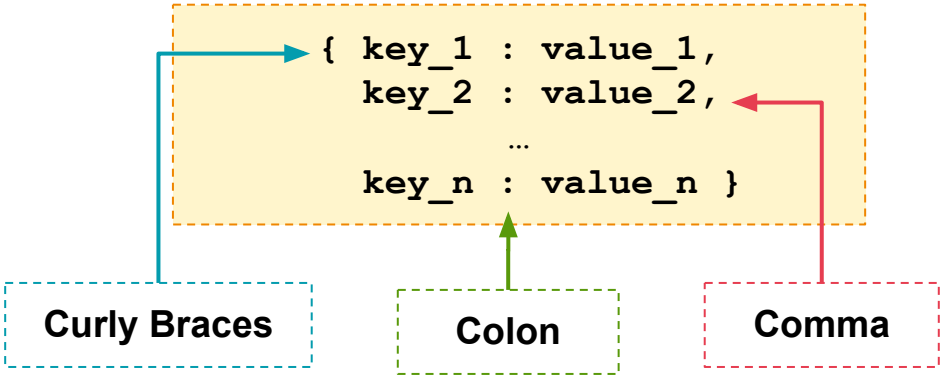
- 4 - (EN) Exercises - Data Structure p.I

Dictionaries

Dictionaries store data as key-value pairs, allowing for efficient data retrieval. Keys in a dictionary are unique, ensuring each key maps to a single value. This structure can accommodate a diverse set of data types, from integers and strings to tuples and other dictionaries. Unlike lists, which use index-based access, dictionaries use keys to access and modify their values. They are mutable, meaning their content can be altered, and values can be updated, added, or removed. Furthermore, as of Python 3.7, dictionaries maintain the insertion order of their items, making the order predictable during iteration.

```
1 dictio = {"apple" : 3,
2           "cherry": 24,
3           "banana": 6}
```

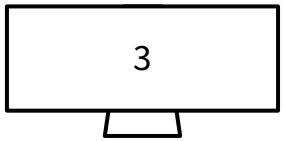
Dictionaries are created by placing all the key-value pairs separated by a **colon** inside **curly braces { }**. Key-value pairs are separated by **commas**.



Accessing a Dictionary Value

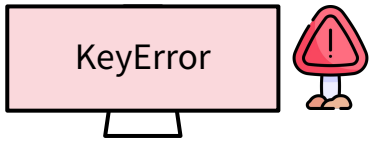
Dictionaries are key-value stores, which means you access values using their corresponding keys rather than by a numerical index.

```
1 dictio = {"apple" : 3,
2           "cherry": 24,
3           "banana": 6}
4 print(dictio["apple"])
```



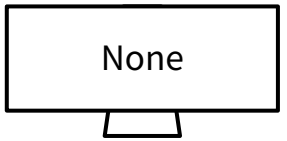
If you try to access a key that doesn't exist in the dictionary using the above method, Python will raise a KeyError.

```
1 dictio = {"apple" : 3,
2           "cherry": 24,
3           "banana": 6}
4 print(dictio["pear"])
```



To avoid this, you can use the get() method of dictionaries which will return None if the key doesn't exist.

```
1 dictio = {"apple" : 3,
2           "cherry": 24,
3           "banana": 6}
4 print(dictio.get("pear"))
```



.get(key)

Changing a Dictionary Value

Values of a dictionary can be changed by calling the dictionary with the corresponding key. It is also possible to update multiple values at once with the **.update()** method.

```
dictio = {"apple" : 3,
          "cherry": 24,
          "banana": 6}
dictio["apple"] = 10
print(dictio)
```

```
{'apple': 10,
 'cherry': 24,
 'banana': 6}
```

```
dictio = {"apple" : 3,
          "cherry": 24,
          "banana": 6}
dictio.update({"apple":10,
               "banana":3})
print(dictio)
```

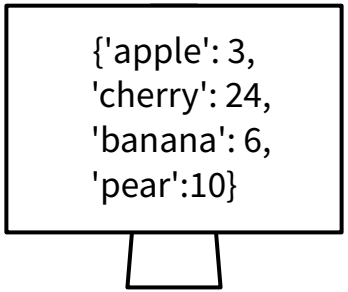
```
{'apple': 10,
 'cherry': 24,
 'banana': 3}
```

update() *is an inplace method that returns None. It is generally used along with another dictionary but a list of tuples (key, value) can also be provided.*

Adding a Key-Value Pair

You can add a new key-value pair by directly assigning a value to a key. However, this method will update the value associated with the key if the key already exists.

```
1 dictio = {"apple" : 3,
2           "cherry": 24,
3           "banana": 6}
4 dictio["pear"] = 10
5 print(dictio)
```



Removing a Key-Value Pair

You can use four different methods to remove a key-value pair from a dictionary.

Using the **pop()** Method

This method removes a key and returns its value.

```
1 dictio = {"apple" : 3,
2           "cherry": 24,
3           "banana": 6}
4 fruit = dictio.pop("apple")
```

it removes “apple”

Using the **popitem()** Method

This method removes and returns the last key-value pair as a tuple.

```
1 dictio = {"apple" : 3,
2           "cherry": 24,
3           "banana": 6}
4 fruit = dictio.popitem()
```

it removes “banana”

Using the **clear()** Method

This method empties the dictionary, removing all key-value pairs.

```
1 dictio = {"apple" : 3,
2           "cherry": 24,
3           "banana": 6}
4 dictio.clear()
```

dictio is empty : {}

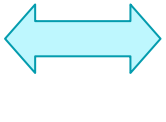
*The **del statement** in Python is a versatile tool used to delete objects, including variables, items from data structures, or slices of sequences.*
Thus, `del dictio["apple"]` would remove the corresponding Key-Value pair from the dictio dictionary.

Updating a Value using Augmented Assignment Operators

Augmented Assignment Operators (AAO) provide a way to perform an operation and an assignment in a single step, making code more concise and sometimes more efficient. Using augmented assignment is considered more "Pythonic" for simple operations. It's a common pattern in many Python programs.

Add and assign	+=
Subtract and assign	-=
Multiply and assign	*=
Exponentiation and assign	**=
Divide and assign	/=
Floor divide and assign	//=
Modulus and assign	%=

```
1 a = 5
2 a += 3
```



```
1 a = 5
2 a = a + 3
```

It is cleaner to use the AAO to update a value in a dictionary.

```
1 dictio = {"apple" : 3,
2           "cherry": 24,
3           "banana": 6}
4 dictio["apple"] += 4
```

*AAO can be used for **lists** as well but not for tuples because of their immutability.*

Get Values and Keys

Three methods, known as **.keys()**, **.values()**, and **.items()**, can be used to access various components of the dictionary.

.keys()

To get all the keys

```
1 dictio = {"apple" : 3,
2           "cherry": 24,
3           "banana": 6}
4 print(dictio.keys())
```

```
dict_keys(['apple',
           'cherry',
           'banana']
```

.values()

To get all the values

```
1 dictio = {"apple" : 3,
2           "cherry": 24,
3           "banana": 6}
4 print(dictio.values())
```

```
dict_values([3, 24, 6])
```

.items()

To get all the Key-Value pairs

```
1 dictio = {"apple" : 3,
2           "cherry": 24,
3           "banana": 6}
4 print(dictio.items())
```

```
dict_items([('apple', 3),
            ('cherry', 24),
            ('banana', 6)])
```

The resulting dict_keys, dict_values and dict_items are Python objects that you can convert to a list using the list() function – it also works with tuple().

Nested Dictionary

A **nested dictionary** refers to a dictionary where some of its values are dictionaries themselves. This is akin to having a tree-like structure or nested JSON data.

```python
person = {
    "name": "John",
    "age": 30,
    "address": {
        "street": "123 Main St",
        "city": "Springfield",
        "zipcode": "12345"
    }
}
```

In the example above, person is a dictionary that has another dictionary as the value for the "address" key. The indentation spaces shown here are not “real” ones, they’re just there for code readability.

```python
street = person["address"]["street"]
print(street)
```

123 Main St

```python
person["address"]["zipcode"] = "54321"
zipcode = person["address"]["zipcode"]
print(zipcode)
```

54321

To access or modify values in a nested dictionary, you chain the key assignments.

Key Restriction & Hash Tables

In Python, dictionary keys must be immutable, which means they cannot be changed after they are created. This is why you can use integers, strings, floats, and tuples as dictionary keys but not lists, sets, or other dictionaries. The core reason for this restriction is the way dictionaries are implemented in Python, which is through **hash tables**.

A **hash table** is a data structure that allows for efficient data retrieval. It uses a hash function to map keys to specific locations in an array. By knowing where data is stored, retrieval can be near-instantaneous.

Imagine a vast library. This library has millions of books, and you're in charge of storing them in a way that allows visitors to find any book quickly. If books were placed randomly, it would take ages for anyone to find what they're looking for. You need an efficient system.

Enter a magical librarian with an extraordinary ability: for any book handed to them, they immediately suggest a specific shelf and position on that shelf. This librarian is your hash function. You give her a book (key), and she provides a location (hash code).

```
1 dictio = {2 : 2, 1 : 1, 0 : 0}
2 print(dictio[0])
```

```
0
```

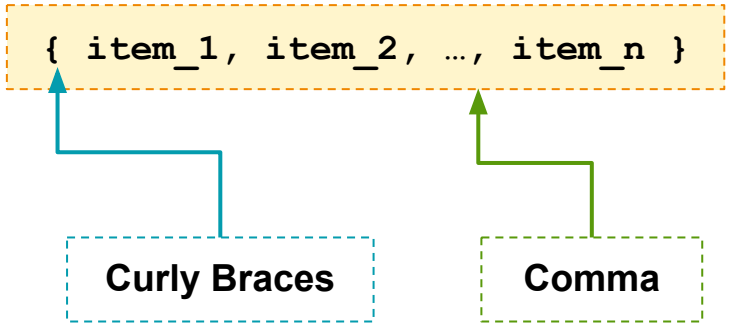
Do not mix-up keys with indexes. Indeed, `dictio[0]` refers to the key 0, NOT to the first index position in the dictionary as it is for lists.

Sets

Sets in Python are collections of unique elements, ensuring no duplication within the data. They are similar to lists and dictionaries in terms of being a collection type, but unlike lists and dictionaries, sets are unordered, meaning the items have no sequence. Sets can contain various data types, such as strings, integers, and floats, but the items must be hashable (immutable). An essential feature of sets is their ability to perform mathematical set operations like union, intersection, and difference. They are mutable, allowing you to add or remove elements, but since they're unordered, they don't support indexing or slicing like lists.

```
1 myset = {1}
2 myset = {1, 2, 3, 4}
3 myset = {1, True, 3.14, 'abc'}
```

Sets are created by placing all items inside **curly braces { }**, separated by **commas**.



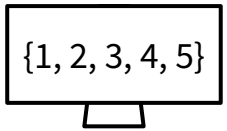
Adding Set Elements

To add an element to a set in Python, you would use the **add()** or the **update()** methods.

If the elements are already present in the set, calling add() or update() won't have any effect because sets do not allow duplicate values.

add()

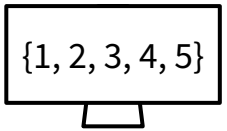
```
1 myset = {1, 2, 3, 4}
2 myset.add(5)
3 print(myset)
```



update()

It can be used to add multiple elements to the set at once.

```
1 myset = {1, 2, 3, 4}
2 myset.update({5})
3 print(myset)
```



Removing Set Elements

To remove an element from a set in Python, there are 5 methods available.

.remove()

This method removes the specified element from the set. If the element is not found, it raises a KeyError.

```
1 myset = {1, 2, 3, 4}
2 myset.remove(3)
3 print(myset)
```

{1, 2, 4}

.discard()

Removes the element from the set if it exists.

```
1 myset = {1, 2, 3, 4}
2 myset.discard(3)
3 print(myset)
```

{1, 2, 4}

.pop()

Removes and returns an arbitrary element from the set. There's no guarantee about which element will be removed.

```
1 myset = {1, 2, 3, 4}
2 myset.pop()
3 print(myset)
```

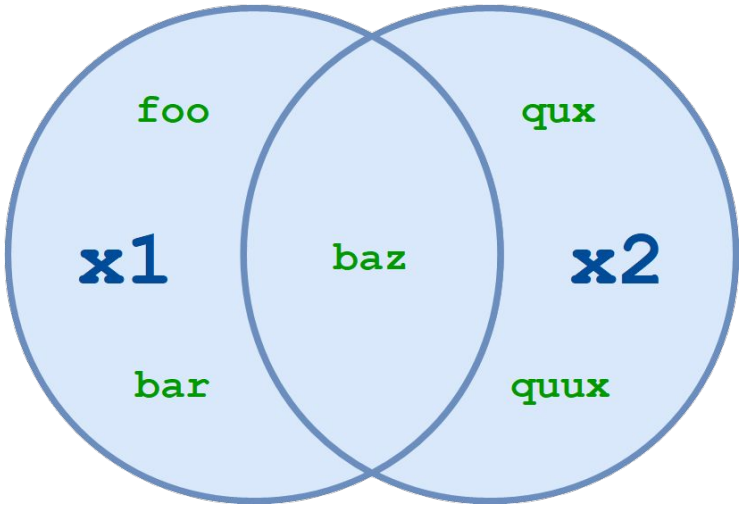
???

All of the aforementioned methods are in place, which means that they will return None.

*The **clear()** method and the **del statement** also work on sets.*

Mathematical Set Operations - Union

The **set union** combines the elements of two sets and removes duplicate elements.



union()

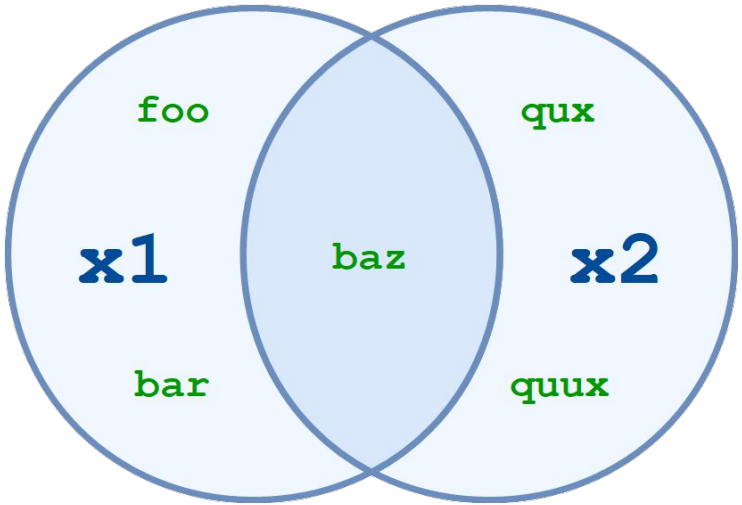
```
1 x1 = {"foo", "bar", "baz"}
2 x2 = {"qux", "quy", "baz"}
3 x3 = x1.union(x2)
4 print(x3)
```

```
{'qux', 'bar', 'foo', 'quy', 'baz'}
```

*The **.union()** method can be replaced by **|**.*
For example, `x3 = x1 | x2`

Mathematical Set Operations - Intersection

The **set intersection** returns the elements that are common to both sets.



intersection()

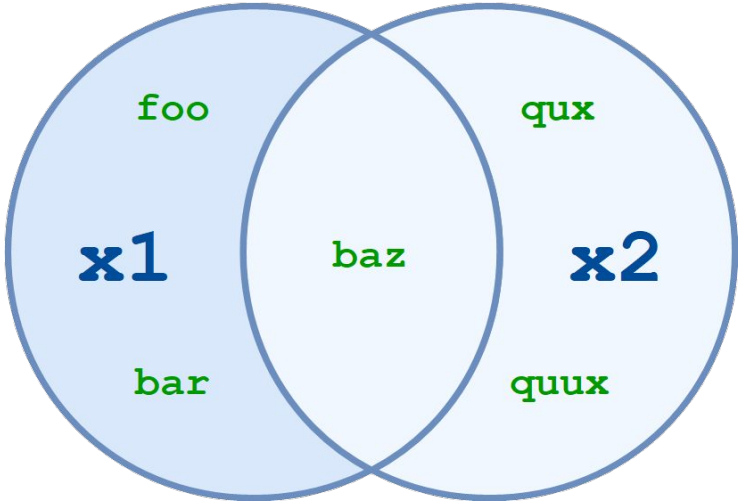
```
1 x1 = {"foo", "bar", "baz"}
2 x2 = {"qux", "quy", "baz"}
3 x3 = x1.intersection(x2)
4 print(x3)
```

```
{'baz'}
```

*The **.intersection()** method can be replaced by **&** .*
For example, `x3 = x1 & x2`

Mathematical Set Operations - Difference

The **set difference** returns the elements that are in the first set but not in the second set.



difference()

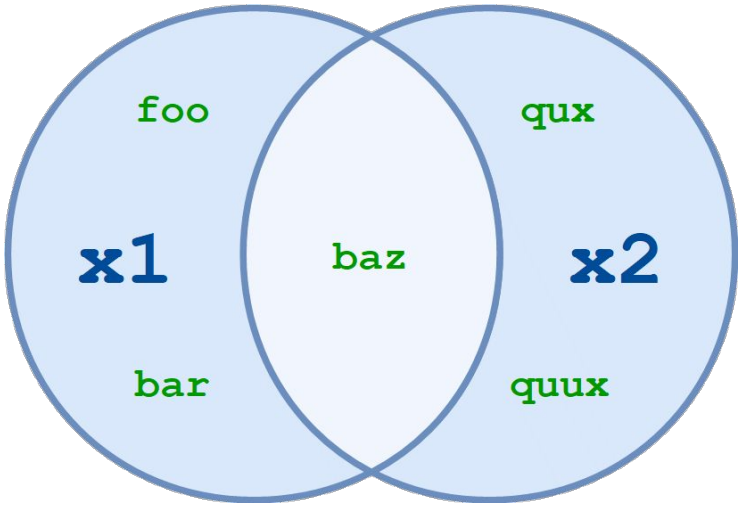
```
1 x1 = {"foo", "bar", "baz"}
2 x2 = {"qux", "quy", "baz"}
3 x3 = x1.difference(x2)
4 print(x3)
```

{‘foo’, ‘bar’}

*The **.difference()** method can be replaced by **-** .*
For example, `x3 = x1 - x2`

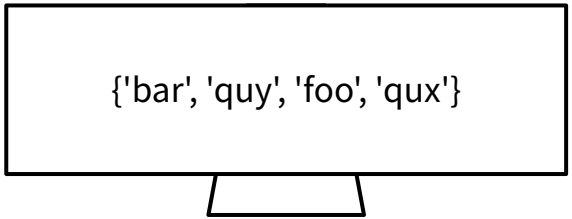
Mathematical Set Operations - XOR

The **set symmetric difference** returns the elements that are NOT common to two sets.



symmetric_difference()

```
1 x1 = {"foo", "bar", "baz"}
2 x2 = {"qux", "quy", "baz"}
3 x3 = x1.symmetric_difference(x2)
4 print(x3)
```



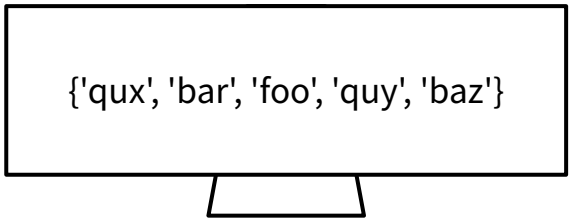
*The **.symmetric_difference()** method can be replaced by **^**.*
For example, `x3 = x1 ^ x2`

Mathematical Set Operations

Some augmented assignment operators allow for modifying the set in place. This can be more efficient, especially when working with large sets, as it avoids the overhead of creating a new set object.

Updates the set with the **union** of itself and another set.	|=
Updates the set with the **intersection** of itself and another set.	&=
Updates the set by **removing all common elements** relative to another set.	-=
Updates the set with the **symmetric difference** of itself and another set.	^=

```
1 x1 = {"foo", "bar", "baz"}
2 x2 = {"qux", "quy", "baz"}
3 x1 |= x2
4 print(x1)
```



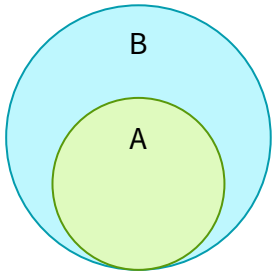
Subsets

Python provides 3 methods, **issubset()**, **issuperset()** and **isdisjoint()** to ascertain relationships between sets, helping in scenarios ranging from basic membership tests to advanced set theory-based logic operations.

issubset()

```
1 A = {1, 2}
2 B = {1, 2, 3, 4}
3 print(A.issubset(B))
```

True



issuperset()

```
1 A = {1, 2}
2 B = {1, 2, 3, 4}
3 print(B.issuperset(A))
```

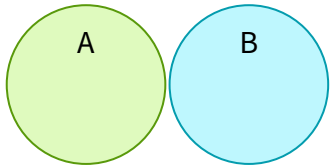
True

The concept of a proper subset or superset is not directly available in Python. If the two sets are identical, it will still return True.

isdisjoint()

```
1 A = {1, 2}
2 B = {3, 4}
3 print(A.isdisjoint(B))
```

True



Frozen Sets

A **frozenset** is a built-in set in Python that is immutable, meaning its elements cannot be modified after it is created. It is, essentially, a set that is "frozen." Because of its immutability, a frozenset is hashable, and thus can be used as a key in a dictionary, while a regular set cannot.

You can use frozensets when you want to ensure that a particular set of elements remains constant throughout the execution of the program.

Most of the set operations like union, intersection, difference, etc., can be applied on frozensets, but remember, these operations will always return a new frozenset without altering the original.

Since frozensets are immutable, you cannot add an element using methods like add() or remove an element using methods like remove() or discard().

frozenset()

It accepts any iterable i.e. lists, tuples, strings, sets …

```
1 frozen = frozenset([1, 2, 3, 4, 5])
2 print(frozen)
```

```
frozenset({1, 2, 3, 4, 5})
```

Conditional Logic with Dictionary & Sets

Dictionary

Key or Value Membership

in keyword

```python
dictio = {"a": 1, "b": 2, "c": 3}

if "a" in dictio.keys():
    print("'a' is a key in the dictionary")

if 2 in dictio.values():
    print("2 is a value in the dictionary")
```

*It's possible to check the size of a dictionary or a set using the **len()** function. Additionally, you can check if two dictionaries are equal using the **==** operator.*

Sets

Membership testing

in keyword

```python
my_set = {1, 2, 3, 4}
if 3 in my_set:
    print("3 is in the set")
```

Set Relations

.issubset(), .issuperset(), and .isdisjoint()

```python
A = {1, 2, 3}
B = {1, 2, 3, 4, 5}
if A.issubset(B):
    print("A is a subset of B")
```

Copy & Set/Dict Built-in Functions

As for lists and tuples, a copy of a set or a dictionary can be made using the **copy()** method or by using the constructor.

Sets

```
1 myset = {1, 2, 3}
2 copied_set = myset.copy()
```

```
1 myset = {1, 2, 3}
2 copied_set = set(myset)
```

Dictionaries

```
1 dictio = {'a': 1, 'b': 2}
2 copied_dict = dictio.copy()
```

```
1 dictio = {'a': 1, 'b': 2}
2 copied_dict = dict(dictio)
```

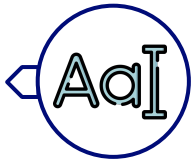
Exercises

Now, it's time to **practice**!

Complete the following **exercise sheet**:

- 5 - (EN) Exercises - Data Structure p.II

05



String Formatting

Introduction to Python

Strings as Data Structure

Strings, though primarily used for representing text, can also be thought of as a data structure in Python. Once a string is created, you cannot modify its content, they are immutable. Any operation that seems to change the string actually creates a new string. Strings are ordered and maintain the order of characters as they appear. Thus, each character in a string has a specific index, starting from 0 for the first character.

Indexing

```
1 word = "Python"
2 print(word[0])
```

P

Slicing

```
1 word = "Python"
2 print(word[1:4])
```

yth

Membership

```
1 print("Py" in "Python")
```

True

Strings are more space-efficient than lists or tuples of single characters because of their internal storage mechanism.

String Built-In Functions – Case Manipulation

Strings can be transformed based on case, which can be useful for formatting or case-insensitive comparisons.

.upper()

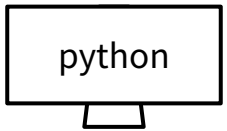
Converts all characters in a string to uppercase.

```
1 word = "Python"
2 print(word.upper())
```

.lower()

Converts all characters to lowercase.

```
1 word = "PYTHON"
2 print(word.lower())
```



.title()

Capitalizes the first character of each word in a string.

```
1 word = "hello world!"
2 print(word.title())
```

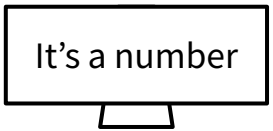
*There are several other case manipulation functions provided by Python for strings such as **.capitalize()** that capitalizes the first character of a string.*

String Built-In Functions – Checking Properties

.isdigit()

Returns True if all characters in the string are digits, otherwise False.

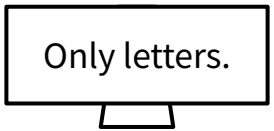
```
1 word = "123"
2 if word.isdigit() :
3     print("It's a number")
```



.isalpha()

Returns True if all characters are alphabetic, otherwise False.

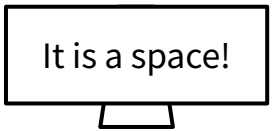
```
1 word = "abc"
2 if word.isalpha() :
3     print("Only letters.")
```



.isspace()

Returns True if there are only whitespace characters in the string.

```
1 word = " "
2 if word.isspace():
3     print("It is a space!")
```



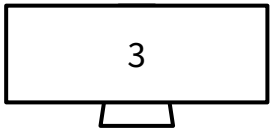
*There are several other case manipulation functions provided by Python for strings such as **islower()**, **isupper()** and **istitle()** which return True if the string is respectively all lowercase, all uppercase or has its first character of each word in uppercase.*

String Built-In Functions – Search & Replace

.find(substring)

Returns the index of the first occurrence of the substring. If not found, returns -1.

```
1 word = "Python"
2 print(word.find("h"))
```



.replace(old, new)

Replaces all occurrences of the old substring with new.

```
1 word = "Hello World!"
2 print(word.replace("Hello", "Python"))
```

*There are several other Search & Replace functions in Python such as **.index(substring)** that is similar to find(), but raises an exception if the substring is not found. Additionally, **.startswith(substring)** and **.endswith(substring)** check if a string starts or ends with a given substring, respectively.*

String Built-In Functions – Splitting & Joining

.split(delimiter)

Splits a string based on a delimiter (default is a whitespace) into a list of substrings.

```
1 word = "Hello World !"
2 print(word.split(" "))
```

.join(iterable)

Combines an iterable into a single string using the string as a delimiter.

```
1 delimiter = " "
2 words = ["Hello", "World", "!"]
3 print(delimiter.join(words))
```

The delimiter can be anything : a comma, a dash, a dot or even a string. The substrings can be stored inside of any iterable such as a list or a tuple.

String formatting

String formatting is a way to insert and combine data into strings. Imagine you're writing a small script where a user inputs their name, and you want to greet them with a message like "Hello, [name]!". Instead of adding strings together using the + sign, string formatting provides a cleaner and more versatile way to include variable content within strings.

The old-style % operator

This older method uses placeholders in the string, followed by the % operator and the variable (or variables) to insert.

The str.format() method

This method introduces curly braces {} as placeholders where you'd like to insert values.

The modern f-Strings

Introduced in Python 3.6, this is the most modern way to format strings and is generally considered the most readable and concise.

While the % operator and str.format() are still used in older codes and can be helpful to know, f-strings are more concise and beginner-friendly for new projects.

The Old-Style – % method

```
"... [format modifier] ..." % variable
```

Format Modifier

Strings	%s
Integers	%d
Floats	%f
Floats *with a restricted number of decimals*	%.<number>f

%2f would turn 3.14159265359 into 3.14

```
1 name = "Alice"
2 var = "Her name is %s." % name
```

```
1 age = 30
2 var = "She is %d years old." % age
```

```
1 price = 3.54
2 var = "She paid $ %f." % price
```

The str.format method

The **str.format()** method is a versatile way to format strings. It provides more readability and functionality compared to old-style string formatting. Instead of using % specifiers like in old-style formatting, you use **{ }** (curly braces) as placeholders where you want to insert values.

```
"... {} ...".format(variable)
```

```
1 name = "Alice"
2 var = "Her name is {}.".format(name)
```

```
1 name = "Alice"
2 age = 30
3 var = "Her name is {}, she is {} years old.".format(name, age)
```

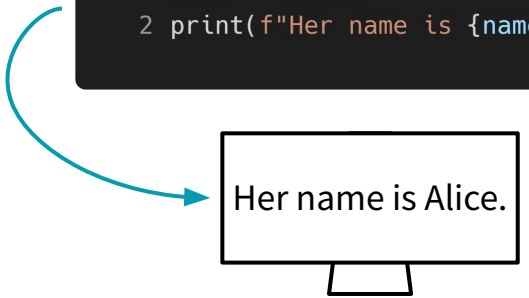
It's possible to specify multiple variables.
By default, values are inserted in the order they're provided.

The New-Style – f-Strings

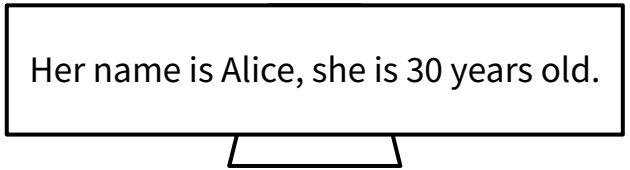
One of the primary benefits of f-strings is the directness and readability they offer. Expressions inside the string are right where you'd expect the output to appear, making it easier to see the formatted output structure at a glance. Additionally, f-strings can be faster than both old-style formatting and the str.format() methods.

```
f"... {variable} ..."
```

```
1 name = "Alice"
2 print(f"Her name is {name}.")
```




```
1 name = "Alice"
2 age = 30
3 print(f"Her name is {name}, she is {age} years old.")
```



Formatting Options

Formatting options are similar to the tools you'd use in a word processor to make your text look a certain way. It allows to pad, truncate or transform strings.

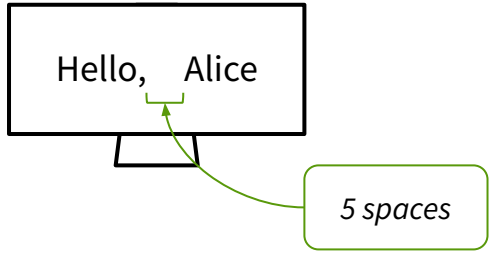
```
f"... {variable:formatting} ..."
```

Align text to the left	{ *var* : **> x** }
Align text to the right	{ *var* : **< x** }
Center text	{ *var* : **^ x** }
Truncate text	{ *var* : **.x** }

In { var : < x }, x is the total width of the final string.

For example, "Alice" has a length of 5. If you want to add 5 spaces at the right of Alice, you would use { "Alice" : < 10 }

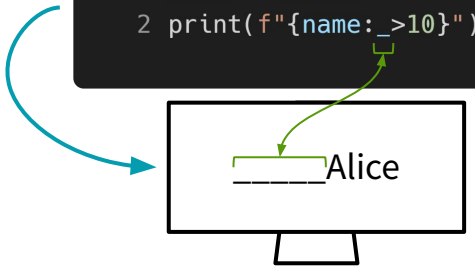
```
1 name = "Alice"
2 print(f"Hello,{name:>10}")
```



Formatting Options

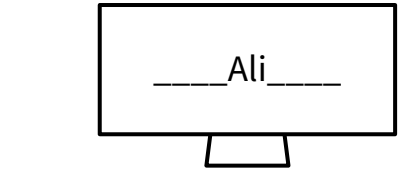
It's possible to specify a padding character.

```
1 name = "Alice"
2 print(f"{name:_>10}")
```

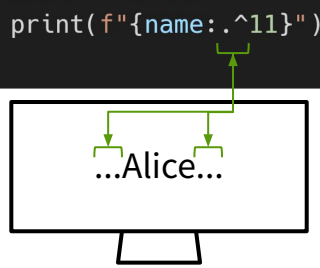


It's also possible to combine multiple formatting options. Here, a truncation and a padding.

```
1 name = "Alice"
2 print(f"{name:_^11.3}")
```




```
1 name = "Alice"
2 print(f"{name:.^11}")
```



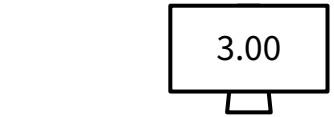
Formatting Options – Numbers

Decimal Places Rounding	{ *var* : **.xf** }
Scientifique Notation	{ *var* : **e** } { *var* : **.xe** }
Thousands Separator	{ *var* : **,** } { *var* : **.** }
Positive Sign	{ *var* : **+** }
Percentage	{ *var* : **%** } { *var* : **.x%** }
Padding *(same as text)*	{ *var* : **< x** } { *var* : **> x** } { *var* : **^ x** }

As for text, the dot tells you that it is a truncation and the x is the number of decimals you want to keep.

```
1 pi = 3.14159
2 print(f"{pi:.2f}")
```


```
1 nb = 3
2 print(f"{nb:.2f}")
```



Formatting Options – Numbers

By default, the positive sign is not shown.

```
1 positive = 42
2 print(f"{positive:+}")
```

+42

```
1 positive = 42
2 print("+" + str(positive))
```

+42

```
1 number = 42
2 print(f"{number:0>4}")
```

0042

```
1 big_number = 123456
2 print(f"{big_number:.2e}")
```

1.23e+05

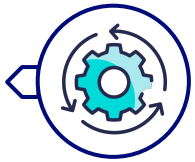
Exercises

Now, it's time to **practice**!

Complete the following **exercise sheet**:

- 6 - (EN) Exercises - String Formatting p.II

06



Iterables and Loops

Introduction to Python

Introduction

Loops are fundamental constructs in most programming languages, allowing for repeated execution of a block of code as long as a certain condition is met or for each item in a collection. It purpose is to execute repetitive tasks without writing the same code multiple times and to iterate over elements of data structures.

Imagine a factory assembly line where workers are tasked with doing a specific repetitive job on each item that comes their way. Each worker represents a loop, and each item on the assembly line is an iteration of that loop.

Definite Loop

Number of iterations is known beforehand. E.g., looping over a fixed list of items. The two most commons are:

- The **for** loop
- The **for-each** loop

Indefinite Loop

Iterations depend on certain conditions rather than a predetermined count. The two most commons are:

- The **while** loop
- The **do while** loop

A lots of loop variants exist but Python only has the for loop and the while loop. However, while Python lacks certain loop constructs found in other languages (like do-while), its loops can achieve the same results with slightly different structures and thus are very versatile.

For Loops

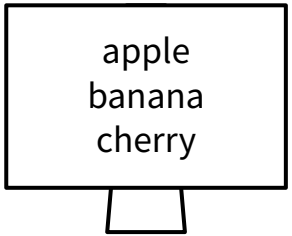
In Python, the **for loop** is used to iterate over a sequence – which can be a list, tuple or string – or other iterable objects – which can be a dictionary, set, file or generator . During each iteration, an item from the sequence is retrieved and executed with the loop's body.

Remember the factory analogy ?

This is like an assembly line that has a predetermined number of items coming down the conveyor belt. Each worker (loop iteration) knows beforehand how many items (data elements) they need to work on.

For example, if there are 100 items on the assembly line, the worker will perform their task 100 times, once for each item.

```
1 fruits = ["apple", "banana", "cherry"]
2 for fruit in fruits:
3     print(fruit)
```



For Loops

for Keyword

*This signals the start of the loop construct. It should **ALWAYS** be in lowercase.*

Temporary Variable

This is a temporary variable that takes the value of each element in the sequence, one at a time.

Colon (:)

*The colon is a **critical aspect of Python syntax**. It indicates the end of the for statement and signifies the beginning of the block of code that will be executed repeatedly.*

```
1 for fruit in fruits :
2     ...
```

Indentation

The indented block of code under the for statement represents what the program should do repeatedly until the sequence/iterable is empty.

in keyword

This keyword is used to specify that the loop will iterate over the given sequence.

Sequence/Iterable

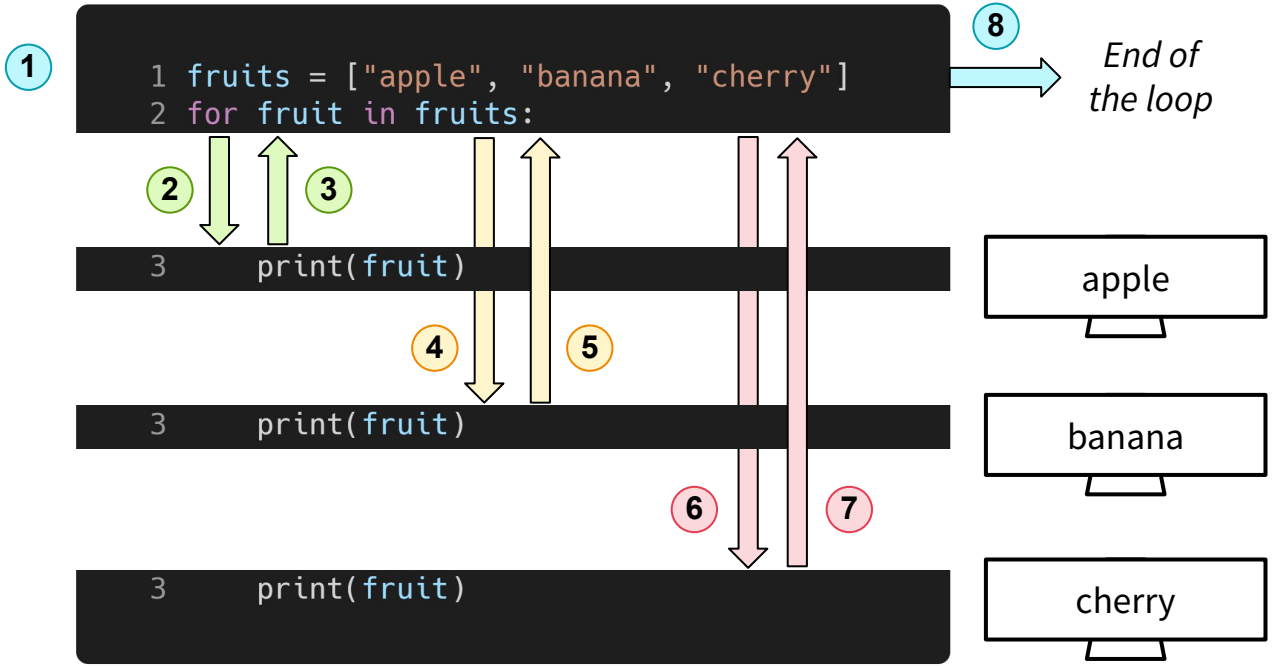
The sequence or iterable over which the loop will iterate. Common sequences include lists, strings, tuples, and more.

For Loops

*The for loop is **iterating** three times because the fruits list contains three items.*

The temporary variable fruit will take the successive values given by item order of the list: first apple, then banana and finally cherry.

When the iterable doesn't contain value to provide, the for loop stops.



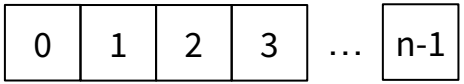
The Range Object

The **range** object represents an immutable sequence of integers and is commonly used for looping a specific number of times in for loops. It can be created using the **range()** function.

The range object can be converted into a list, a tuple or a set using the appropriate built-in methods.

If you need to check if an integer is within a range without iterating, range is faster than a list or tuple because it uses a constant amount of memory and performs the check in constant time.

range(n)




```
1 int_list = list(range(5))
2 print(int_list)
```

[0,1,2,3,4]

The last integer (n) is not included

```
1 nb = 5
2 if nb in range(6):
3     print("Is in range.")
4 else :
5     print("Not in range.")
```

Is in range.

The Range Object

Range has up to three arguments: **start**, **stop** and **step**.

stop

*The endpoint of the sequence. This value is **not included** in the sequence.*

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

```
0
1
2
3
4
```

start

(Optional) The value of the start point of the sequence. Default is 0.

```
1 for i in range(1, 5):
2     print(i)
```

```
1
2
3
4
```

step

(Optional) The difference between each number in the sequence. Default is 1.

```
1 for i in range(1, 5, 2):
2     print(i)
```

```
1
3
```

The Range Object

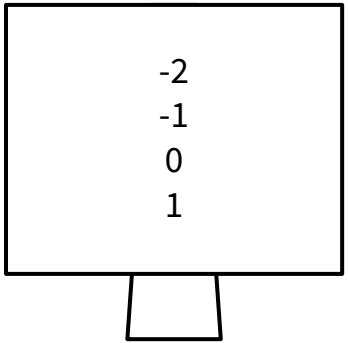
The stop value is always exclusive, whether you're moving in a positive or negative direction.

When using a negative step, ensure that the start value is greater than the stop value, otherwise, the range won't produce any values. The opposite is true for a positive step.

Negative Range

You can start with a negative number.

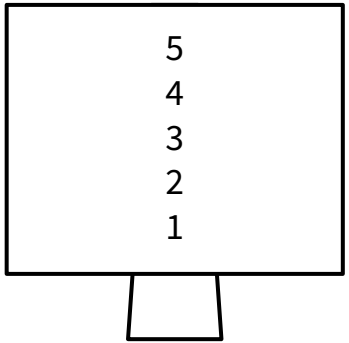
```
1 for i in range(-2, 2):
2     print(i)
```



Reverse Range

You can produce a sequence in reverse by making the start larger than the stop value

```
1 for i in range(5, 0, -1):
2     print(i)
```



For Loops - Sequence Indexing

You can combine **range()** with **len()** to iterate through a sequence in a for loop.

```python
mylist = ["a", "b", "c", "d"]
for i in range(len(mylist)):
    print(f"Element at index {i} is {mylist[i]}")
```

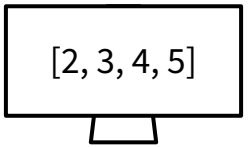
```
Element at index 0 is a
Element at index 1 is b
Element at index 2 is c
Element at index 3 is d
```

It also works with tuples and strings.

For Loops - Modifying a Sequence

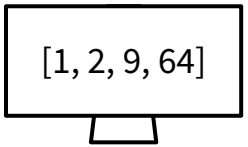
You can't modify the values of a list by looping directly over it. Thus, if you need to modify the values of a list while iterating through it, you should loop using range() and then access the values by their index to modify them.

```
1 mylist = [1,2,3,4]
2 for i in range(len(mylist)):
3     mylist[i] += 1
4 print(mylist)
```



It's possible to use the index in calculations.

```
1 mylist = [1,2,3,4]
2 for i in range(len(mylist)):
3     mylist[i] **= i
4 print(mylist)
```



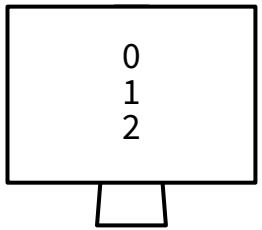
Break, Continue & Pass

break, **continue**, and **pass** are control statements in Python that can be used within for loops to influence the flow of execution.

break

*The break statement is used to terminate the loop **prematurely** when it is encountered inside a loop.*

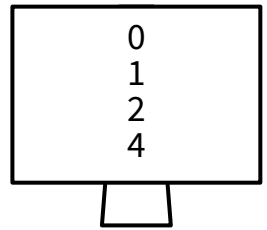
```
1 for i in range(5):
2     if i == 3 :
3         break
4     print(i)
```



continue

The continue statement is used to skip the rest of the code inside the current iteration of the loop and jump to the next iteration.

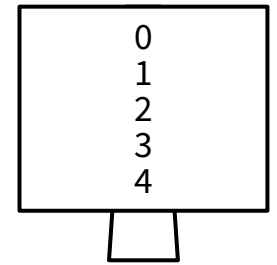
```
1 for i in range(5):
2     if i == 3 :
3         continue
4     print(i)
```



pass

The pass statement acts as a null operation or a placeholder. It does nothing when it's executed.

```
1 for i in range(5):
2     if i == 3 :
3         pass
4     print(i)
```



Enumerate

enumerate() is built-in function used to iterate over a sequence (like lists, strings, tuples) and have an automatic counter. It returns an enumerate object, which contains pairs of an index and a value.

Instead of manually handling an index using `range(len(...))`, `enumerate()` offers a cleaner and more Pythonic way to get both the index and value from an iterable.

You can easily adjust the starting index if you don't want it to begin at 0 by using the start parameter:

`enumerate(..., start=1)`

```
1 fruits = ['apple', 'banana', 'cherry']
2 for index, value in enumerate(fruits):
3     print(index, value)
```

```
0 apple
1 banana
2 cherry
```

Zip

zip() is used to combine two or more iterables (like lists, strings, tuples) element-wise. The function returns an iterator of tuples, where the first item in each passed iterable is paired together, the second item in each passed iterable is paired together, and so on.

If the passed iterables are of different lengths, zip stops creating pairs when the shortest input iterable is exhausted.

Since zip() returns an iterator, it is memory efficient and doesn't create pairs until you ask for them, like when you iterate over the zip object or convert it to a list.

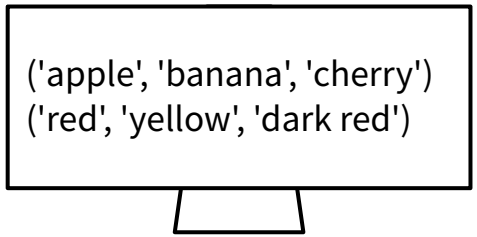
```python
fruits = ['apple', 'banana', 'cherry']
colors = ['red', 'yellow', 'dark red']
for fruit, color in zip(fruits, colors):
    print(f"The color of {fruit} is {color}.")
```

```
The color of apple is red.
The color of banana is yellow.
The color of cherry is dark red.
```

Unzipping

You can use zip() with the asterisk * operator to **unzip** a list.

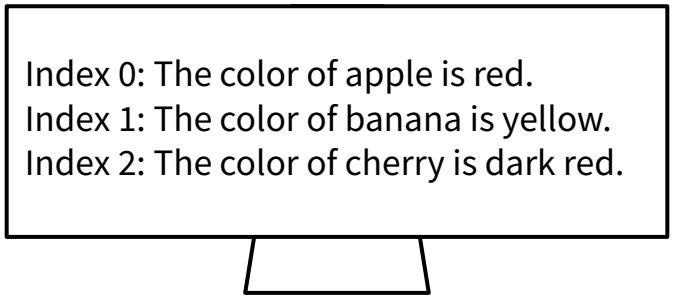
```
1 zipped_pairs = [('apple', 'red'),
2                 ('banana', 'yellow'),
3                 ('cherry', 'dark red')]
4 fruits, colors = zip(*zipped_pairs)
5 print(fruits)
6 print(colors)
```



Combining Enumerate and Zip

zip() can be combined with **enumerate()** to get both the index and values from multiple iterables.

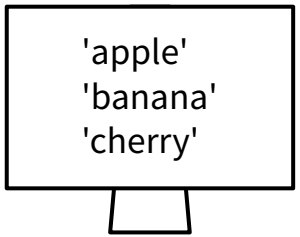
```python
fruits = ['apple', 'banana', 'cherry']
colors = ['red', 'yellow', 'dark red']
for index, (fruit, color) in enumerate(zip(fruits, colors)):
    print(f"Index {index}: The color of {fruit} is {color}.")
```



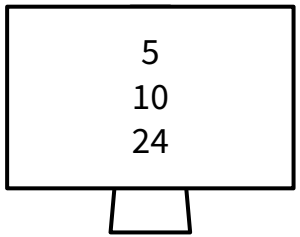
For Loops & Dictionaries

By default, when you loop through a dictionary, you're iterating over its keys. However, you can explicitly iterate through the values of a dictionary using the **.values()** method. Additionally, you can iterate iterate through the key-value pairs of a dictionary using the **.items()** method.

```
1 dictio = {"apple" : 5,
2           "banana" : 10,
3           "cherry" : 24}
4 for key in dictio :
5     print(key)
```




```
1 dictio = {"apple" : 5,
2           "banana" : 10,
3           "cherry" : 24}
4 for value in dictio.values():
5     print(value)
```



For Loops & Dictionaries

```python
dictio = {"apple" : 5,
          "banana" : 10,
          "cherry" : 24}
for key,value in dictio.items():
    print(key,value)
```

```
'apple', 5
'banana', 10
'cherry', 24
```

If you need to modify the values of the dictionary while iterating through it, you should loop through the keys and then access the values by their keys to modify them.

```python
dictio = {"apple" : 5,
          "banana" : 10,
          "cherry" : 24}
for key in dictio :
    dictio[key] += 10
print(dictio)
```

```
{'apple': 15,
 'banana': 20,
 'cherry': 34}
```

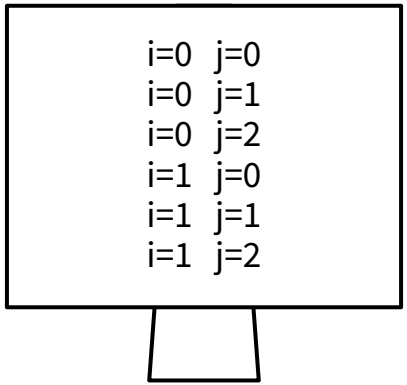
Nested For Loops

Nested for loops involve placing one for loop inside another. They can be instrumental when you need to iterate over multiple dimensions of data, such as matrices, lists within lists, or combinations of different data types. The idea behind nested for loops is simple: for each iteration of the outer loop, the inner loop will complete all its iterations.

Be cautious with the number of nested loops you use. The computational complexity increases rapidly with each level of nesting. For example, with two loops, the complexity can be O(n^2), with three it's O(n^3), and so on.

Too many levels of nested loops can make code harder to read. If you find yourself nesting loops deeply, consider if there's a more efficient or cleaner way to achieve your goal.

```python
for i in range(2) :
    for j in range(3) :
        print(f"i={i} j={j}")
```



Exercises

Now, it's time to **practice**!

Complete the following **exercise sheet**:

- 7 - (EN) Exercises - Iterables & Loops p.I

While Loops

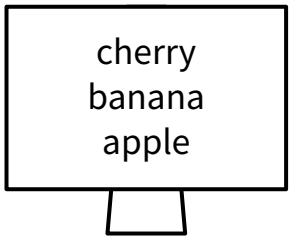
A **while loop** repeatedly executes a block of code as long as a specified condition is true. When the condition becomes false, the loop stops and the program moves to the next line of code after the loop. The condition is checked before each iteration, and if it's never true to begin with, the loop's block of code won't run at all.

Remember the factory analogy ?

The while loop is an assembly line where the worker doesn't know how many items will come down the line. They just keep working on each item until a specific condition is met, like the end of the working day or the conveyor belt running out of items.

The worker (loop) keeps performing the task (iteration) until a certain condition turns false, like the assembly line stopping.

```
1 fruits = ["apple", "banana", "cherry"]
2 while len(fruits) > 0 :
3     fruit = fruits.pop()
4     print(fruit)
```



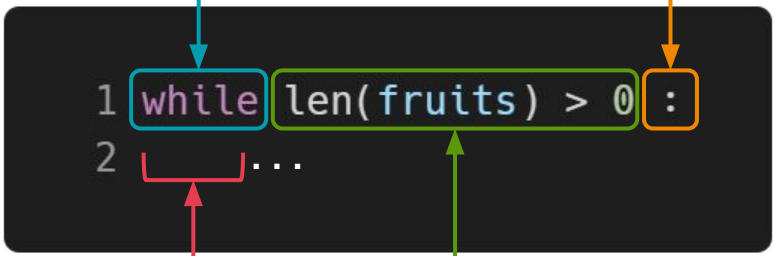
While Loops

while Keyword

*This signals the start of the loop construct. It should **ALWAYS** be in lowercase.*

Colon (:)

*The colon is a **critical aspect of Python syntax**. It indicates the end of the for statement and signifies the beginning of the block of code that will be executed repeatedly.*



Indentation

The indented block of code under the for statement represents what the program should do repeatedly until the sequence/iterable is empty.

Condition

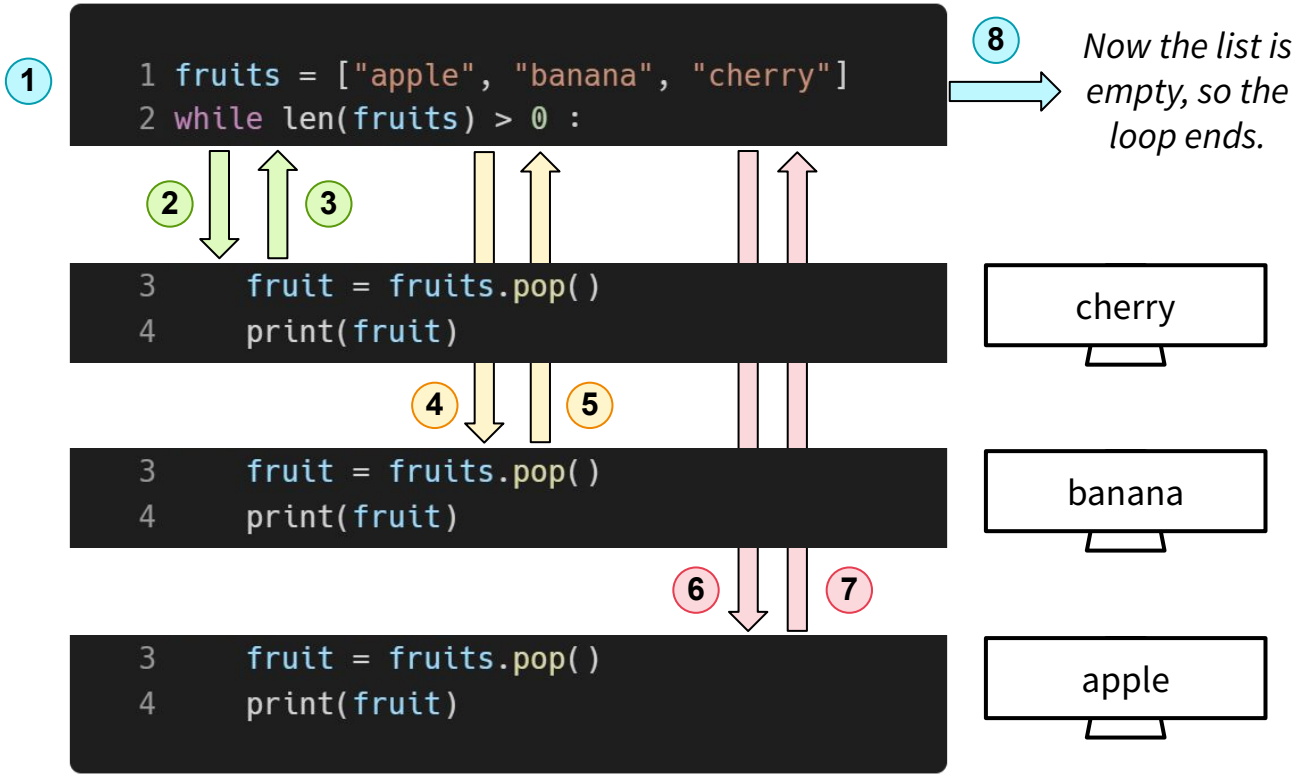
This comes after the "while" keyword. It's a boolean expression (i.e., something that evaluates to either True or False). The code inside the loop will run as long as this condition is True.

While Loops

The while loop is iterating three times because its condition is met three times.

The pop function removes one item from the list each time it is run. Thus, the fruits list has successively a length of 3, then 2, then 1.

When it reaches 0, the condition is not met anymore and thus, the loop stops.



Infinite Loops

An **infinite loop** is a loop that never stops running because the condition that controls its execution never becomes False. Essentially, it's a loop that runs forever, unless externally interrupted or terminated. An infinite loop can cause one or more CPU cores to run at full capacity, leading to high CPU usage. If the loop doesn't contain any blocking operations or delays, it'll run as fast as the processor allows, monopolizing that CPU core.

When writing code, it's essential to be cautious about unintentionally creating infinite loops, as they can lead to unresponsive programs or ***excessive resource consumption****.*

If you find yourself stuck in an infinite loop, you can typically interrupt the loop with a keyboard shortcut like Ctrl+C (in VScode Terminal) or by stopping the execution through the environment's interface.

```
1 var = 10
2 while var > 0 :
3     var += 1
4     print(var)
```

Here, the variable is NEVER lower than 0 because it keeps increasing from 10.

Thus, the condition is ALWAYS met which leads to an infinite loop.

```
11
12
13
14
15
16
17
18
19
```

Breaking the While Loop

The **break statement** can be used within a while loop. It allows you to terminate the loop even if the loop's condition remains True. It is often used in conjunction with conditional logic to break out of the loop based on some dynamic condition that arises during the loop's execution.

Imagine you're **searching for a specific value in a list**, and you want to exit the loop as soon as you find it

```
1 values = [1, 3, 5, 7, 9]
2 nb = 5
3
4 index = 0
5 while index < len(values):
6     if values[index] == nb:
7         print(f"Found {nb} at index {index}!")
8         break
9     index += 1
```

It starts the search at index 0 i.e. the beginning of the list.

If the number at the current index is matching with the searched number, the loop is stop thanks to the break statement.

If the number is not matching, the search continues by adding one to the index.

Nested While Loops

Nested while loop are very similar to nested for loops.

Searching for the location of the 0 in a matrix of an unknown size.

```
1 matrix = [[1,2,1], [4,7,3], [4,0,8]]
2
3 found = False
4 i = 0
5 while not(found):
6     j = 0
7     while j < len(matrix[i]) :
8         if matrix[i][j] == 0 :
9             found = True
10            print(f"Found at row {i+1}, columns {j+1}.")
11            break
12        j += 1
13    i += 1
```

While vs For Loops

While Loops

Use while loops **when the loop's duration depends on a condition that's dynamic and not known ahead of time**.

The while loop excels in situations where you need to keep performing an action until a particular condition changes—like when waiting for user input, reading from a file until a specific marker is found, or looping until a calculation converges to a desired accuracy.

It offers flexibility but requires careful management of the loop condition to avoid infinite loops. When you have a task driven by changing conditions rather than a fixed sequence or range, the while loop is your tool of choice.

For Loops

Use for loops **when you know in advance how many times you want the loop to iterate** or **when you're working with a sequence** (like a list, tuple, string, or range).

The for loop in Python is designed for iterating over items in a collection or sequence, making it especially suited for tasks where you wish to apply a set of operations on each element.

Its clear syntax ensures you deal with every item in a sequence, and there's no risk of unintentionally skipping elements or creating infinite loops.

Exercises

Now, it's time to **practice**!

Complete the following **exercise sheet**:

- 8 - (EN) Exercises - Iterables & Loops p.II

Comprehension in Python

Comprehension in Python is a concise way to create lists, dictionaries, or sets based on existing iterables, allowing for a more readable and often faster approach than using loops. Python does not have a direct comprehension for creating tuples.

Imagine you have a machine where you throw raw materials in at one end, set some filters or transformation rules in the middle, and out comes a finished product at the other end, all in a single step.

In Python, comprehension is like this machine, processing and transforming data from an existing iterable into a new one without the need for multiple steps or loops.

A list comprehension

```python
mylist = [i for i in range(10)]
```

A regular loop

```python
mylist = []
for i in range(10):
    mylist.append(i)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

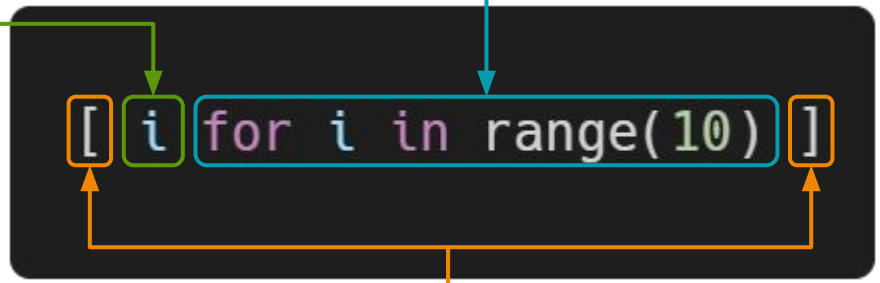
List Comprehension

i (before the for)

This is the expression that determines what each element in the new list will be. It is similar to the block of code part of a regular for loop.

for loop

This the core looping mechanism of the comprehension. It goes through an iterable one step at a time.



Outer Brackets

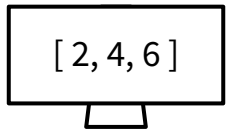
These denote that the result will be a list. All list comprehensions are enclosed in square brackets.

List Comprehension with a Filter

An if statement can be used within a list comprehension to filter out certain items.

```
[ new_value for item in iterable if condition ]
```

```python
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = [x for x in numbers if x % 2 == 0]
print(even_numbers)
```

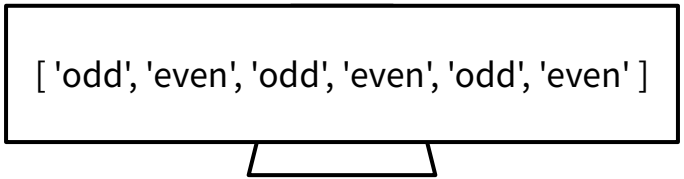


List Comprehension with a Mapper

Each element of the original list can be mapped to a new value based on some condition using a combination of the if and else statements.

```
[ new_value_true if condition else new_value_false for item in iterable ]
```

```
1 numbers = [1, 2, 3, 4, 5, 6]
2 labels = ["even" if x % 2 == 0 else "odd" for x in numbers]
3 print(labels)
```

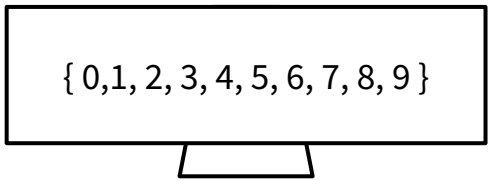


Set & Dictionary Comprehension

Set and **dictionary comprehensions** are similar to list comprehensions but are used to create sets and dictionaries, respectively, in a concise and readable way.

```
{ new_value for item in iterable }
```

```
1 unique_numbers = {x for x in range(10)}
2 print(unique_numbers)
```

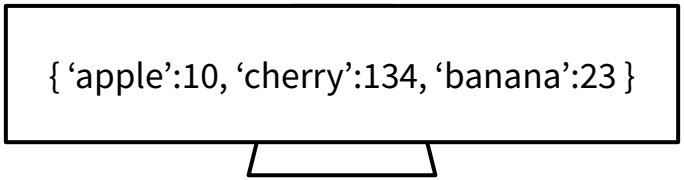


Set & Dictionary Comprehension

Set and **dictionary comprehensions** are similar to list comprehensions but are used to create sets and dictionaries, respectively, in a concise and readable way.

```
{ key: value for key,value in iterable(s) }
```

```
1 fruits = ["apple", "cherry", "banana"]
2 quantity = [10, 134, 23]
3 fruits_dict = {key:value for key,value in zip(fruits, quantity)}
4 print(fruits_dict)
```



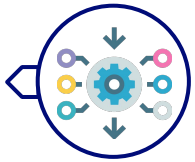
Exercises

Now, it's time to **practice**!

Complete the following **exercise sheets** in the order:

- 9 - (EN) Exercises - Iterables & Loops p.III
- 10 - (EN) Exercises - Iterables & Loops p.IV

07



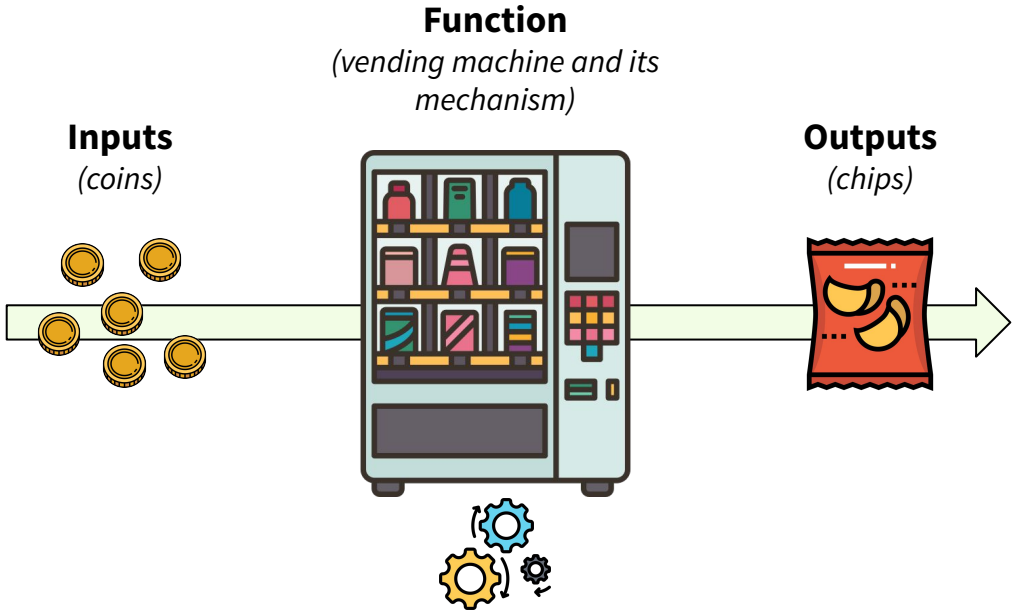
Functions

Introduction to Python

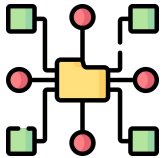
Definition

A **function** is a named set of instructions in a program that performs a specific task. When called upon, it executes these instructions and may return a result. Functions can also receive inputs, known as arguments or parameters, to influence their behavior.

Think of a function like a vending machine. You give it some coins and make a selection (these are like the inputs or arguments). In return, the vending machine provides you with a snack or drink (this is like the result or return value). The internal mechanism of the machine, which you don't see, processes your coins and selection to give you the desired item (this is like the set of instructions inside the function).



General Advantages



Organization

Functions break complex problems into smaller parts (**Modularity**).

They hide internal details, showing only their main purpose (**Abstraction**).

Efficiency

Functions can be used multiple times without rewriting the same code (**Reusability**).

Making changes is simpler as you adjust in one place (**Maintainability**).

Reliability

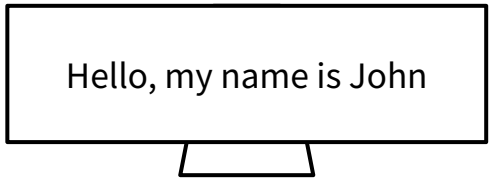
Functions limit unexpected behaviors in programs (**Reduction of Side Effects**).

Testing

Individual functions can be checked separately (**Easier Testing and Debugging**).

Python's Functions

```
1 def hello_function(name):
2     hello = f"Hello, my name is {name}"
3     return hello
4
5 result = hello_function(name = "John")
6 print(result)
```



Defining a function

This code defines the function. It specifies a set of instructions that can be executed whenever needed by referring to the function's name.

Here, line 1,2 and 3 will be executed every time the function's name is called.

Calling a function

This code calls the function. It executes the set of instructions previously defined by that function's name.

Here, it executes line 1,2 and 3 once and returns the result of the function in the result variable.

Defining a Function

def keyword

*This signals the start of the function definition. It should **ALWAYS** be in lowercase.*

function name

*The function is named hello_function. **It follows the same rules as variable names**. This is what you'll use to call the function elsewhere in your code.*

Colon (:)

The colon signifies the beginning of the block of code that will be executed when the function is called.

```
1 def hello_function (name) :
2     ...
3     return ...
```

Indentation

The indentation delimits the block of code of the function.

return keyword

*It indicates that the function will **send back a value when it's called**. After executing the return statement, the **function exits** and returns control to the caller.*

function parameters

*The variables used inside of the function. They must be specified inside of a pair of **parentheses**.*

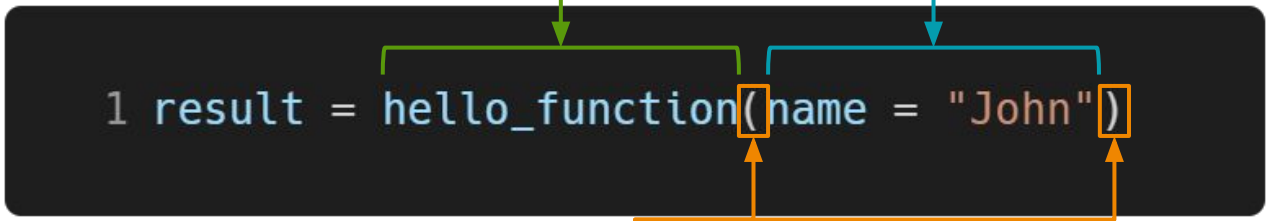
Calling a Function

function name

It calls the function by its name to execute the code it contains.

function arguments

It passes values, also known as arguments, to the parameters of the function



parentheses

Every function in Python must be called with a pair of parentheses.

Function - Step by Step

```
1 def hello_function(name):
2     hello = f"Hello, my name is {name}"
3     return hello
4
5 result = hello_function(name = "John")
6 print(result)
```

1. Python encounters the def keyword at **line 1** and begins defining a new function named hello_function.
2. Although the function is defined at this point, **line 2 and 3** are not executed yet.
3. At **line 5**, the script calls the function with the argument name = "John". Thus, Python goes back to the function to execute its inside.
4. **Line 2** is normally executed with name = “John”.
5. At **line 3**, Python encounters the return keyword. The function returns the value stored in the hello variable and exits, which means that the control returns to the main script where the function was called.
6. Going back to **line 5**, the returned value from the function is assigned to the result variable.
7. At **line 6**, Python encounters and calls the print function with the result variable as its argument. The result variable is then displayed on the console.

Function Parameters & Arguments

A function can have as many parameters of any type.

Parameters

Parameters are the variables listed inside the parentheses in a function's definition.

They act as **placeholders** for values that the function expects to receive when it's called.

Arguments

Arguments are the **actual values** that you provide to a function when you call it.

They are assigned to the function's parameters so that the function can use them in its operations.

```
1 def how_old(name, age):
2     return f"{name} is {age} years old."
3
4 result = how_old(name = "Alice", age = 25)
5 print(result)
```

Positional Arguments

```python
def integer_sum( a , b ):
    return a + b

result = integer_sum( a = 1, b = 2 )
print(result)
```

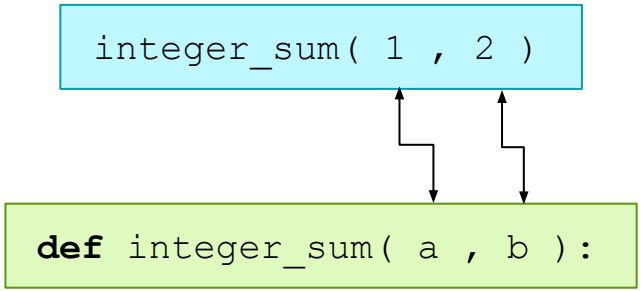
3

Positional Argument Calling

```python
def integer_sum( a , b ):
    return a + b

result = integer_sum( 1 , 2 )
print(result)
```

3

The first positional argument gets matched with the first parameter in the function's definition, the second with the second parameter, and so on.



Default Argument

In Python, function parameters can have **default values**. This means that if a value for that parameter is not provided when the function is called, the default value will be used but if a value is provided, the default value is overridden, and the provided argument is used instead.

```python
def integer_sum( a , b , c = 10 ):
    return a + b + c

result = integer_sum(1,2)
print(result)
```

13

```python
def integer_sum( a = 10 , b , c ):
    return a + b + c

result = integer_sum(1,2)
print(result)
```

SyntaxError

*Parameters with default values **must be listed after parameters without default values** in the function's definition.*

Return Statement

The **return** statement returns a value or multiple values back to the caller and makes the function exit. Multiple return statement can be used in the same function !

*Return **one** value*

```python
def add(a, b):
    return a + b
c = add(5,9)
print(c)
```

14

*Return **multiple** values*

```python
def modulus_5(a, b):
    return a%5, b%5
c = modulus_5(5,9)
print(c)
```

(0, 4) *it's a tuple !*

***Multiple return** statements*

```python
def check_ab(a, b):
    if a < b :
        return "a < b"
    elif a > b :
        return "a > b"
    else :
        return "a = b"
c = check_ab(5,9)
print(c)
```

a < b

If a function doesn't have a return statement, or if the return statement doesn't specify a value, the function will return None by default.

Return vs Print

The return statement is not a print function.

*The **print** function displays a message or a value to the console. Its primary purpose is to provide information to the user or the developer. It does not send any data back to parts of the code that called it.*

*The **return** statement, on the other hand, is used inside a function to send a result back to where the function was called from. It doesn't display anything on the console by itself.*

```python
def function_with_print():
    print("This is printed from the function.")

function_with_print()
```

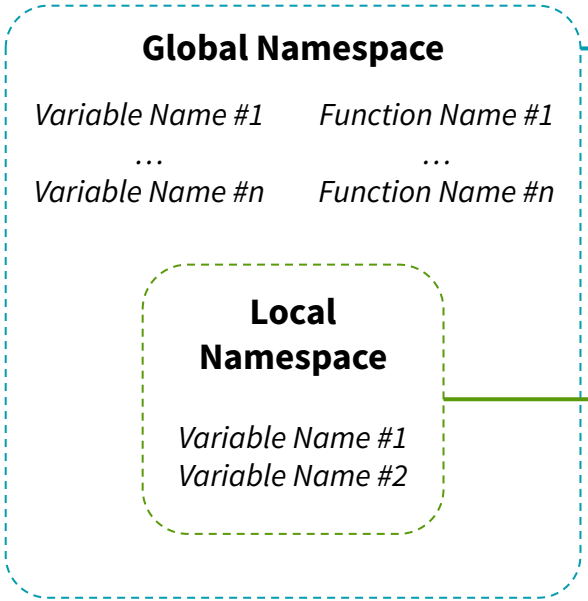
This is ***printed*** from the function.

```python
def function_with_return():
    return "This is returned from the function."

result = function_with_return()
print(result)
```

This is ***returned*** from the function.

Namespace

A **namespace** in Python is a system to ensure that all the names in a program are unique and can be used without any interference.



A **global namespace** contains names defined at the main level of a script. The **global variable** are accessible from any function in the script.

A **local namespace** is specific to the current function and contains names defined inside that function. The **local variables** are created when the function is called and cleared when the function completes.

Namespace

```
1  mystring = "Hello"
2  mylist = [1,2,3,4]
3  var = 3.1459
4
5  def myfunction(a, b, c):
6      mystring = f"Hello {a}!"
7      mylist = [i for i in range(b)]
8      var = c // 2
9      return mystring, mylist, var
10
11 result = myfunction("John", 5, 15.8)
12 print(result)
```

Global Variables

Local Variables

*In this example, the global and the local variables have the **same names but they belong to different namespaces**. Thus, the global variable is not the same as the local variable of the myfunction namespace and a change on one of them will not affect the other.*

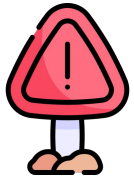
Scope of Variables

Variables created within a function are local to that function. This means they can't be accessed or used outside of that function. On the other hand, global variables, which are defined outside any function or class, are accessible from anywhere in the code.

```python
def add_student(name):
    students = []
    students.append(name)

add_student("Alice")
print(students)
```

NameError

The students list is a local variable and only exist in the function !



```python
students = []
def add_student(name):
    students.append(name)

add_student("Alice")
print(students)
```

[“Alice”]

The students list is a global variable, it can be used inside of the function !

Argument Initialization and Namespace

In Python, function definitions (including default parameter values) are evaluated only once when the function is defined, not each time the function is called. This means that the default mutable objects, like lists, are created once and then persist across function calls.

```python
def add_student( student, students = [] ):
    students.append(student)
    return students

result = add_student("John")
print(result)
result = add_student("Alice")
print(result)
result = add_student("Rose")
print(result)
```

[“John”]

[“John” , “Alice”]

[“John” , “Alice”, “Rose”]

add_student Namespace

students = [ ]

.append(...)

students = [“John”]

.append(...)

students = [“John” , “Alice”]

.append(...)

students = [“John” , “Alice”, “Rose”]

Function Docstrings

A **docstring** is text inside triple quotes (“”” … ”””) at the start of a function. It describes what the function does, what inputs it takes, what it gives back, and any special alerts about it. It's like a **quick guide** so people can understand the function without reading all the code.

```python
def add(a, b):
    """
    Add two numbers together.

    Parameters:
    - a: First number
    - b: Second number

    Returns:
    - Sum of a and b
    """
    return a + b
```

Ideally, you should include a **brief description**, a list of the **parameters**, the **returns**, and a **list of exceptions** the function might raise and under what conditions, in all docstrings.

help() built-in function

When the `help(function_name)` function is called with a function as its argument, it returns the docstring of that function.

Exercises

Now, it's time to **practice**!

Complete the following **exercise sheet**:

- 11 - (EN) Exercises - Functions p.I

Arbitrary Positional and Keyword Arguments

Arbitrary positional and keyword arguments allow functions to accept a flexible number of arguments without specifying them individually. In Python, positional arguments are called **args** and are captured as a tuple, while keyword arguments are called **kwargs** and are captured as a dictionary.

Imagine you're hosting a movie night, and you're asking your friends what snacks they'd like. You have no idea how many snacks each friend might request.

Arbitrary Positional Argument (args)

It's like each friend listing snacks they want, one after another. You write down each snack on individual sticky notes and stick them on a board in the order they are mentioned.

Arbitrary Keyword Arguments (kwargs)

Now, suppose each friend not only names a snack but also specifies a particular detail about it such as "popcorn: unsalted" or "chocolate: dark". You have small labeled boxes for each friend, where the label is the snack and the content of the box describes the detail.

In both cases, you're prepared to handle any number of requests, whether it's just the snack names or the snack names with details.

*args

```python
def sum_numbers(*args):
    print(type(args))
    return sum(args)

print(sum_numbers(1,2,3))
print(sum_numbers(1,2,3,4,5,6))
```

```
def function_name ( *args ):
```

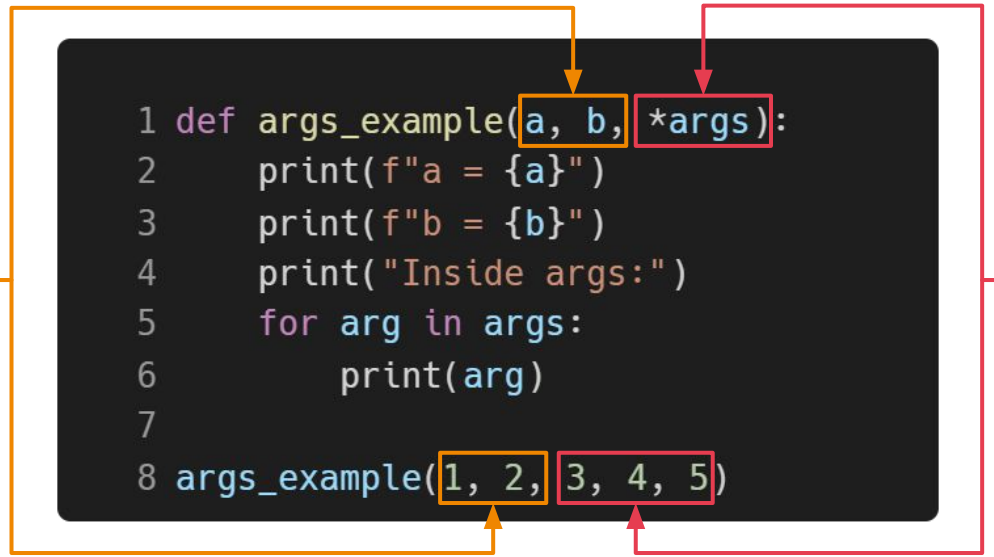
```
function_name ( var1, var2, ...)
```

```
<class 'tuple'>
6
<class 'tuple'>
21
```

All the **arguments** passed to the function will be stored into a tuple named args. This tuple can then be used into various operations.

The actual name "args" is just a convention; what's important is the * prefix. You could technically use *var or *anything, but *args is commonly accepted.

*args




```
def args_example(a, b, *args):
    print(f"a = {a}")
    print(f"b = {b}")
    print("Inside args:")
    for arg in args:
        print(arg)

args_example(1, 2, 3, 4, 5)
```

```
a = 1
b = 2
Inside args:
3
4
5
```

*When calling a function with *args and regular parameters, you need to **ensure that the regular parameters receive their values before the values meant for *args**.*

*args and Built-in Functions

Some Python built-in functions accept arbitrary positional arguments such as the print(), the min() or the sum() functions. Thus, the **print()** function allows for multiple arguments to be passed, which it then prints sequentially separated by a space.

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

Two arguments have been passed to the print() function …

print() has other parameters such as **sep** that separate the *args argument with a whitespace by default or end that adds a new line after each print by default.

```
1 print(*objects,
2       sep=' ',
3       end='\n',
4       file=None,
5       flush=False)
```

… that's because print() uses ***objects** to accept multiple unnamed arguments.

**kwargs

```python
def student_info(**kwargs):
    print(type(kwargs))
    return kwargs

result = student_info(Name="Alice",
                      Age=20,
                      Major="Computer Science")
print(result)
```

```
def function_name ( **kwargs ):
```

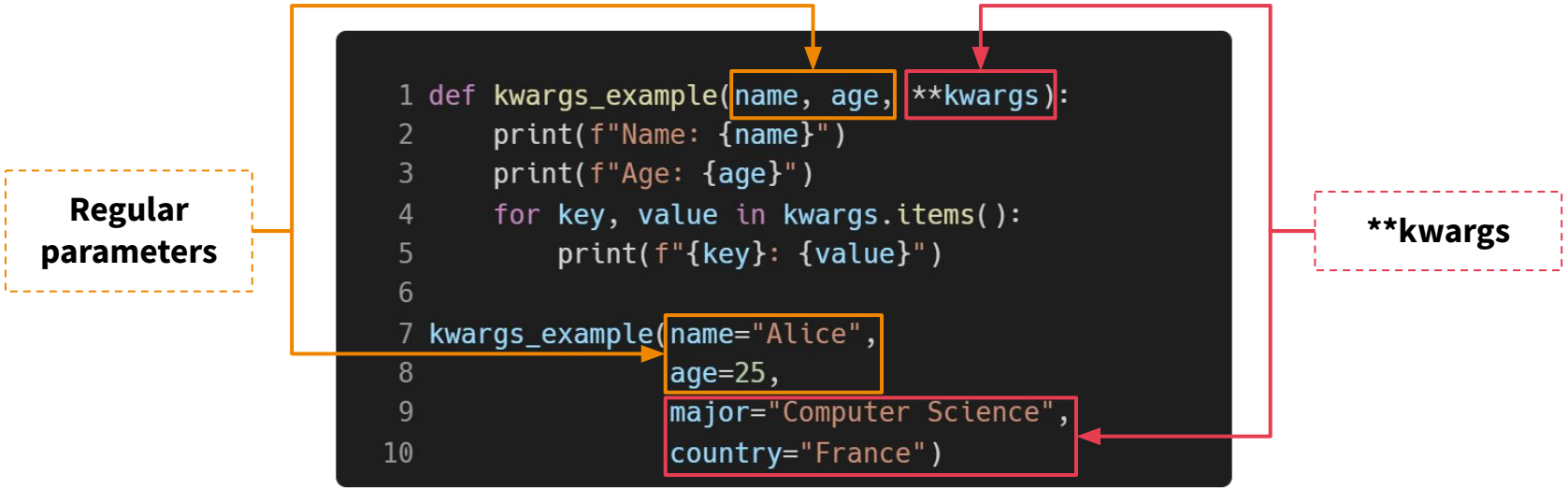
```
function_name ( param = value, ...)
```

```
<class 'dict'>
{'Name': 'Alice', 'Age': 20, 'Major': 'Computer Science'}
```

All the **named arguments** passed to the function will be stored into a dict named kwargs.

Just like with "args", "kwargs" is a convention. The ** prefix is the key. You could use **keyargs or **attributes, but **kwargs is standard.

**kwargs



Name = “Alice”
Age = 25
major: “Computer Science”
country: France”

*When calling a function with **kwargs and regular parameters, you need to* ***ensure that the regular parameters receive their values before the values meant for **kwargs****.*

Lambda Functions

Lambda functions in Python are small, anonymous functions that can have any number of arguments, but can only have one expression. They are called "anonymous" because they don't have a name like regular functions defined using the def keyword. Lambda functions are used for short, simple operations that can be defined in a single line of code.

A regular function

```
1 def squared(x):
2     return x**2
```

*A **lambda** function*

```
1 squared = lambda x:x**2
```

```
1 result = squared(5)
2 print(result)
```

25

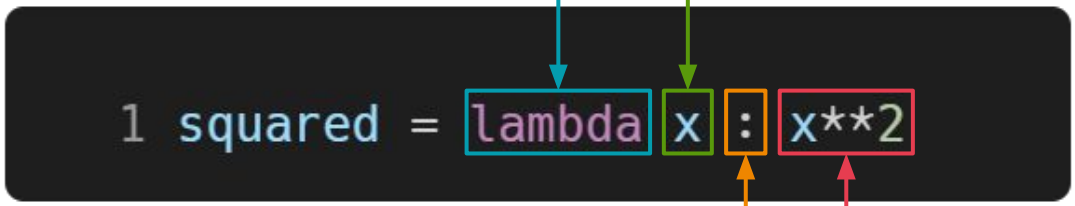
Lambda Functions

lambda keyword

A reserved keyword in Python used to create an anonymous, inline function.

arguments

Inputs to the lambda function. These are values that the lambda function will process. A lambda function can have multiple arguments.



colon (:)

A delimiter in the lambda function that separates the arguments from the expression.

expression

A single statement within the lambda function that is processed and whose result is returned when the lambda function is called.

Lambda Functions - Multiple Arguments

Multiple arguments

```
1 lamb_func = lambda x, y, z : (x**2 + y) / z
```

Multiple arguments & default argument

```
1 lamb_func = lambda x, y, z = 5 : (x**2 + y) / z
```

***args**

```
1 lamb_func = lambda *args : sum(args)
```

****kwargs**

```
1 lamb_func = lambda **kwargs : kwargs
```

Built-in and Lambda Functions

Lambda functions can be used within many built-in functions in Python, especially those that accept another function as an argument.

map(func, *iterables**)**	**Applies a function to all items** in an iterable (list, tuple, set).
filter(func, iterable**)**	**Filters the items** of an iterable (list, tuple, set) based on a function.
max(iterable, key=func, …**)**	**Finds the largest item** from the iterable based on the key function.
min(iterable, key=func, …**)**	**Finds the smallest item** from the iterable based on the key function.
sorted(iterable, key=func, reverse=False, …**)**	**Sorts the items** of an iterable based on the key function.

map(func, *iterables)

```
1 ml = [1,2,3,4]
2 func = lambda x:x**2
3 result = map(func, ml)
4 print(result)
```

<map object at … >

```
1 ml = [1,2,3,4]
2 func = lambda x:x**2
3 result = list(map(func, ml))
4 print(result)
```

[1, 4, 9, 16]

If you add multiple iterables to the map() function, the first item from each iterable is taken and fed as arguments to the function, then the second item from each iterable, and so on, until one of the iterables runs out of items.

```
1 ml1 = [1,2,3,4]
2 ml2 = [1, -1, 1, -1]
3 func = lambda x, y : x**2 * y
4 result = map(func, ml1, ml2)
5 print(list(result))
```

[1, -4, 9, -16]

filter(func, iterable)

```
1 ml1 = [1,2,3,4,5,6]
2 func = lambda x : x%2 == 0
3 result = filter(func, ml1)
4 print(result)
```

<filter object at … >

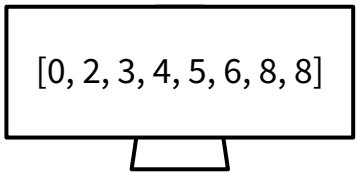
```
1 ml1 = [1,2,3,4,5,6]
2 func = lambda x : x%2 == 0
3 result = filter(func, ml1)
4 print(list(result))
```

[1, 3, 5]

sorted(iterable, key=func, reverse=False, …)

By default, the **sorted()** function returns a new list containing all items from the original iterable in ascending order. You can sort various iterables like lists, tuples or strings.

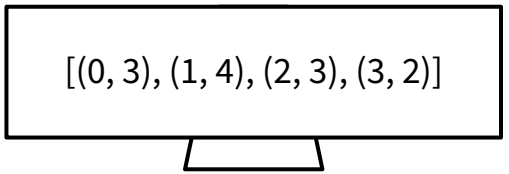
```
1 ml = [4,8,2,3,5,0,6,8]
2 result = sorted(ml)
3 print(result)
```



*The result is **always a list**.*

The **key argument** lets you provide a custom function that defines how the sorting should be performed. This function will be applied to each item in the iterable, and sorting will be based on the function's output for each item.

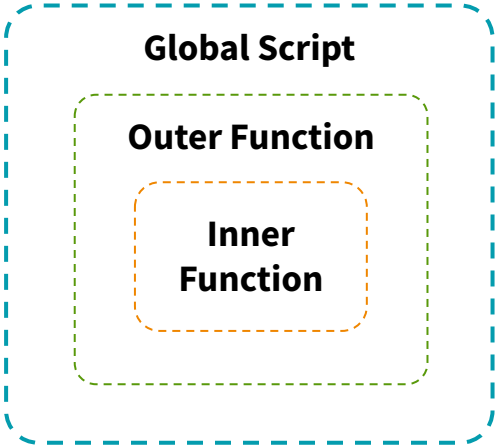
```
1 data = [(1, 4), (3, 2), (2, 3), (0, 3)]
2 result = sorted(data, key=lambda x: x[0])
3 print(result)
```



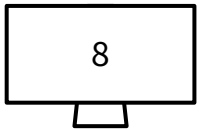
The list is sorted based on the first index of each tuple.

Nested Functions

In Python, a function defined inside another function is known as a **nested function**. The nested function, also called inner function, is only in scope within the enclosing function, meaning it cannot be called outside of its parent function.




```python
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

add_five = outer_function(5)
result = add_five(3)
print(result)
```



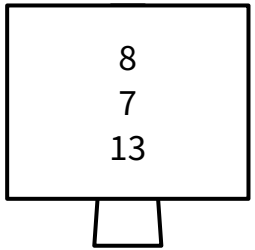
1. `inner_function` is nested inside `outer_function`.
2. `outer_function` returns `inner_function` when called.
3. The `add_five` function is essentially the `inner_function` with x set to 5.
4. When `add_five(3)` is called, it's like calling `inner_function(3)` with x set to 5.

The type of `add_five` is ***<class 'function'>.***

Nested Functions - Enclosing Scope

Instead of using global variables, nested functions can use variables from the main function they're inside of. They remember these variables even after the main function finishes.

```python
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

add_five = outer_function(5)
add_four = outer_function(4)
add_ten = outer_function(10)
print(add_five(3))
print(add_four(3))
print(add_ten(3))
```



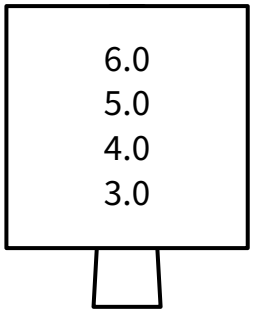
`add_five` function remember `x = 5` as a “global” variable from the `outer_function`.

Similarly, `add_four` and `add_ten` remember `x = 4` and `x = 10` from the `outer_function` respectively.

Nested Functions - Enclosing Scope

```python
def mean():
    samples = []
    def compute_mean(number):
        samples.append(number)
        return sum(samples)/len(samples)
    return compute_mean

update_mean = mean()
print(update_mean(6)) 1
print(update_mean(4)) 2
print(update_mean(2)) 3
print(update_mean(0)) 4
```

The `samples` list is a global variable of the compute_mean function. Thus, it is updated and stay in "memory" of the function through the successive calls.



1. [6] → 6 / 1 → **6.0**
2. [6 , 4] → 10 / 2 → **5.0**
3. [6 , 4 , 2] → 12 / 3 → **4.0**
4. [6 , 4 , 2 , 0] → 12 / 4 → **3.0**

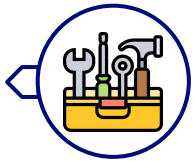
Exercises

Now, it's time to **practice**!

Complete the following **exercise sheet**:

- 12 - (EN) Exercises - Functions p.II

08



Libraries

Introduction to Python

Python Library

A Python **library** is a collection of functions and methods used to perform many actions without writing code from scratch. They are typically written in Python but they can also have components in other languages like C or C++ for performance optimization. They can be categorized into two main types.

Standard Library

They are already built in Python **without the need to install them**. They generally provide basic functionalities and are designed to be universally applicable for Python.

The standard library contained over 200 modules (see documentation).

Third-Party Libraries

They are developed by independent developers outside of the Python core development team. These libraries provide additional functionalities, enhancements or specialized capabilities that are not available in the standard library.

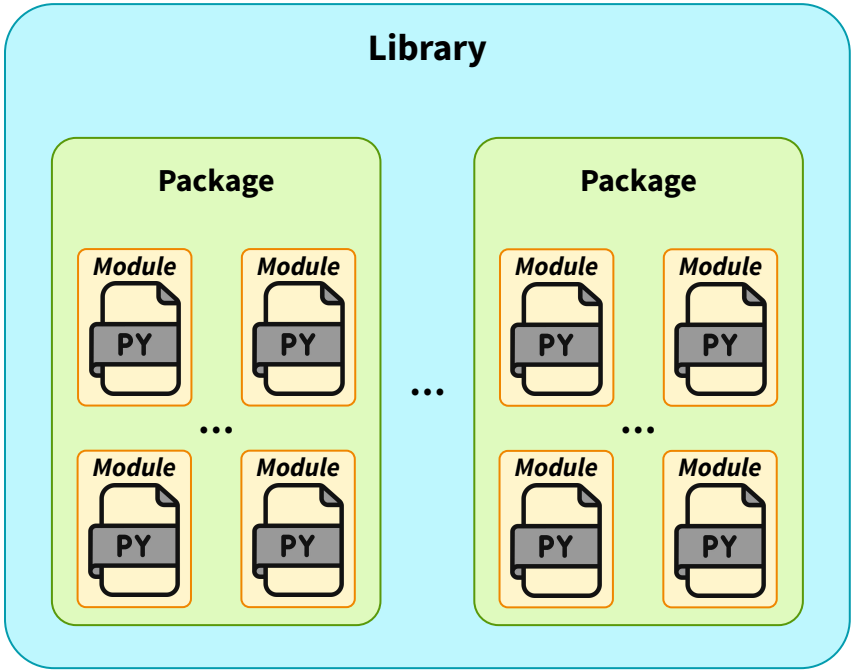
More than 500 000 packages are available on PyPi.

Modules ? Packages ? Libraries ?

A **module** is a single file with the suffix **.py** containing Python definitions and statements.

A **package** is a way of organizing related modules into a directory hierarchy. Thus, a package is a collection of modules.

A **library** is a collection of modules and/or packages. That said, a library can be as simple as a single module.



Importing a Library

```python
import math

radius = 3
area = math.pi * radius**2

print(f"Area is {area}")
```

```
import library_name
```

```python
from math import pi

radius = 3
area = pi * radius**2

print(f"Area is {area}")
```

```
from library_name import function
```

The import is not limited to functions. You can import classes or variables as well.

*Avoid using **from library_name import *** at all costs. It can lead to significant issues like namespace pollution, where the code gets flooded with too many names from the imported module, increasing the risk of accidentally overwriting existing functions or variables.*

Aliases

If a library name is too long, it is possible to import it with an **alias** using the as keyword.

Without alias

```
1 import matplotlib.pyplot
2 ml = [0,1,2,3,4,5]
3 matplotlib.pyplot.plot(ml, list(map(lambda x:x**2, ml)))
4 matplotlib.pyplot.show()
```

With alias

```
1 import matplotlib.pyplot as plt
2 ml = [0,1,2,3,4,5]
3 plt.plot(ml, list(map(lambda x:x**2, ml)))
4 plt.show()
```

```
import library_name as alias
```

It is also possible to rename a class, a function or a variable using an alias with:

from library_name **import** function_name **as** alias

Standard Library – A non-exhaustive List

Concurrent Execution	threading, multiprocessing …
Cryptography	hashlib, secrets …
Data Compression	zlib, gzip, tarfile …
Data Persistence	pickle, dbm, sqlite3 …
Data Types	datetime, collections, array …
Development Tools	pdb, unittest, timeit, cProfile …
File & Directory Access	os.path, fileinput, pathlib …
File Formats	csv, json, configparser …
Generic OS Services	os, logging, argparse …
Graphical User Interface	tkinter …
Importing Modules	importlib, pkgutil …

Internationalization	gettext, locale …
Internet Data Handling	urllib, http …
Language Services	ast, tokenize …
Markup Processing	html, xml …
Math	math, random, statistics …
Multimedia Services	audioop ,wave, colorsys …
Networking & Internet	socket, asyncio, email …
Package Distribution	venv, zipapp …
Program Frameworks	cmd, shlex …
Runtime Services	sys, importlib …
Text Processing	string, re, textwrap …

The exhaustive list of Python's standard libraries can be found here.

Math – Documentation

The **math** library is a module that provides a set of mathematical functions and constants.

```python
import math

## Basics
pi = math.pi
sqrt = math.sqrt(pi)
power2 = math.pow(pi, 2)
floor = math.floor(pi)   # round down
ceil = math.ceil(pi)     # round up

## Trigonometry
theta = 45
rad = math.radians(theta)
sin = math.sin(rad)
cos = math.cos(rad)
tan = math.tan(rad)
theta = math.degrees(rad)

## Power & Logorithm
x = 5
log = math.log(x) # or log(x, base)
exp = math.exp(x)
```

```python
import math

val = math.inf
is_inf = math.isinf(val)  # True
nan = math.nan
is_nan = math.isnan(nan)  # True

## Others
p = [1,2,3,4]
q = [5,6,7,8]
n, k = 10, 5

fact = math.factorial(5)
gcd = math.gcd(15,20)
perm = math.perm(n, k) # n!/(n-k)!
prod = math.prod(p, start=1)
sprod = math.sumprod(p, q)
```

Itertools – Documentation

The **itertools** module provides a set of fast and memory-efficient tools for looping.

```python
from itertools import combinations

p = ['a', 'b', 'c']

for combi in combinations(p, 2):
    print(combi, end=' ')
```

```
('a', 'b') ('a', 'c') ('b', 'c')
```

product('ABC', repeat=2)	AA AB AC BA BB BC CA CB CC
permutations('ABC', 2)	AB AC BA BC CA CB
combinations('ABC', 2)	AB AC BC
combinations_with_replacement('ABC', 2)	AA AB AC BB BC CC

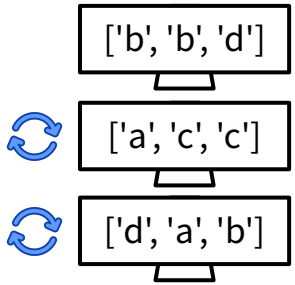
Random – Documentation

The **random** module provides tools for generating pseudo-random numbers for various distributions. This library shouldn't be used for cryptography because the pseudo-random algorithms used there are not secured enough.

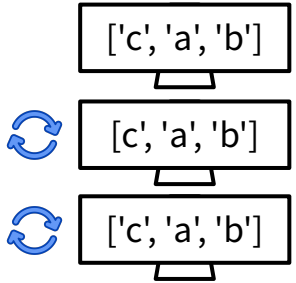
```python
import random

ml = ['a', 'b', 'c', 'd']
k = 3

# Random float between 0 and 1 excluded
x1 = random.random()
# Random integer between 1 and 10
x2 = random.randint(1, 10)
# Random float between 1 and 10
x3 = random.uniform(1, 10)
# Random float from normal distribution N(0,1)
x4 = random.gauss(mu=0, sigma=1)
# Shuffle list !! IN-PLACE !!
random.shuffle(ml)
# Random element from list
x5 = random.choice(ml)
# k random elements from list
x6 = random.choices(ml, k=k)
# k random unique elements from list
x7 = random.sample(ml, k=k)
```

Pseudo-random algorithm can generate a sequence of numbers that looks like random. However, every sequence of numbers generated by a pseudo-random algorithm is determined by a "**seed**". Given the same seed, the PRNG will always produce the same sequence of numbers.

```python
import random
ml = ['a', 'b', 'c', 'd']
print(random.choices(ml, k=3))
```




```python
import random
random.seed(42)
ml = ['a', 'b', 'c', 'd']
print(random.choices(ml, k=3))
```



Datetime – Documentation

The **datetime** module provides tools for manipulating dates and times such as calculating durations, formatting dates / times for display or parsing date strings from user input.

date & time

```python
from datetime import date, time
d = date(2022, 1, 1)
t = time(12, 30)
dt = datetime(2024, 1, 1, 12, 30)
```

datetime

Combines date and time into a single object.

```python
from datetime import datetime
now = datetime.now()
```

timedelta

```python
from datetime import date, time, timedelta
d = date(2024, 1, 1)
today = date.today()
yesterday = today - timedelta(days=1)
```

strftime & strptime

```python
from datetime import datetime
# From datetime object to string
date = datetime.now()
formatted = date.strftime("%Y-%m-%d %H:%M:%S")

# From string object to datetime
date = "2022-01-01"
parsed = datetime.strptime(date, "%Y-%m-%d")
```

Os – Documentation

The **os** modules provide ways to interact with the operating system.

```python
import os

# Current directory
cdir = os.getcwd()

# Files & directories in current directory
list_dir = os.listdir(cdir)

# Creating a New Directory
new_dir = "dir_name"
if not os.path.exists(new_dir):
    os.makedirs(new_dir)

# Renaming a directory
os.rename("dir_name", "new_dir_name")

# Removing a directory
if os.path.exists("dir_name"):
    os.rmdir("dir_name")
```

The **os** module can be categorized into several key areas of functionality:

- **File / directory management, properties and metadata** – creating, removing, renaming and moving files and directories, accessing file properties like size, modification date, permissions …
- **Environment interaction** – Accessing and modifying environment variables.
- **Executing system commands** – running shell commands or external programs
- **Accessing system information** – memory usage, system configuration details, hardware information …

Third-Party Libraries

Web Development

Data Science

Game & GUI

Artificial Intelligence

And more …

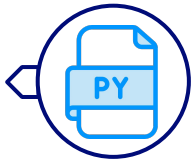
Exercises

Now, it's time to **practice**!

Complete the following **exercise sheet**:

- 13 - (EN) Exercises - Libraries

09



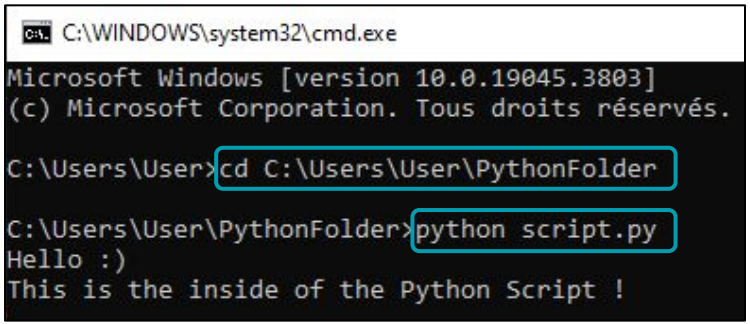
Script Handling and Notebooks

Introduction to Python

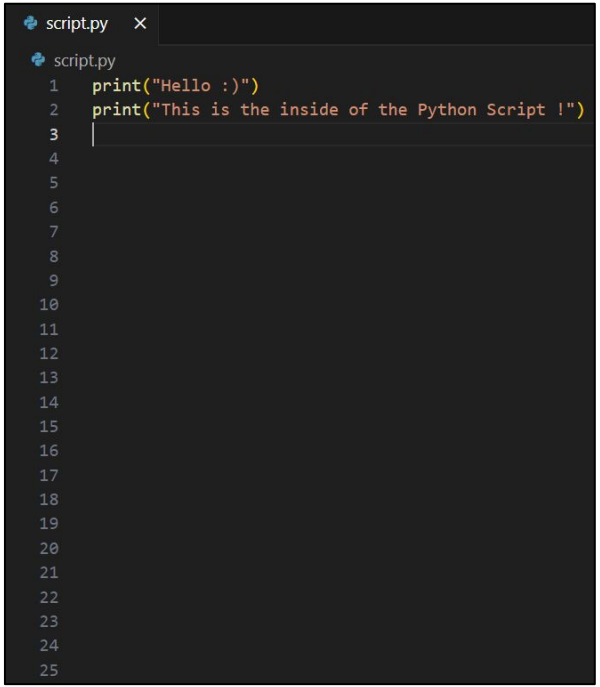
Python Script

A **Python script** is a file containing Python code that is intended to be directly executed. They are saved with a **.py** extension.

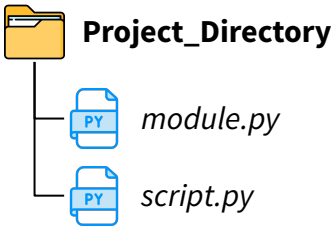
Running a Python script can be done using an **IDE** (Integrated Development Environment) or a **terminal**.



*First, navigate to the directory containing your script using the **cd path_to_directory** command line. Then, run the script by using either the **python script_name.py** or **python3 script_name.py** command line*



Importing Python Scripts



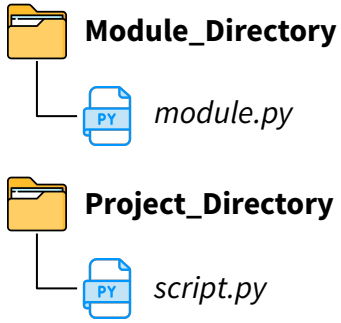
module.py

```
1 def myfunction():
2     return "I am a function from module.py"
```

script.py

```
1 from module import myfunction
2 print(myfunction())
```

I am a function from module.py



module.py

```
1 def myfunction():
2     return "I am a function from module.py"
```

script.py

```
1 import sys
2 sys.path.append("...\Module_Directory")
3 from module import myfunction
4 print(myfunction())
```

if __name__ == "__main__"

In Python, every module has a built-in attribute called **__name__**. The value of __name__ is set to **“__main__”** when the module is run as the main program. However, if the module is imported into another script, the __name__ attribute is set to the module's name. This part often contains test code or a demo of the functions in the script. It prevents certain code from being run when the module is imported into another script.

module.py

```python
def myfunction():
    return "Function in module"

if __name__ == "__main__":
    print("module is being run directly")
    print(myfunction())
```

…\Project_Directory > python module.py

```
module is being run directly
Function in module
```

script.py

```python
import module
print(module.myfunction())
```

…\Project_Directory > python script.py

```
Function in module
```

Python Project Structure

```
my_project/
│
├── main.py              # Main script to run your project
├── config.py            # Configuration settings and constants
├── requirements.txt     # List of dependencies
│
├── src/                 # Source code folder for all your scripts
│   ├── __init__.py      # Makes src a Python package
│   ├── module1.py       # Python script/module
│   ├── module2.py       # Another script/module
│   └── ...
│
├── data/                # Data files (could be .csv, .json, etc.)
│
├── tests/               # Unit tests
│   ├── __init__.py
│   ├── test_module1.py
│   └── ...
│
└── docs/                # Documentation
    ├── README.md
    └── ...
```

The **config.py** file is typically used to store configuration settings. These settings might include constants, file paths or settings.

The **requirements.txt** file is used for specifying what Python packages are required to run the project. It's a text file that lists them with their version numbers.

Python_Project_Demo > requirements.txt

```
1  flask==1.1.2
2  requests>=2.24.0
3  pandas
```

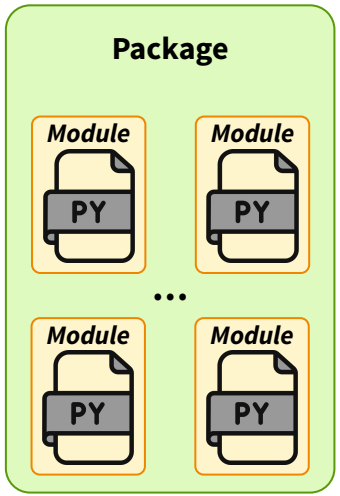
To install all the dependencies listed in requirements.txt, you run:

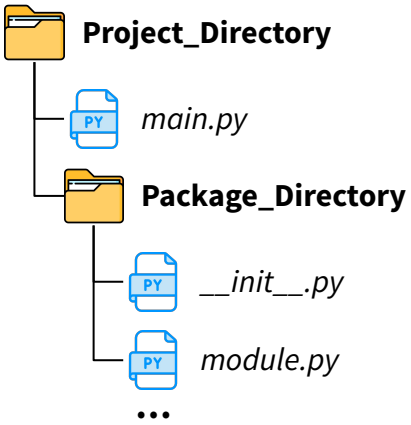
...\env > pip install -r requirements.txt

Package & __init__.py

The presence of an **__init__.py** file in a directory indicates that the directory should be treated as a package. This file can be empty or contain Python code. When a package is imported, the code in its __init__.py file is executed.

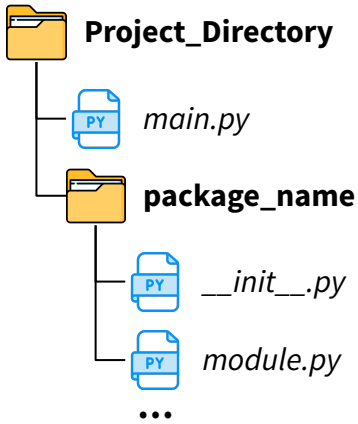
It should mainly be used for package initialization and import management. Additionally, functions and variables defined in __init__.py are available at the package level.





The presence of __init__.py is not strictly required to define a package since Python 3.3. However, *explicit is often better than implicit* and using __init__.py can help avoid confusion.

Package & __init__.py



module.py

```python
def myfunction():
    return "Function in module"

if __name__ == "__main__":
    print("module is being run directly")
    print(myfunction())
```

__init__.py

```python
from package_name.module import myfunction
```

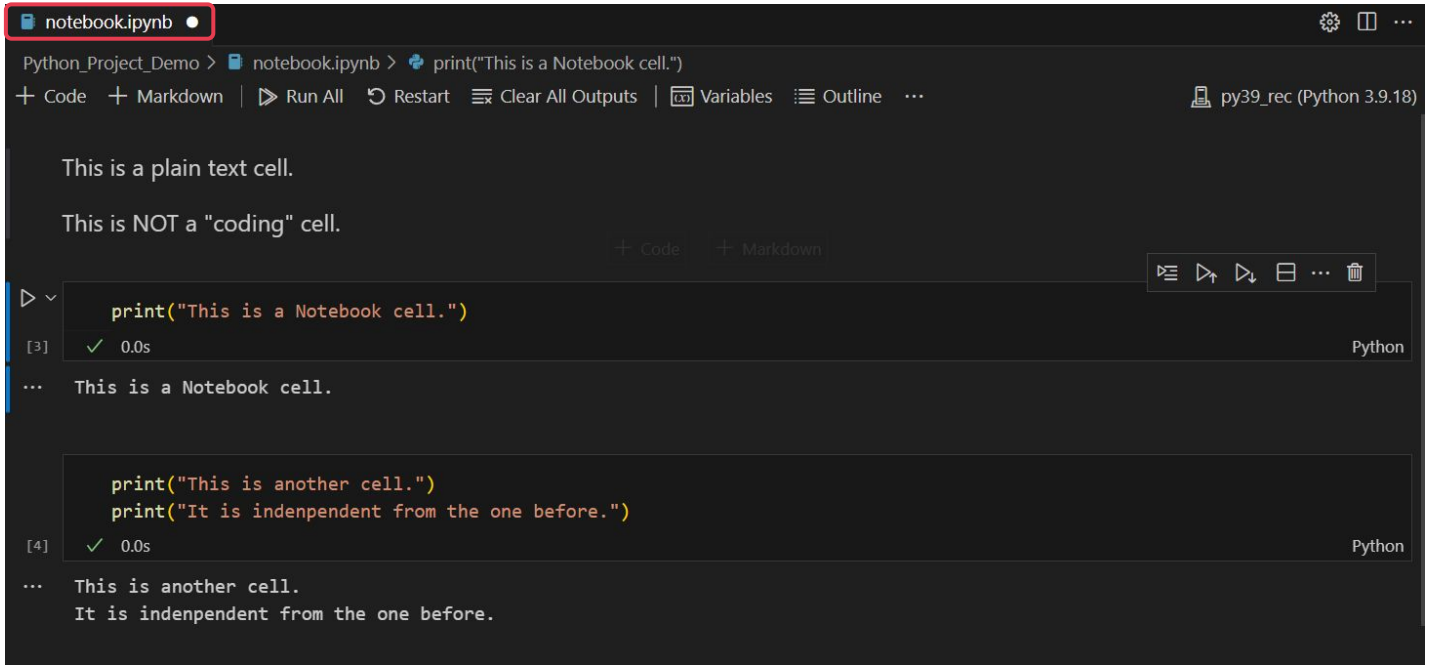
main.py

```python
from package_name import myfunction
print(myfunction())
```

```
Function in module
```

Notebooks

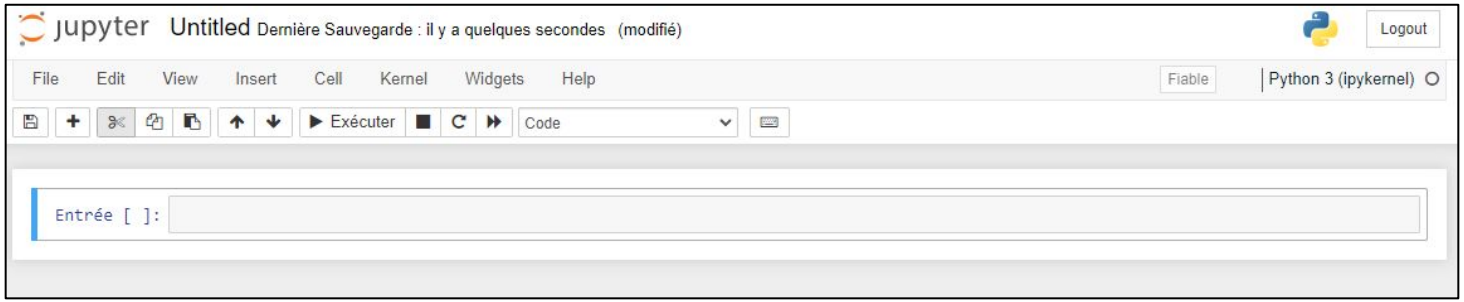
Notebooks refer to an interactive document that allows to mix executable code, text, images, visualizations and other types of media in a single document.



Jupyter Notebooks

One of the most popular types of notebooks is the **Jupyter Notebook**. It can be installed using the `pip install notebook` command line or by installing the Anaconda distribution.

Jupyter Notebooks can be run using the `jupyter notebook` command line or via Anaconda.



Advantages

Interactivity, support for multimedia and rich text, great for teaching and for **data science**.

Disadvantages

Not ideal for **software development** because it's difficult to track changes, it lacks of modularity, it's hard to test and it has performance concerns.

10

Errors, Exceptions and Debugging

Introduction to Python

Python Errors & Exceptions

A **Python error** is a problem in a piece of code that causes it to fail. These errors can come in various forms. Each type of error signifies a different kind of issue in the code. Being able to quickly identify and fix errors can significantly reduce development time and cost.

In Python, errors are generally categorized into two main types: **syntax errors** and **exceptions**.

Syntax Errors

Syntax errors are **detected** during the parsing stage which happens **before the code execution**. If your code has a syntax error, it won't run at all until the error is fixed.

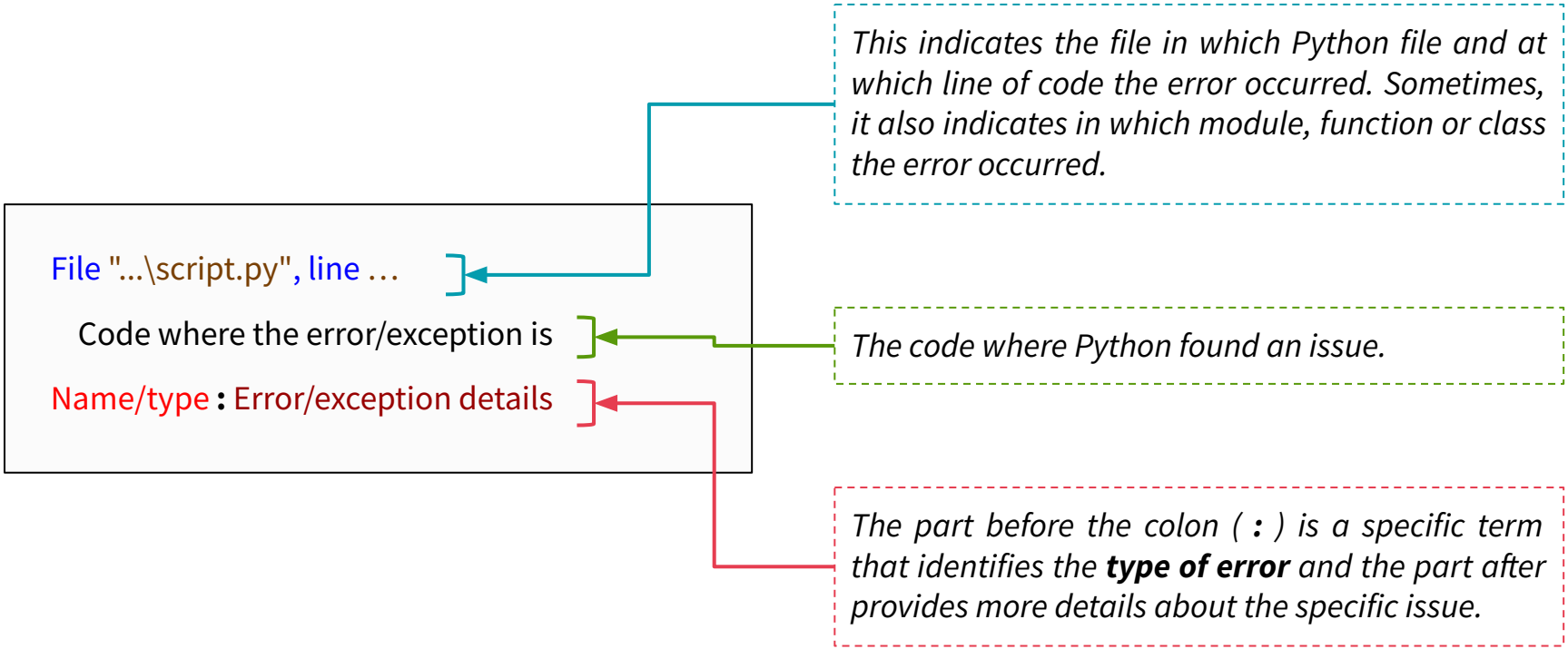
That is why tools like IDEs can often detect syntax errors even before you run your code.

Exceptions

Exceptions are errors that are **detected during the execution of code**. Unlike syntax errors, exceptions can occur even if your code is syntactically correct. They indicate that something went wrong during the execution of a program.

Python has built-in exceptions and custom exceptions can also be defined.

Python Errors & Exceptions



Syntax Error

Occurs when Python encounters code that does not conform to the syntax of the language. This could happen because of a **missing colon**, **incorrect indentation**, **unclosed or mismatched brackets/parentheses/quotes/…**, **misplaced or missing operators**, and so on.

```
1 def my_function( )
2     print("Hello, World!")
```

```
File "...\script.py", line 1
  def my_function( )
                   ^
SyntaxError: invalid syntax
```

```
1 mytuple = (2,4,6,8)
2 print(type(sorted(mytuple))
```

) ?

```
File "...\script.py", line 3

  ^
SyntaxError: unexpected EOF while parsing
```

The error is on line 3 because Python was expecting that missing parenthesis to be on line 3.

Common Exceptions – Name Error

Occurs when a name such as a variable or a function name is not found. This could happen because of **typos**, the **reference to a name before it is defined**, **function/class/module scope issues**, and so on.

```
1 print(my_var)
2 my_var = "Hello"
```

```
File "...\script.py", line 1, in <module>
  print(my_var)
NameError: name 'my_var' is not defined
```

```
1 mylist = [2,4,6,8]
2 print(myllist)
```

```
File "...\script.py", line 2, in <module>
  print(myllist)
NameError: name 'myllist' is not defined
```

Common Exceptions – Type Error

Occurs when an operation or function is applied to an object of an inappropriate type. This could happen because of a code that **combines incompatible types** (an integer with a string, a list with a float …), an **incorrect number of arguments in a function call**, a **code that tries to iterate over a non-iterable** such as an integer, a code that **uses a method on an unsupported type** (`.append()` on an integer …), and so on.

```
1 nb = 5
2 result = "Number: " + nb
```

```
File "...\script.py", line 2, in <module>
  result = "Number: " + nb
TypeError: can only concatenate str (not "int") to str
```

```
1 def add(a, b):
2     return a + b
3 result = add(10)
```

```
File "...\script.py", line 3, in <module>
  result = add(10)
TypeError: add() missing 1 required positional argument: 'b'
```

Common Exceptions – Key & Index Errors

The **index error** occurs when a code try to access an index in a sequence that doesn't exist.

The **key error** occurs when a code try to access a key in a dictionary that doesn't exist.

```
1 dictio = {'a': 1, 'b': 2}
2 print(dictio['c'])
```

```
File "...\script.py", line 2, in <module>
  print(dictio['c'])
KeyError: 'c'
```

```
1 ml = [1,2,3]
2 for i in range(4):
3     ml[i] += 1
```

```
File "...\script.py", line 3, in <module>
  print(ml[i])
IndexError: list index out of range
```

Common Exceptions – Value Error

Occurs when a function receives an argument of the right type but an inappropriate value. This generally happens because of a code that uses an **invalid value with a function or a method**.

```
1 mystr = "Hello"
2 myint = int(mystr)
```

```
File "...\script.py", line 2, in <module>
  myint = int(mystr)
ValueError: invalid literal for int() with base 10: 'Hello'
```

```
1 mylist = [1, 2, 3]
2 mylist.remove(4)
```

```
File "...\script.py", line 2, in <module>
  mylist.remove(4)
ValueError: list.remove(x): x not in list
```

Exception Handling – try and except

In Python, **try** and **except** are keywords used in exception handling – dealing with unexpected situations in the code. An error in the try block of code will be catched and handled by the corresponding except block of code.

```
1 while True:
2
3     try:
4         x = int(input("Enter a number: "))
5         break
6
7     except ValueError:
8         print("Not a valid number.")
```

The **try block** is used to wrap a section of code that might raise an exception. If any statement within the try block raises an exception, the rest of the block is skipped and Python looks for an except block to handle the exception.

The **except block** is used to handle the exception that was raised in the try block.

In this example, if a ValueError exception is caught, the code inside the except block is executed.

Exception Handling – try and except

```
1  try:
2      f = open('myfile.txt')
3      s = f.readline()
4      i = int(s.strip())
5
6  except OSError as err:
7      print(f"OS error: {err}")
8
9  except ValueError:
10     print("Could not convert data to an integer.")
```

The **first except block** is specifically designed to catch OSError exceptions. The `as err` part assigns the exception to the variable err in order to access its details.

The **second except block** catches ValueError exceptions.

The excepts are checked in the order they are defined. When an exception is raised in the try block, Python goes through each except in sequence until it finds a matching exception.

Exception Handling – Exception Hierarchy

```python
try:
    f = open('myfile.txt')
    s = file.readline()    # NameError
    i = int(s.strip())

except OSError as err:
    print(f"OS error: {err}")

except ValueError:
    print("Could not convert data to an integer.")

except BaseException as err:
    print(f"Unexpected {err = }")
```

In Python, **BaseException** is the base class for all built-in exceptions. It's at the top of the exception hierarchy. This means it will catch almost every exception that can occur. However, catching BaseException is generally not recommended (see next slide).

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
      │    ├── ChildProcessError
      │    ├── ConnectionError
      │    │    ├── BrokenPipeError
      │    │    ├── ConnectionAbortedError
      │    │    ├── ConnectionRefusedError
      │    │    └── ConnectionResetError
      │    ├── FileExistsError
      │    ├── FileNotFoundError
      │    ├── InterruptedError
      │    ├── IsADirectoryError
      │    ├── NotADirectoryError
      │    ├── PermissionError
      │    ├── ProcessLookupError
      │    └── TimeoutError
```

Full hierarchy here (click)

Exception Handling – Good Practice and Misuses

Using try and except blocks for exception handling is a standard practice in Python and many other programming languages. It is a good practice in cases where known errors are expected and must be catched or isolated to prevent the entire program from crashing. However, it should not be used for the following cases:

Catching too broad exceptions

Using an except without any specified error or catching everything with `except Exception:` can lead to catch unexpected exceptions and hide errors.

If something goes wrong in the code, it will be harder to debug.

Silencing exceptions

Catching exceptions and not handling them properly by just skipping them for example can lead to silent failures.

Thus, if something goes wrong in the code, it will be harder to debug.

Don't replace conditional statements by catching exceptions unless you can’t do otherwise. Exceptions should remain exceptional.

Exception Handling – Good Practice and Misuses

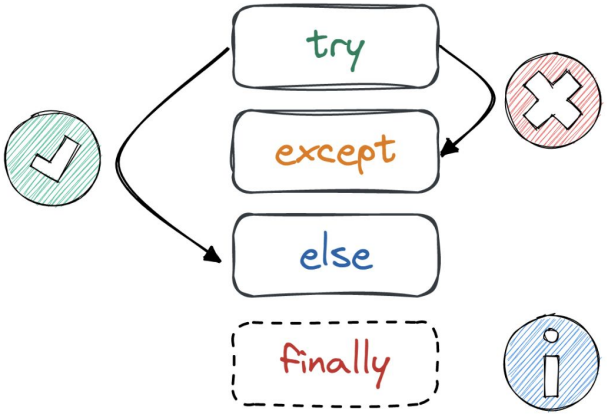
In the scenario below, the potential error is predictable and easily checkable without the need to an exception handling. Using try and except in this context is like using a hammer to crack a nut – it's an **overuse**. Exceptions should be used for handling exceptional, less predictable conditions.

```python
mylist = ["Example", "of", "Misuse"]
while True:
    index = int(input("Enter an index: "))

    try:
        value = mylist[index]
        print(value)
        break

    except IndexError:
        print("Index out of range!")
```

```python
mylist = ["Example", "of", "Misuse"]
while True:
    index = int(input("Enter an index: "))

    if index < len(mylist) and index >= 0 :
        value = mylist[index]
        print(value)
        break

    else :
        print("Index out of range!")
```

Exception Handling – try, except, else and finally

The **else** in a try-except block is executed if the code inside the try block does not raise an exception. The **finally** in a try-except block is designed to be executed regardless of whether an exception was raised in the try block and irrespective of how the try and except blocks are exited. It is generally used to ensure that cleanup actions such as closing connections, releasing file handles, or cleaning up temporary objects are always executed.



Source : datacamp.com

```python
f = None
try:
    f = open('myfilfe.txt')

except OSError as err:
    print(f"OS error: {err}")

else :
    print(f"File has {len(f.readlines())} lines")

finally :
    if f : f.close()
```

raise

The **raise** statement is used to trigger an exception intentionally.

The raise statement can also be used in the except block to propagate the exception. This behavior is called **Exception Chaining**.

It can be seen as a way of delaying the handling of the exception in order to perform some immediate actions related to the exception such as cleanup or logging before raising the exception.

```python
def divide(x, y):
    if y == 0:
        raise ValueError("Division by zero !")
    return x / y
```

```python
try:
    f = open('myfile.txt')
    line = f.readline()
    nb = int(line)

except ValueError as err:
    f.close()
    raise ValueError(err) from None
```

By default, the raise statement will include the traceback of the original exception. The from None syntax is used to raise a new exception without this chaining behavior.

Warnings

Python issues **warnings** during code execution to alert programmers about issues that could lead to unwanted behavior. These warnings are different from syntax errors and exceptions as they don't stop the execution of the program.

SyntaxWarning	Raised for suspicious syntax. It often indicates a misusage in the code.
DeprecationWarning	Indicates that a feature, function or method is deprecated and may be removed in future versions.
RuntimeWarning	Warns about potentially problematic runtime behavior such as division by zero.

Some common Warnings in Python
Full list here

```
1 a = 5
2 if a is 5 :
3     print("a is 5")
```

```
File "...\script.py":2: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if a is 5 :
a is 5
```

This is the output of the code.
The warning didn't stop the execution.

Debugging in Python

Python provides several tools and techniques to facilitate debugging.

print statement

Inserting print() statements in the code can help track the flow of execution and inspect the values of variables at different points.

Best for simple script debugging where setting up a debugger is overkill.

assert statement

Testing a condition. If the condition is False an AssertionError exception is raised with an optional message.

Best for quick debugging during development to catch bugs early.

logging library

Using the built-in logging module to record various levels of information about the execution of the program.

Best for tracking the behavior of a program over time, especially in production environments.

pdb library

pdb is a standard interactive debugger that allows to set breakpoints, step through code, inspect stack frames …

Best for more complex debugging to understand the flow of execution.

assert

When Python encounters an **assert statement** it evaluates the condition.

- If condition is True, the program continues to execute as normal.
- If condition is False, it raises an **AssertionError** exception with the specified message.

```
assert condition, message
```

```
1 def calculate_average(numbers):
2
3     assert numbers, "list should not be empty"
4
5     assert all(isinstance(num, (int, float)) for num in numbers), "Elements must be numbers"
6
7     return sum(numbers) / len(numbers)
```

At line 3, if numbers is empty then the condition is False and the assert statement raises an AssertionError.
At line 5, if at least one element of the list is not either a float or an integer then the condition is False and the assert statement raises an AssertionError.

logging

The **logging** library is used for logging messages from applications and provides a way to configure how these messages are handled – including their severity level, destination and format.

It is more suitable for production environments than print because it provides a much more flexible and controlled approach to outputting diagnostic information, it supports different severity levels and it allows output to various destinations and formats.

```python
import logging
logging.basicConfig(filename='myfile.log', encoding='utf-8', level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
logging.error('And non-ASCII stuff, too, like Øresund and Malmö')
```

[Python logging guide]

myfile.log

```
DEBUG:root:This message should go to the log file
INFO:root:So should this
WARNING:root:And this, too
ERROR:root:And non-ASCII stuff, too, like Øresund and Malmö
```

root is the logger name

Python Debugger (pdb)

pdb is an interactive debugging environment that provides a way to set breakpoints, step through code, inspect variables, and evaluate expressions at runtime. There are two ways to enter the debugger:

Using the breakpoint() function
(Python 3.7+)

```
1 n = 5
2 sum_ = 0
3 for i in range(1, n):
4     breakpoint()
5     sum_ += i
```

Using the set_trace() function
while importing pdb directly

```
1 import pdb
2 n = 5
3 sum_ = 0
4 for i in range(1, n):
5     pdb.set_trace()
6     sum_ += i
```

In both cases, code execution will pause showing a (Pdb) prompt for **interactive** commands.

```
> …\script.py (3) <module>()
-> sum_ += i
(Pdb)
```

(3) refers to the current line which the debugger is looking at.

<module>() refers to the current level (module, function, class …).

Python Debugger - Interactive Commands

p	To print the value of an expression / variable (**p** [variable/expr]). **pp** pretty-prints the value of an expression.
n	To execute the current line and move to the next one in the current function.
s	To dive into a function call.
c	To continue execution until the next breakpoint.
l	To view the source code around the current line. **ll** for all the source code.
b	To list all breakpoints (**b**), set a breakpoint at a specific line number (**b** [line]) or function (**b** [function_name]).
w	To print a stack trace with the context of the current line **u** to move up in the stack. **d** to move down in the stack.
h	To display the list of available commands (**h**) or find out more about a specific command (**h** [command]).
q	To exit the debugger and terminate the program.

Two tools have been implemented to help with displaying variables :

locals() and **globals()** show all the variables in a dictionary on command – i.e. you need to type it.

On the contrary, **display** can be set up once for a variable to automatically keep track of it and see how its value changes over time.

Python pdb documentation

Exercises

Now, it's time to **practice**!

Complete the following **exercises**:

- MCQ : https://forms.gle/MrTSZ9ozUfycJsqU6
- 14 - (EN) Exercises - Errors & Exceptions