Module 02 - Programming basics and first steps in Python

Fundamentals of programming

To program, *whatever the language we use*, we *need* to be able to:

- Manipulate **simple values**: numbers, strings, dates, etc.
- Do some **primitive operations**: addition, multiplication, read data, write data, etc.
- Handle **collections** of things and manipulate **data structures**
- **Compare things** to **make decisions** (choose an instruction to execute in given circumstances)
- **To repeat** some instructions, until something happens or some conditions are met
- Write **functions** (procedures) to **organize**, **reuse** code and build *black-box* **abstractions**
- **Reason** about our programs (understand, read, modify)
- **Group** things into **modules** to **share** and reuse code
- Etc.

Those concepts are *fundamental* and *universal*!

Python in interactive mode (REPL)

We can use the Python interpreter in **interactive mode**, called a **REPL** (*Read Eval Print Loop*)

1. **Read**: type an expression
2. **Eval**: evaluate the expression (press Enter)
3. **Print**: see the result
4. Go back to 1.

In the terminal, **type** `python` to open the REPL or open the *Python Console* directly on PyCharm:

```
python
Python 3.11.2 (main, May 12 2026, 05:17:27) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2
2
>>> help
```

Use `Ctr+d` to **close** the REPL.

> The REPL is a powerful development tool to quickly test some code and ideas!

Advices on how to learn programming

- Follow this document, copy/paste examples, *test* your understanding
- *Program*! To learn programming, there is no *royal road*, you have to *write* and *run* programs! Start with basic instructions. Solve easy problems first
- *Browse* the official Python Documentation. This is the **ultimate reference** to check information and learn new things. Follow tutorials, check they syntax
- *Find resources* that *talk to you* (websites like W3School, video courses, etc.). Choose one good book and browse it (use the table of contents!)
- *Ask questions* to a LLM (Gemini, Claude, Chat GPT). Ex: *"What are lists in Python and what are the basics operations I can do with them. Give me a few examples and an exercise"*. Be sure to understand and to check given information
- Check the course progression to be sure to cover important topics
- Solve more interesting problems and complete challenges on web platforms: DataCamp, CodinGame, Code Wars

Information and values

- IT technologies and programs help us to solve problems and make *better decisions*
- How do we make *good* decisions? By *being informed*!
- What is information? Facts, *values*! (e.g. amount of money left, outside temperature, meeting date and time, etc.)
- We need to manipulate *values*!

Primitives in Python

Python provides ways to manipulate **values** of **different types**. We speak of **data types**.

You can inspect the type of a value using the `type(<value>) function`:

```
# Type in the Python Console:

42  # Integer
1.5  # Floating-point number (decimal)
-50.05

"a"  # String
"The quick brown fox jumps over the lazy dog"  # String
True #Boolean
False

# Inspect data types
type('a')  # str
type("The quick brown fox jumps over the lazy dog")  # str
type(1.5)  # float
type(42)  # int
type(True) # bool
```

Writing comments

- Comments start with `#`: a comment is text *ignored* by the interpreter
- Comments are **useful** and **important**, they allow you to explain the **why** behind the code, to **give some context** that can *not be expressed in the code*. Useful to take some notes!

```
#This is a one line comment. Comments are ignored by the interpreter!
x=2

#We can use 'one line comment' syntax
#to write comments on several lines.

42 # The rest of this line is a comment now

"""
This
is
a
block comment
spanning several lines
"""
```

> See the shortcut **to comment and uncomment multiple lines** of code easily in PyCharm (really useful!)

Primitive procedures: arithmetic operators

Python provides **primitive procedures** called **arithmetic operators** such as addition(+), subtraction(-), multiplication(*), division (/), integer division (//), modulo(%), power(**), etc.

```
# Type in the console
1.5 + 42
+42
42 * 42
5+-21
1 / 5
1 // 5
13 // 3
1e6 + 1
5E2
(1+2)*3
2**8 # power operator
(0+1j)**2 # Complex numbers
```

> Here, the addition operation is encoded by the `+` character. I don't need to know *how* the addition is performed (low level details). This primitive (*built-in*) procedure called `+` is *provided* by Python; I only need to know *what* it does (adds two numbers) and *how to use it*.

Expressions

An **expression** is a combination of *values*, *variables*, *constants*, *operators*, and *functions* that is **evaluated** to produce **a new value**.

For example, `a + b * 2 * 5` or `7 % (5 // 2)` are *expressions*.

Using variables

A **variable** allows you to:

- **give a meaningful name** to a special value;
- **store some data** during the program (like our piece of paper!);

A variable is created the moment you first **assign a value** to it, with the **assignment operator** `=`:

```
# 5 is assigned to the variable 'x'
x = 5
player1 = "John"

# Be careful, variable names are case-sensitive!
Player1 = "Jane"
print(player1, Player1)
```

> A variable is like a *bag* with a label on it to store and retrieve values

Reassign a value to a variable

You can **reassign a value to an existing variable**, of a completely different type, at any point in the program :

```
x = 5
x = 12.5
x = "some text"
x = [1, 2, 3] # a list of values
```

> This is similar to "updating" our piece of paper (erase and rewrite a new value)

Constants

Sometimes, variables must *not* change value during the execution of the program, like pi. We need "constant variables".

For *Constant variables*, we rely only on **naming convention**: use uppercase letters with underscores separating words (`ALL_CAPS_WITH_UNDERSCORES`),

```
# We declare constants with uppercase letters
PI = 3.14
MAX_TEMP_CELSIUS=100

radius = 2
circle_area = PI * radius ** 2

70 > MAX_TEMP_CELSIUS
```

> Python has no language-level `const` keyword (unlike C). There are ways to enforce constant values at *runtime* or during static analysis in Python, but they require more advanced concepts that we will explore later.

Evaluating a variable

A variable name is an *expression* and can be **evaluated**:

```
>>> a=42
>>> a
42
>>> a + 5 # a is evaluated to 42, 42 + 5 is then evaluated
47
```

> In the REPL, you can evaluate a variable by typing its name directly without using the `print()` function

Thought on assignment

Assignment operators (`==` , `+=` , `-=` , etc.) are **special** because **they perform an action**: they **change the memory**.

In most programming languages (including Python), the assignment operator (`=`) **does not have the same meaning as in mathematics**.

When we write `x = 2` , we say that we *assign* the value 2 to `x` . In mathematics, `x = 2` means that `x` is *identical* to 2.

In programming, you will often encounter statements like `x = x + 1` , which do not exist and have no meaning in mathematics! It means that we assign a *new* value to `x` equal to its previous value (`x` is evaluated to one) plus one.

```
x = 1       # x <---assign-- 1
x = x + 1   # x <---assign-- eval(x) + 1
print(x)
```

Assignments

```
x = 1
y = x # x is evaluated to 1, 1 is assigned to y
print(x, y)
x = 2
print(x, y)
x += 2 # short to x = x + 2, x stores the value 4
y *= x # short to y = y * x
print(y)
```

Rules to follow for Python variable names

Variable name:

- **must start with a letter or the underscore character**
- **cannot start with a number**
- can only contain **alpha-numeric characters** and **underscores** (`A-z` , `0-9` , and `_`)
- **is case-sensitive** (`age` , `Age` and `AGE` are three different variables!)
- cannot be any of the Python keywords (`if` , `not` , `None` , etc.)

Examples of legal variables names

```
myvar = "Jane"
my_var = "Jane"
_my_var = "Jane"
myVar = "Jane"
MYVAR = "Jane"
myvar4 = "Jane"
```

Examples of illegal variables names

```
1myvar = "Jane"
my-var = "Jane"
my var = "Jane"
myv$r = "Jane"
```

Multi Words Variable Names

Variable names with more than one word can be difficult to read. There are several techniques you can use to make them more readable (camelCase, PascalCase, kebab-case, snake_case, etc.).

We will use **snake_case** format (see naming conventions section):

```
my_variable_name = "John"
```

Variables is our first mean of *abstraction*

To write useful, readable, and maintainable programs, it is essential to have useful **abstractions**: an *abstraction* allows you to *strip away unnecessary and cumbersome details* and *name* the entities you manipulate.

Variables are a necessary *mean of abstraction* at our disposal.

> Natural language is itself a form of *abstraction*: naming things enables us to discuss and reason about them, without having to specify *every single detail*!

Why data types?

Values (and thus variables holding values) has **types**.

Why do we need a type system?

```
42 + 'a' # Can I add a string and an integer?
12 * 'a' # Can I multiply strings?
'aaa' / 3 # What does it mean to divide a string? Is it meaningful?
```

For every expression, what *should* happen? It depends!

Data types

The **type** of an *object* (i.e, *a piece of data* manipulated in the program) determines:

- The **values** it can represent;
- The **operations it can undergo**, **which operators can interact with an object and how they behave**. The same operator (like + or *) can perform completely different operations depending on the types involved!

> In lower-level languages like C, types are necessary to define how much memory to allocate and how to interpret the binary data

Data types in Python

In Python, the *built-in* data types fall into these primary families:

- **Integers** (`int`) and **Booleans** (`bool` , which are a subtype of `int`)
- **Floating-point numbers** (`float`) and **Complex numbers** (`complex`)
- **Strings** (`str`)

```
# int
12
16//3

# float
0.52
1/3

# complex
1+5j
(1+1j)*(1-1j)

# str
"hello, world"
'some text'

# bool
```

Arithmetic operators: integer division and modulo

```
1/2     # 0.5
1//2    # 0
1 % 2   # 1
4 % 2   # 0
```

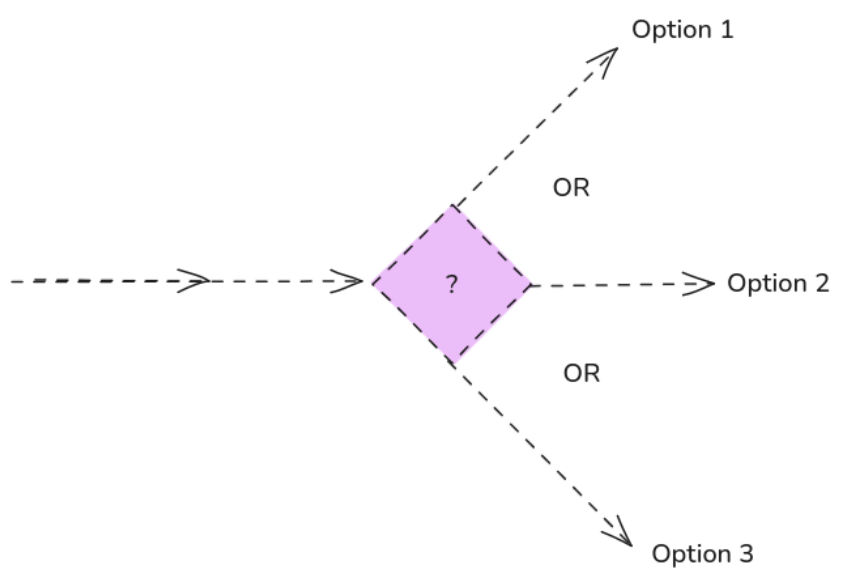
- Integer division (`//`): Keeps only the integer part of the division. The result is an *integer*. (`2//3` is asking *how many times there is 3 in 2? 0*)
- Modulo operator (`%`): The *modulus* (i.e. the result of the modulo operation) is the *remainder* of integer division ($1 \% 2 = 0 * 2 + 1$ = 1)

> Modulo allows to define *even* or *odd* numbers: the integer x is an even number if it can be divided by 2 with no remainder, i.e, x is equal to 0. By definition, 0 is an even number

Compare things

- How do you *make a decision*? By *choosing* between *different* options
- "*Is something* bigger/less expansive/more expansive/better/sooner/later, etc. *than another thing* ?"
- Answers to these questions are always either *Yes* or *No*, *True* or *False*: they are *Boolean* values
- To make *informed decision*, we need to *compare things*, we need to compare *values*!

> You spend your time comparing things in life!



Comparison operators: ==, !=, >, <, >=, <=

```
1 == 1    # is equal to
2 > 3     # is greater than
1 + 5 < 9 # is lower than
2 != 3    # is not equal to, is different from
100 + 100 >= 200 # is greater or equal to
True == True
1 == True
2 == True
0 == False
8 % 2 == 0 # this is the definition of even number
9 % 2 == 1
```

> Expressions evaluated to `True` or `False` are special expressions, also called **predicates**.

Comparison example

```
price = 1000
my_budget = 1200

# Can I buy it?
is_affordable = my_budget >= price

print(is_affordable)
```

Statement

A **statement** (instruction) in Python is a **line** that executes an action or specifies an operation to perform:

```
price = 1000 # statement: assign 1000 to price
my_budget = 1200 # statement: assign 1200 to my_budget
is_affordable = my_budget >= price # statement: comparison and assignment

print(is_affordable) # statement: print on the output
```

Execution flow

A program executes line by line, **statement after statement**.

The *order* of the statements *encodes time*: it's called the **execution flow** (or flow of execution) of the program.

```
price = 1000                          # 1 |
my_budget = 1200                      # 2 |
is_affordable = my_budget >= price    # 3 |
                                      #   |
print(is_affordable)                  # 4 v time

# ...
```

Making the choice: if statement

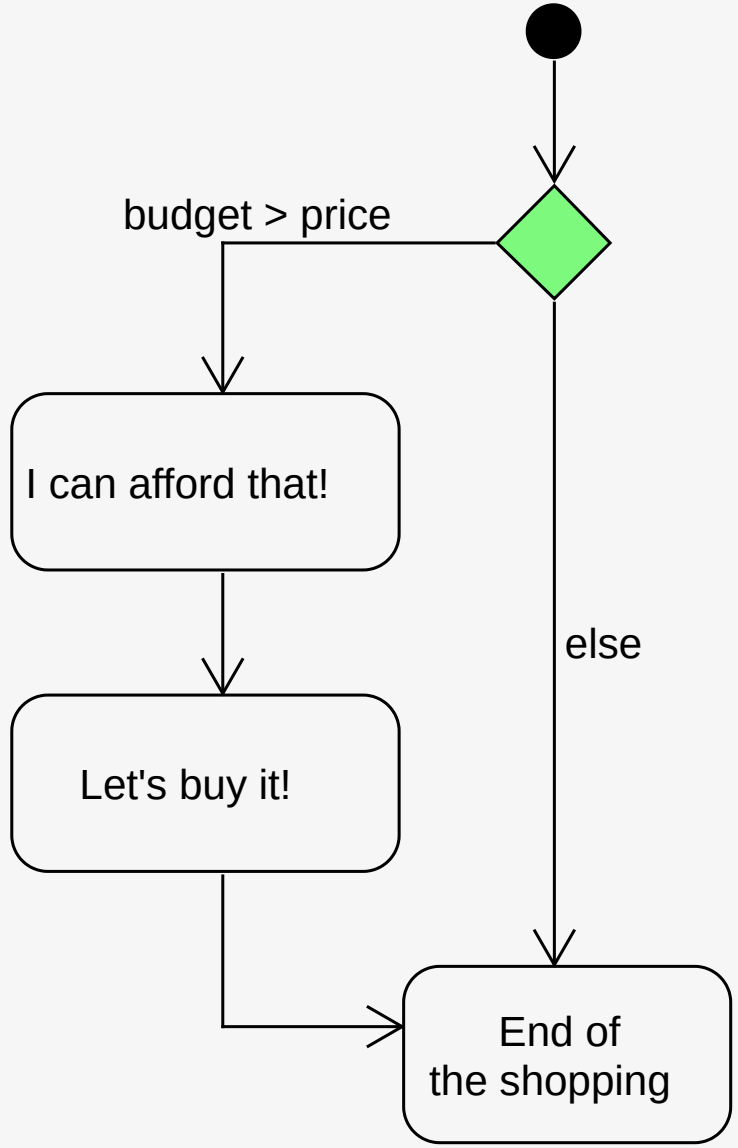
```
price = 1000
my_budget = 1200

# Compare
is_affordable = my_budget >= price

# Decision (branching)
if(is_affordable):
  # Executed only if is_affordable is 'True'
  print("I can afford that!")
  print("Let's buy it!")

print("End of the shopping")
```

Change `my_budget` to `500`. What happens? Why?



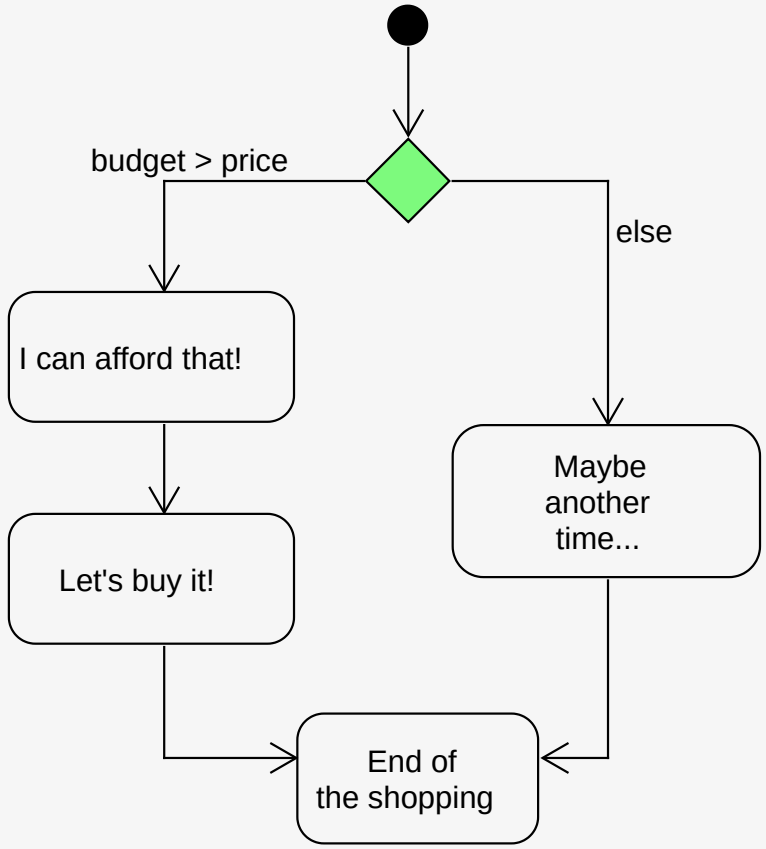
Making the choice: else/if statement

```
price = 1000
my_budget = 500

# Compare
is_affordable = my_budget >= price

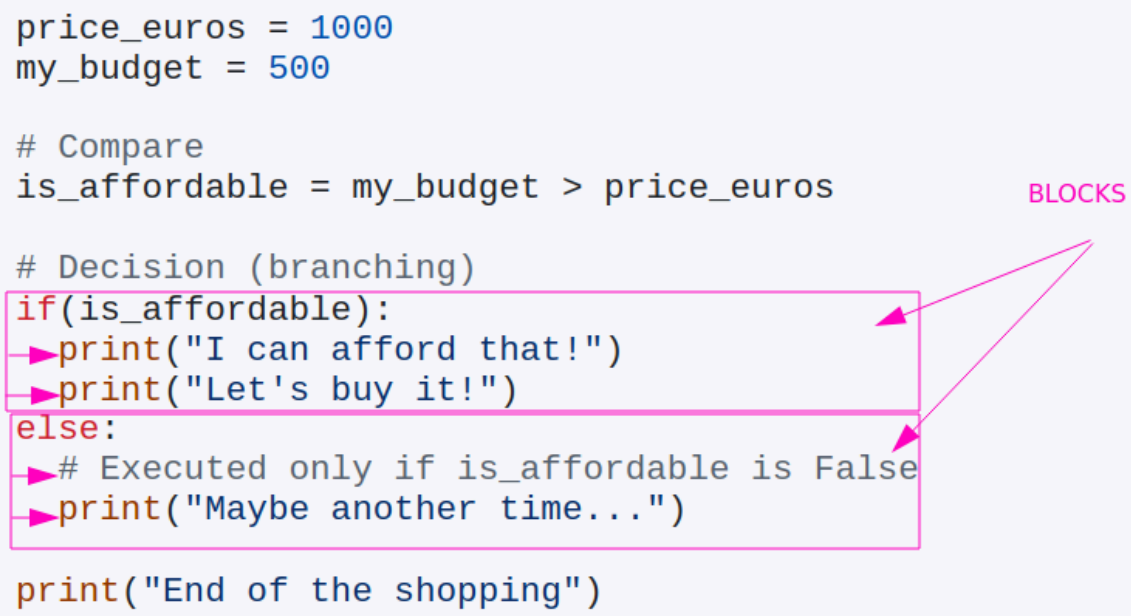
# Decision (branching)
if(is_affordable):
  print("I can afford that!")
  print("Let's buy it!")
else:
  # Executed only if is_affordable is False
  print("Maybe another time...")

print("End of the shopping")
```



Code blocks

- A *code block* is a *group* of programming statements that are executed *as a single unit*.
- Python use **indentation level** to group statements. All consecutive lines indented *at the same depth* belong to *the same block*.



Dynamic data types

We have seen previously that **Python is a dynamically typed language**.

That means Python **infers the type** of objects **based on the value** assigned to them **at runtime**.

```
#my_variable stores an int
my_variable = 5
my_variable = "my_variable stores now some random text"
#my_variable stores a float
my_variable = 1/10
```

Casting

You can **cast** (*convert*) the data type of a value *from one type to another* using *built-in* conversion functions: `int()`, `str()`, `float()`

```
# Python will NOT automatically convert '5' to an int or 10 to a string.
result = "5" + 10  # Raises TypeError: can only concatenate str (not "int") to str

# You must perform explicit type conversion or casting with built-in functions:
result = int("5") + 10  # 15
result = "5" + str(10)  # "510"
```

> Being strongly typed is helpful to reduce the risk of undesired behaviors and hard to catch bugs!

Getting input from the user

Use the `input(prompt)` function to ask the user for input:

```
# Prompt and store the user input in the variable 'answer'
answer = input("What is your budget?")
print(answer)
```

What is the type of `answer`? How to *cast* it to integer?

A few things about strings

String is a common and useful data type. Let's review a few properties of strings.

Single vs Double quotes

String variables can be declared *either* by using *single* or *double* quotes:

```
x = "John"
# is the same as
x = 'John'

# Double quotes allow single quotes inside, and single quotes allow double quotes inside.
x = "He is called 'Johnny'"
x = 'He is called "Johnny"'
```

Escape sequences

Escape sequences in string literals are **interpreted** according to rules similar to those used by Standard C (and typewriters!)

```
#\n for newline feed (useful!)
print("A\nB\nC")
#\t for horizontal tabulation
print("A\tB\tC")
#\r carriage return
print("AB\rC")
```

> See all escape sequences available

Multiline strings

```
multiline_str="""This is a
multiline
string"""

another_multiline_str='''this is
also
a multiline
string'''
print(multiline_str)
```

> Wait...But this is a multiline comment? Python will ignore string literals that are *not* assigned to a variable, that's why you can add a multiline string (triple quotes) in your code, and place your comment inside it

Concatenation

```
a = "And now, "
b = "for something completely "
msg = a + " "+ b + "different"
print(msg)
```

Formatted strings (*f-string*)

F-String was introduced in Python 3.6, and is now the preferred way of formatting strings.

To specify a string as an *f-string*, simply put an `f` in front of the string literal (as a prefix), and **add curly brackets** `{}` as **placeholders** for variables and other operations:

```
char="A"
formatted_string=f"Your are looking for the letter {char}"
print(formatted_string)

a=2
b=3
print(f"a+b={a+b}")

price = 42
txt = f"The price is {price:.2f} dollars" #format the price with 2 decimal digits
print(txt)
```

> There are several prefixes available to influence the content of the string

Escaping special characters

The escape character (`\`) allows you to use **"escape" special characters in strings** (suppressing their standard interpretation):

```
quote = "He said, \"Python is great!\""
print(quote)

txt = "We are the so-called \"Vikings\" from the north. Isn\'t it amazing?"
print(txt)

path="\\Some\\Path\\On\\Windows"
print(path)
```

> Note: in the REPL, evaluating a variable show you the complete structure of the variable (debug), similar to use the function repr(). It does not give you the same result as using the print() function

`None`, the absence of value

`None` is a built-in literal representing the **intentional absence of a value**.

```
data = None
print(data)
print(type(data))
```

Useful to indicate missing data, **uninitialized state**, or search failure.

Operators

Operators specify **what can be done** with variables and values.

We already have seen a few of them (`=` , `+` , `*` , `==` , `!=` , `>` , etc.)

Different kinds of operators

Python features a **wide variety of operators**. They can be **categorized** by their nature:

- **Arithmetic**: +, -, *, /, //, %, **
- **Relational** (Comparison): >, <, >=, <=, ==, !=
- **Logical**: `and`, `or`, `not`
- **Assignment**: =, +=, -=, *=, /=, :=
- **Identity and Membership**, to check object identity or containment: `is`, `is not`, `in`, `not in`
- **Bitwise**, to manipulate individual bits directly: &, |, ^, ~, <<, >>

Assignment operators and *lvalue*

Because the assignment operator is special (*it modifies memory*), its **left operand is subject to certain restrictions**. We use the term *lvalue* ("left" value) to designate this special operand.

- Left-hand side: **Must be a valid identifier**: **variable name**, attribute (`obj.attr`), item lookup (`x[0]`)
- Right-hand side: Can be **any expression** that evaluates to an object.

```
n = 5          # n IS a lvalue
n = b + 3;     # n IS a lvalue
5 = x          # 5 IS NOT a lvalue. SyntaxError: cannot assign to literal here
n + 1 = b + 3 # n + 1 IS NOT a lvalue
```

Logical operators: Truth tables

AND

`and`	**True**	**False**
True	True	False
False	False	False

OR

`or`	**True**	**False**
True	True	True
False	True	False

NOT

`not`	
True	False
False	True

Practice

```
# Gives the value of each expression

True and True
True and False
False or False
False or True
False or False
True or (not False)
True and (True and not True)
```

Operator precedence (priority) and associativity

- Not all operators have the same priority or **precedence**
- When two operators share the same precedence, their **associativity** (*left-to-right* or *right-to-left*) determines the evaluation order.

```
# What is the value of a?
a = 5 / 2 + 3 * 3;

# What are the value of a and b?
b = a = 5 * 2;

# What is the value of c?
c = True or False and False

# What is the value of d?
d = not 1 == 2

# What is the value of e?
e = (not 1) == 2
```

Precedence table

How to read and use precedence table:

- Rows are ordered by *decreasing* precedence (from highest to lowest priority);
- Operators on the **same row** have the **same precedence**;
- If operators share the same precedence, their *associativity* determines how they are grouped (either *left-to-right* or *right-to-left*);

> See the official documentation

Operators	Description	Associativity
`()`	Parentheses (grouping)	N/A
`x[i]`, `x[i:j]`, `f(...)`, `x.attr`	Subscription, slicing, function call, attribute reference	Left-to-right
`**`	Exponentiation	**Right-to-left**
`+x`, `-x`, `~x`	Positive, negative, bitwise NOT	**Right-to-left**
`*`, `/`, `//`, `%`	Multiplication, division, floor division, remainder	Left-to-right
`+`, `-`	Addition, subtraction	Left-to-right
`<<`, `>>`	Bitwise shifts	Left-to-right
`&`	Bitwise AND	Left-to-right
`^`	Bitwise XOR	Left-to-right
`|`	`|`	Bitwise OR
`<`, `<=`, `>`, `>=`, `==`, `!=`, `is`, `is not`, `in`, `not in`	Comparisons, identity, membership	**Chained**
`not`	Logical NOT	**Right-to-left**
`and`	Logical AND	Left-to-right
`or`	Logical OR	Left-to-right
`if`, `else`	Conditional expression (ternary)	Left-to-right
`:=`	Assignment expression (walrus operator)	**Right-to-left**
`=`, `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, etc.	Assignment operators	**Right-to-left**

Precedence table, to remember

```
Priority
   ^
   | Arithmetic
   | Relational(Comparison)
   | Logical
   | Assignment
```

> True for (almost) all programming languages !

Use parenthesis

- Parentheses allow you to **control evaluation order** and **make the code more explicit and readable**
- **Use parenthesis!**, when needed.

Practice: basic types, variables, operators, basic else/if statement

Solve:

- Problem 1
- Problem 3

from the Problem Sheet

Means of combination : collections of things

Python offers 4 different *built-in* data types to store **collection or combination of data**, called **data structures**:

- List
- Tuple
- Dictionary
- Set

Each *data structure* has its own properties and use cases. Let see some **basic properties** of each one.

> Lean more about Python built-in data structures

Lists

Values can be organized into *collections* called **lists** in Python.

A list of values is declared using *square brackets* (`[]`), with each element separated by a **comma**(`,`):

```
#A 'list'
numbers = [1, 2, 3, 4, 5]

#A 'list' can contain values of different types (heterogenous)
items = [-1, 42, "Jane Doe", ['A', 'B', 'C'], 12 / 5 ]
```

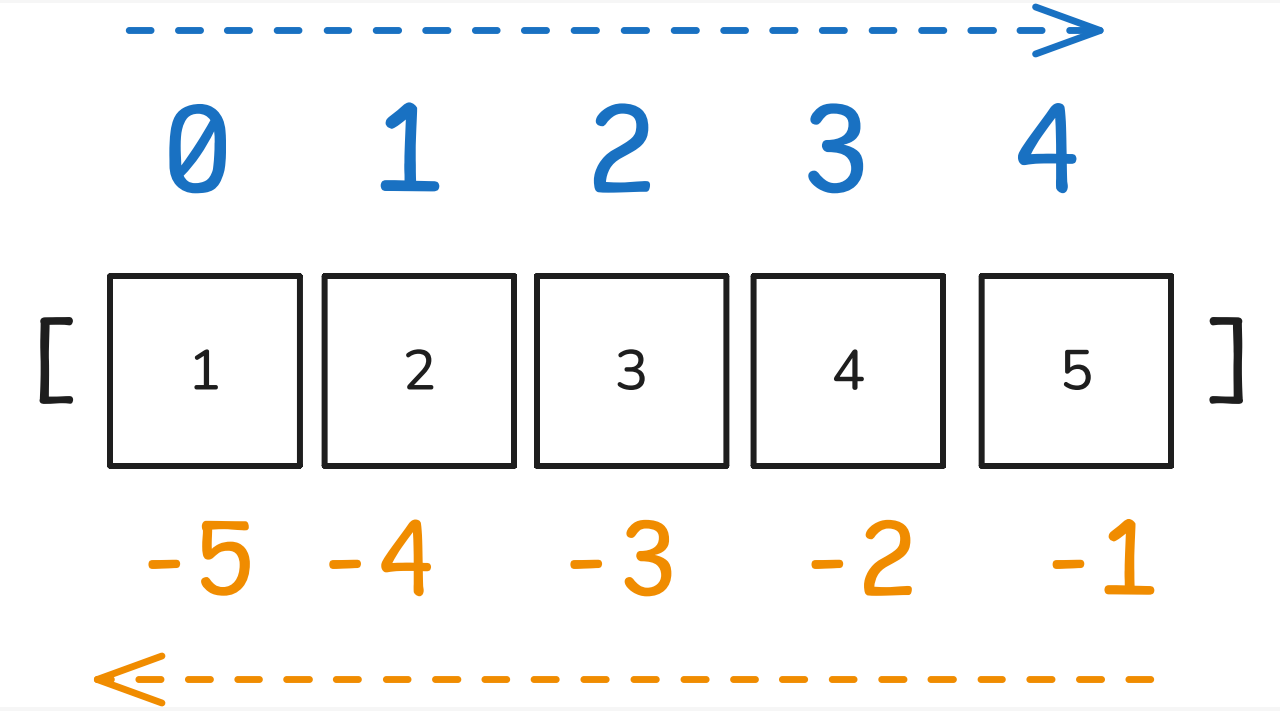
Accessing elements by index

- A list is an **ordered** collection of values.
- Each element has a **position** (an integer) called **index**.
- List indexing **always starts at 0**!

```
#index:   0       1       2
names = ["john", "jane", "charlie"]
```

Indexes

- Indexing (lists, strings) always **start at 0**
- A **negative index** accesses elements by counting **backward from the end**, rather than forward from the beginning.
- -1 is the last item, -2 the second last item, etc.



Accessing elements by index, an example

You can access any element by specifying its position using square brackets:

```
items = [-1, "Jane Doe", ['A', 'B', 'C'], 12/5]

# Each item is indexed and accessible by its index (from 0 to length-1)
print(items[0])
print(items[1])
print(items[2])
print(items[3])
print(items[4])  # What happens ? IndexError: list index out of range

print(items[-1]) # -1 refers to the last item
print(items[-2]) # -2 refers to the second last item
```

Counting elements in a list

You can count *how many* elements are in a list with the *built-in* `len()` function:

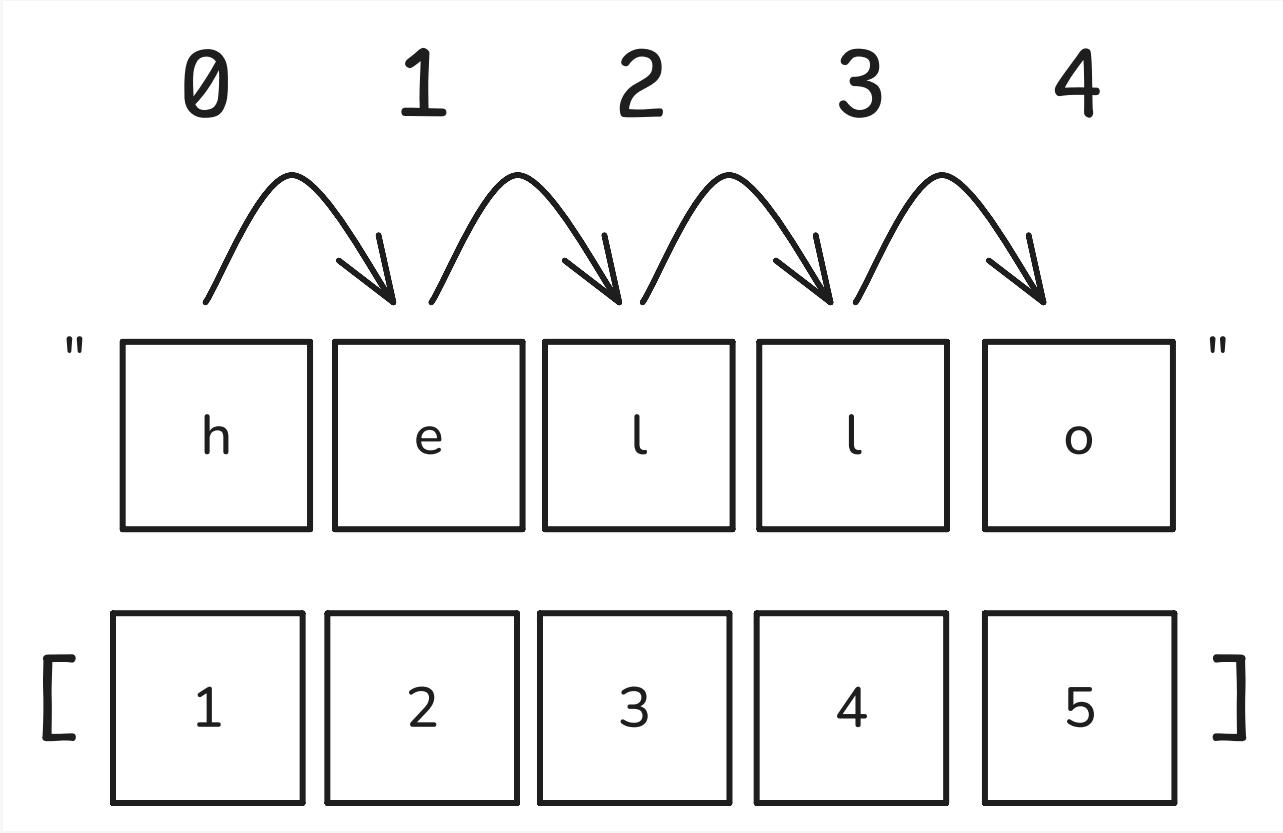
```
items = [-1, "Jane Doe", ['A', 'B', 'C'], 12/5]
nb_items = len(items)
print(f"There are {nb_items} in the collection")

msg="Does it work on a string?"
print(len(msg))
```

> strings share properties with lists, they are **iterable** data structures!

Iterable data structures

An *iterable* data structure (or simply an iterable) is any data collection that allows you to step through its items one by one, typically inside a *for loop*.



Iterate over a list using a for loop

A *for loop* allows you to go through a list and process *each item*, one by one:

```
numbers = [0, 1, 2, 3, 4]
# In each pass, the variable 'n' holds the current number from the list
for n in numbers:
  print(n)

# Output:
# 0
# 1
# 2
# 3
# 4
```

Iterate over a string

Because strings are *iterable*, we can do the same thing with strings and visit each character, one by one:

```
message = "hello, world!"

for char in message:
  print(char)

# Output:
# h
# e
# l
# l
# o
# ,
#
# w
# o
# r
# l
# d
# !
```

Lists are mutable

- **Lists are mutable**: you can modify items in place
- A list item is a *lvalue*, like a variable !

```
colors = ['orange', 'blue', 'green', 'orange']
# Change the first item
colors[0] = 'red'
# Change the second item
colors[1] = 'yellow'
print(colors)

colors[10] = 'black' # Will this work ?
```

Strings are immutable

Strings share many properties with lists: they are a **collection** of unicode characters, they are **iterable** but, *unlike lists*, strings are **immutable**! (you can not change the content of a string!)

```
print(len('été')) # 3

name="John"
print(name[0]) # J
print(name[1]) # o

name[0]="j" #TypeError: 'str' object does not support item assignment
```

Adding an element to a list with string methods

- You can add elements to a list (lists are *mutable*!) by using special *built-in* functions called **methods**
- Unlike standard functions like `print()`, a *method* **belongs to a specific data type** and is called directly on the variable using the dot operator (`.`):

```
colors = ['orange', 'blue', 'green', 'orange']

# Lists offer a method 'append(value)' to add an item at the end
colors.append('yellow')

print(colors)
```

> Each datatype offer several methods to perform some common and useful operations on the data!
> Check the string methods

Adding elements with insert and extend

- `lst.insert(index, value)` add an element at a desired index (position)
- `lst.extend(list)` append elements from another list to the current list

```
# Insert an element at some desired index/position:
colors.insert(1, 'red')

# Append elements from another list to the current list
my_favorite_colors = ['cyan', 'gray']
colors.extend(my_favorite_colors)

print(colors)
```

Removing an element from a list

`lst.remove(value)` removes *the first item* from the list whose value is equal to `value`.

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

Reverse elements in the list

`lst.reverse()` reverse the elements of the list in place.

```
numbers = [1, 2, 3, 4]
numbers.reverse()
print(numbers)
```

> Lean more on methods available for lists by scheming the documentation. When programming, you spend much more time reading (documentation, code) than writing

Practice: List methods

Consider this list:

```
names = ["Lamport", "Von Neumann", "Abelson", "Van Rossum", "Ritchie", "Thompson"]
```

1. **Write** a *for loop* to print each name on a new line
2. Browse the documentation on list and **find methods** to:
    i. Find the index of "Van Rossum"
    ii. Remove and return the last item from the list and store it in a variable `last_item`
    iii. Sort names alphabetically
    iv. Remove all items from a list

> This is your first visit to the official documentation! It is the *ultimate source of truth* to learn and check something.

Correction

```
names = ["Lamport", "Von Neumann", "Abelson", "Van Rossum", "Ritchie", "Thompson"]

# 1.
for name in names:
    print(name)

# 2. Discover and test list methods
idx = names.index("Van Rossum")   # 1. Find index
last_item = names.pop()           # 2. Remove last item
names.sort()                      # 3. Sort items
names.clear()                     # 4. Remove all items
```

Practice : List of numbers

1. **Make a list** of the numbers from one to one million (excluded), and then use `min()` and `max()` to make sure your list actually starts at one and ends at one million. Also, use the `sum()` function to see how quickly Python can add a million numbers.
2. Use the third argument of the `range()` function to make a list of the **odd numbers**, from 1 to 20 (excluded). Use a *for loop* to **print each number**.
3. **Make a list** of the multiples of 3 from 3 to 30 (included). Use a *for loop* to **print** the numbers in your list.

> From Prof. Karim ZKIK, PhD Materials

Correction : List of numbers

```
# 1.
numbers = list(range(1, int(1E6)))

print(min(numbers))
print(max(numbers))
print(sum(numbers))

# 2.
odds = list(range(1, 20, 2))
for odd in odds:
 print(odd)

# 3.
threes = list(range(3, 31, 3))
for t in threes:
 print(t)
```

Build new lists with *Slicing*

You can create a new list by slicing an existing one. To *slice* a list, specify a **start index** (included) and an **end index** (excluded).

The result will be a **new list with the specified items**.

```
# Syntax
list_name[start: end: step]
```

> [start:end): Python follows the "closed open interval" principle. If omitted, `start` defaults to 0, `end` defaults to the length of the list (one past the last item)

> Slicing works also on strings!

Slicing, a few examples

```
colors = ['red', 'blue', 'green', 'yellow']

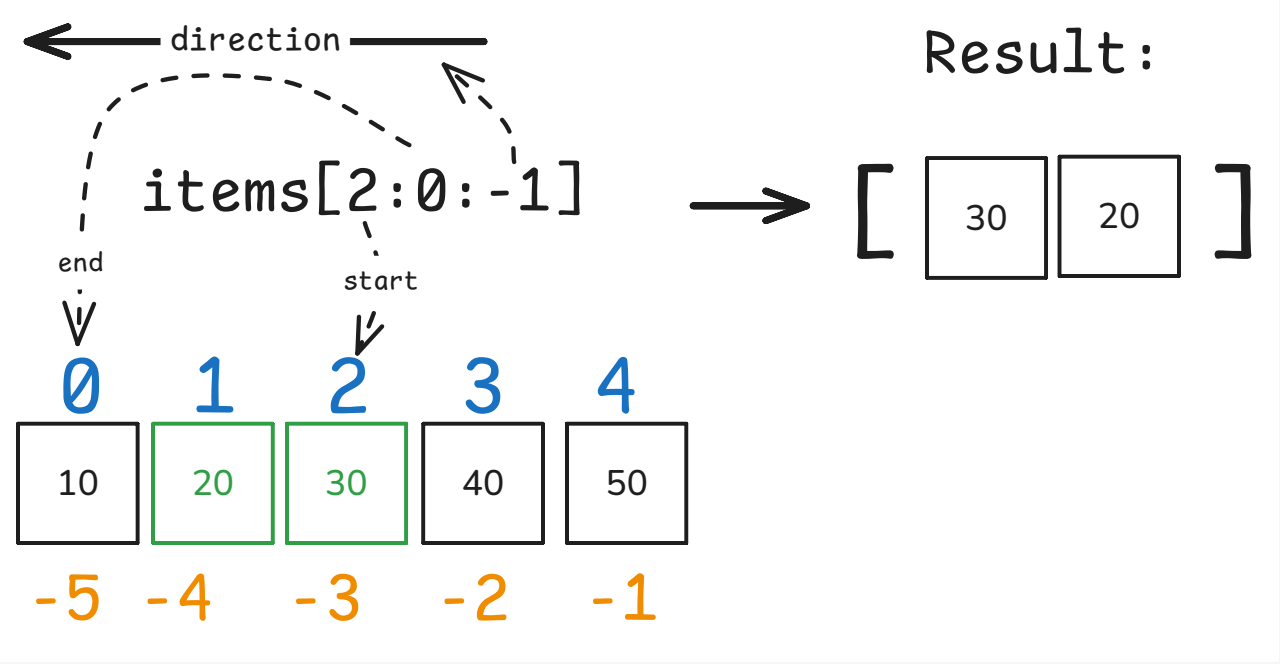
# Slice 'all' (start and end omitted)
print(colors[:])        #  ['red', 'blue', 'green', 'yellow']
print(colors[1:])       #  ['blue', 'green', 'yellow']
print(colors[:2])       #  ['red', 'blue']
print(colors[1:3])      #  ['blue', 'green']
print(colors[-2:])      #  ['blue', 'green', 'yellow']
print(colors[-3:-1])    #  ['blue', 'green']
# Reverse the list (negative step)
print(colors[::-1])     # ['yellow', 'green', 'blue', 'red']
```

> Positive step (1, 2, etc.) steps *forward* (left to right). Negative step (-1, -2, etc.) steps *backward* (right to left)

Slicing with a negative step

When slicing with a *negative step*, the **reading direction reverses** (from *right to left*)

> Which result gives the following slice:
> `items[-3:-5:-1]` ?



Slicing with a negative step, examples

```
items =  [0,  10, 20, 30, 40, 50]
# index : 0   1   2   3   4   5

print(items[::-1])   # [50, 40, 30, 20, 10, 0], reversed list!
print(items[:2:-1])  # [50, 40, 30]
print(items[4:2:-1]) # [40, 30]
print(items[1::-1])  # [10, 0]
print(items[3:1:-1]) # [30, 20]
```

> Note: You can't get *circular wrapping* with slicing only, e.g `[10, 0, 50, 40]` from `items`

Practice: slicing lists 1/2

Consider the following list representing temperature readings (in degrees Celsius) throughout a week:

```
temperatures = [18, 21, 19, 24, 22, 20, 25]
# Index:          0   1   2   3   4   5   6
# Days:         Mon Tue Wed Thu Fri Sat Sun
```

Use list slicing, to create the following lists:

1. *Weekdays*: Extract the readings from *Monday through Friday* (`[18, 21, 19, 24, 22]`)
2. *Weekend*: Extract the readings for *Saturday* and *Sunday* (`[20, 25]`).
3. *Mid-week*: Extract *Wednesday* and *Thursday* (`[19, 24]`).

Practice: slicing lists 2/2

Consider this list representing daily sales amounts (in dollars) over a 10-day period:

```
sales = [120, 85, 200, 150, 90, 300, 400, 110, 95, 250]
# Days:   1   2    3    4   5    6    7    8   9   10
```

1. **Slice** the list to extract the sales for the first 5 days.
2. **Write** a *for loop* over this slice to **calculate** and **print** the **total sales** for the first 5 days.
3. **Slice** the list to retrieve days 6 through 8 (`[300, 400, 110]`).
4. **Loop** through this slice and **print** only the *sales amounts that are greater than $150*.
5. **Compute** and **print** the *average sales amount* over the 10-day period.
6. **Create** a *reversed slice* of the last 4 days (`[250, 95, 110, 400]`).

Correction: slicing lists 1/2

```
temperatures = [18, 21, 19, 24, 22, 20, 25]

weekdays = temperatures[0:5]       # Output: [18, 21, 19, 24, 22]
weekend  = temperatures[5:7]       # Output: [20, 25]
mid_week = temperatures[2:4]       # Output: [19, 24]
```

Suggestion: slicing lists 2/2

```
sales = [120, 85, 200, 150, 90, 300, 400, 110, 95, 250]

# 1. Total sales for the first 5 days
first_half = sales[:5]

# 2.
total_first_half = 0

for amount in first_half:
    total_first_half += amount

print(f"Total sales (Days 1-5): ${total_first_half}")

# 3. High sales in days 6 to 8
peak_days = sales[5:8]

# 4.
for amount in peak_days:
    if amount > 150:
        print(f"High sales amount: ${amount}")
# Output:
# High sales amount: $300
# High sales amount: $400

# 5. Average amount sales
average_sales=0
for amount in first_half:
    average_sales += amount

average_sales /= len(sales)
print(f"Average sales: ${average_sales:.2f}")

#6. Reversed list of the last 4 days
print(sales[9:5:-1])
# OR
print(sales[:5:-1]) #we start at the end, until 5th position
# OR
print(sales[-1:-5:-1])
```

List comprehension

List comprehension offers a shorter syntax **when you want to create a new list based on the values of an existing list**:

- **filter** (keep only elements satisfying a predicate)
- **map** (transform every element to another one)

Syntaxe:

```
new_list = [function(item) for item in list if condition(item)]
```

Filtering a list

```
a = [1,4,2,7,1,9,0,3,4,6,6,6,8,3]
# We would like to filter values greater than 5
b = []
for x in a:
    if x > 5:
        b.append(x)
print(b)
```

Filtering a list with list comprehension

Same operation with *list comprehension syntax*

```
a = [1,4,2,7,1,9,0,3,4,6,6,6,8,3]
b = [x for x in a if x > 5]
print(b)
```

Mapping items of a list

```
a = [1,4,2,7,1,9,0,3,4,6,6,6,8,3]
# We would like to square each item in the lsit
b = []
for x in a:
    b.append(x**2)
print(b)
```

Mapping items of a list with list comprehension

```
a = [1,4,2,7,1,9,0,3,4,6,6,6,8,3]
b= [x**2 for x in a]
print(b)
```

Other examples

```
fruits = ["apple ", " banana", " cherry", "kiwi ", "  mango      "]

# Transform every item in uppercase
newlist = [x.upper() for x in fruits]

# Trim a string (remove space at the begining and at the end)
# Chaining method calls
newlist = [x.strip().upper() for x in fruits]

# Replace every item by 'hi'
newlist = ['hi' for _ in fruits]
```

Creating lists

```
import random
random_numbers = [ random.randrange(1,10) for _ in range(100) ]
print(random_numbers)
```

Practice

Given the following list `a = [1,4,2,7,1,9,0,3,4,6,6,6,8,3]`, use comprehensive list instruction to:

1. add 1 to each element
2. filter only odd values
3. filter even values and cube them

Correction

```
#1. add 1 to each element
[x+1 for x in a]
#2. filter only odd values
[x for x in a if x %2 != 0]
#3. filter even values and cube them
[x**3 for x in a if x % 2 == 0]
```

Tuples

A tuple is an **immutable** collection. Once defined, you can not change a tuple. It is defined with parenthesis:

```
# A tuple
vowels = ('a','e','i','o','u','y')

# Alternative syntax
some_tuple = "apple", "banana", "cherry"

# Tuple are ordered collections like lists, each element can be accessed via its index
print(some_tuple, some_tuple[0], some_tuple[1])
```

Tuple unpacking

Tuple unpacking allows you to extract elements from a tuple directly into individual variables **in a single line**.

```
# We want to extract data in x and y variables
point = (10, 20)

# With manual indexing
x = point[0]
y = point[1]

# Withe tuple unpacking
x, y = point # or (x, y) = point
print(x, y)
```

Practice: tuples

Given the following two coordinate pairs represented as tuples:

```
# Format: (latitude, longitude)
paris = (48.8566, 2.3522)
lyon = (45.7640, 4.8357)
```

1. **Extract** the latitude and longitude of Paris into two separate variables using indexing (`[]`), then **print** them.
2. **Extract** the latitude and longitude of Lyon into two variables in a single line using *tuple unpacking*.
3. **Try to update** Paris's latitude to `48.9000`. **Observe** what happens when you run the code.

Correction: tuples

```
paris = (48.8566, 2.3522)
lyon = (45.7640, 4.8357)

# 1. Accessing elements by index
lat_paris = paris[0]
lon_paris = paris[1]

print(f"Paris -> Latitude: {lat_paris}, Longitude: {lon_paris}")

# 2. Tuple Unpacking
lat_lyon, lon_lyon = lyon

print(f"Lyon -> Latitude: {lat_lyon}, Longitude: {lon_lyon}")

# 3. Immutability Check
# Try attempting to reassign a tuple element:
# paris[0] = 48.9000
# Raises TypeError: 'tuple' object does not support item assignment
```

Dictionaries

Dictionaries are used to store data values in **key:value pairs**.

Dictionaries are among Python's **most common and versatile data structures**:

- A dictionary acts as a *map* (a *hash map*). Like a mathematical *function*, it associates each unique key with a specific value. It's the simplest function!
- It allows you to **model** real-world *entities* using *named fields*. For example, a person can have a name, email, and age.
- Retrieving a value by its key is *extremely fast*, it takes constant time regardless of the dictionary's size!

Create a dictionary

Dictionaries are written with curly brackets(`{}`), and have keys/values, separated by commas:

```
car = {
  #key      #value
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(car)
```

Dictionaries are **iterable**, **mutable** and **do not allow duplicates** (can't have two identical keys in a dictionary!).

Dictionary items are **ordered**.

On dictionary keys

Any data type that is **immutable** can be used as a dictionary key: strings, numbers, booleans, tuples

```
# Valid dictionary with mixed key types:
data = {
    "name": "Jane",         # String key
    101: "Admin User",      # Integer key
    (48.85, 2.35): "Paris" # Tuple key (coordinates)
}

print(data[101])           # Output: Admin User
print(data[(48.85, 2.35)]) # Output: Paris
```

> String keys are by far the most common

Accessing, Changing and Adding Values

```
car = {
  #key      #value
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

# Access a value by key
print(car['brand'])

# Change a value under a key
car['brand'] = 'Peugeot'

# Add a key/value
car['color'] = 'Black'
print(car)

# Add a key/value using the method update()
car.update({"color": "red"})
print(car)
```

Remove keys from a Dictionary

Removing a key from a dictionary removes the associated value.

```
car = {
  #key      #value
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

# Remove the 'model' key with the pop() method
car.pop('model')

print(car)
```

Loop through a Dictionary

Like lists or strings, you can loop through a dictionary by using a for loop: dictionaries are **iterable**!

When looping through a dictionary, the return value are the **keys** of the dictionary, but there are methods to return the values as well.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

for x in car:
  print(x) # will print brand, model and year

# Or with keys() method
for x in car.keys():
  print(x)
```

Loop through Dictionary Values

```
# With square bracket notation
for x in car:
  print(car[x])

# With values() method
for x in car.values():
  print(x)
```

Loop through both keys and values

```
#items() return a collection of each key/value represented as a tuple
for key, value in car.items():
  print(key, value)

# OR, with parenthesis (tuple notation)
for (key, value) in car.items():
  print(key, value)
```

Practice: Dictionary basics

1. **Create a dictionary** and stores in `customer_a` to *model* a customer. A customer is defined by four keys or *fields*:
    - `first_name` ("Jane")
    - `last_name` ("Doe")
    - `email` ("j.doe@email.com")
    - `total_purchases` : (150.00)
2. Jane made another purchase of $50.00. **Update** `total_purchases` to 200.00.
3. **Add** a new *key-value pair* "is_vip" with the boolean value `True`
4. **Print** the customer's full name using string formatting ("Jane Doe").
5. **Remove** the "email" key from `customer_a`

Correction: Dictionary basics

```
# 1. Creation
customer_a = {
    "first_name": "Jane",
    "last_name": "Doe",
    "email": "jane.doe@example.com",
    "total_purchases": 150.00
}
# 2. Updates
customer_a["total_purchases"] = 200.00  # Modify existing key

# 3. Add key/value pair
customer_a["is_vip"] = True             # Add new key

# 4. Access
full_name = f"{customer_a['first_name']} {customer_a['last_name']}"
print(f"Customer Name: {full_name}")

# 5. Remove
customer_a.pop("email")

print(customer_a)
```

Practice: Dictionary collection

Given the following list of daily temperature recordings (celsius):

```
temperatures = [
    {"date": "2026-03-01", "temperature": 12.5},
    {"date": "2026-03-02", "temperature": 14.0},
    {"date": "2026-03-03", "temperature": 9.8},
    {"date": "2026-03-04", "temperature": 20.2},
    {"date": "2026-03-05", "temperature": 15.1},
    {"date": "2026-03-06", "temperature": 11.4},
]
```

1. **Calculate** and **print** the average temperature across all recorded days.
2. **Find** and **print** the highest recorded temperature
3. **Identify** and **print** the date corresponding to that maximum temperature (*hottest day*)

Suggestion: Dictionary collection

```
from datetime import datetime

temperatures = [
    {"date": "2026-03-01", "temperature": 12.5},
    {"date": "2026-03-02", "temperature": 14.0},
    {"date": "2026-03-03", "temperature": 9.8},
    {"date": "2026-03-04", "temperature": 20.2},
    {"date": "2026-03-05", "temperature": 15.1},
    {"date": "2026-03-06", "temperature": 11.4},
]

total_temp = 0

for item in temperatures:
    total_temp += item["temperature"]

avg_temp = total_temp / len(temperatures)

print(f"Average temperature: {avg_temp:.2f}°C")

max_temp = temperatures[0]["temperature"]
hottest_date = temperatures[0]["date"]

for item in temperatures:
    if item["temperature"] > max_temp:
        max_temp = item["temperature"]
        hottest_date = item["date"]

print(f"Max temperature: {max_temp}°C")
print(f"Hottest day: {hottest_date}")

date_obj = datetime.strptime(hottest_date, "%Y-%m-%d")
formatted_date = date_obj.strftime("%A, %B %d, %Y")

print(f"Max temperature: {max_temp}°C")
print(f"Hottest day: {formatted_date}")
```

Sets

Sets are the last built-in datatype provided by Python we are going to cover.

A *set* is a collection which is **unordered** (you can't access values by index!), **mutable** (add/remove elements) and **do not authorize duplicates**.

```
my_set = {1, 2, 2, 3, 4, 5}
print(my_set)
print(len(my_set))
```

> It's the equivalent of a *set* in the mathematical sense: a collection of different things.

Adding and removing elements from a set

```
my_set = {1, 2, 2, 3, 4, 5}
my_set.remove(2)
my_set.add(6)
print(my_set)
```

Set operations

Sets are useful to make... **set operations**: union, intersection, difference, symmetric difference, etc.

```
setA={'a','b','c'}
setB={'b','c', 'd'}

# Union
print(setA.union(setB))
print(setA | setB) # union short notation with '|' operator

# Intersection
print(setA.intersection(setB))
print(setA & setB) # intersection short notation with '|' operator

# Difference
print(setA.difference(setB))
print(setB.difference(setA))
```

> Each operation returns a new set! Lean more about sets

Controlling the flow

To write (any interesting!) program you need:

- A way to **make decisions** based on tests/comparisons (e.g. do *this* when *that* happens)
- A way to **repeat** a set of instructions as many times as necessary (e.g. to iterate through a collection and do something to every item)

We need *flow control structures* to change the execution flow, to redirect, branch, loop, or interrupt that flow based on conditions (and data!) *at runtime*.

> With these two ingredients you can write *any* program!

Conditional jumps

Condition and predicate

- A **condition** is an expression that **can be evaluated with certainty** as *True* or *False*.
- A condition is also called a **predicate**.

For example, *"this sentence contains six words"* is a predicate that evaluates to *false*. *"I think you are wrong"* is a statement, but it is *not* a predicate.

Conditional jump with if/else statement

```
if <condition>:
    <instructions>
[else:
    <instructions>]
```

> [...] means it's optional. else block is optional

Example

```
number = 15

if number > 0:
  print(f"{number} is positive")
else:
  print(f"{number} is negative")

# Using 'falsy' values as predicat
if number:
  print(f"{number} is not equal to zero")
```

Why are we talking about "conditional jump"?

```
number = 15 #                          <= 1

if number > 0: #                       <= 2
  print(f"{number} is positive") # <= 3
  #Jump to <= 5                        <= 4
else:
  print(f"{number} is negative") # this is not executed!

x = 5 #                                <= 5
#...rest of the program                <= 6
```

> We "jump" from one instruction (or a block) to another one, based on a condition. This is how we control the flow of execution. One program, several possible execution flows (or paths), several possible outcomes!

Code indentation *matters* in Python

Python relies on **code indentation** (whitespace at the beginning of a line) **to define scope in the code** (if/else block, loop block, function, etc.)

> Other programming languages often use curly-brackets for this purpose.

```
a = 33
b = 200

# WRONG !
if b > a:
print("b is greater than a") # you will get an error!
print("Next instruction...")

# CORRECT !
if b > a:
  print("b is greater than a")
  print("Next instruction...")
```

> You can use spaces or tabs for indentation, but you must use **the same amount of indentation for all statements within the same code block**. By convention we will use **one tab** or **4 spaces**.

Truthy and *Falsy* values

- In Python, every value has an **inherent Boolean evaluation**, it can either be considered *True* or *False* in a Boolean context (eg: a `if` statement).
- Values that evaluate to `True` are called *truthy*, values that evaluate to `False` are called *falsy*.

Truthy values:

- Non empty strings or collections: "hello", [1, 2], (1,)
- Numeric values *not equal to zero*: 1, -4, 3.5, 1j
- `True`

Falsy values:

- Empty strings or collections: "", [], {}, " "
- Numbers: 0(integer), 0.0(float), 0j(complex)
- `None`, `False`

if/elif

Python will check each condition in order and **execute the first one** that is *true*. You can have as many `elif` statements as you need.

```
score = 75

if score >= 90:
  print("Grade: A")
elif score >= 80:
  print("Grade: B")
elif score >= 70:
  print("Grade: C")
elif score >= 60:
  print("Grade: D")

print("End of the program")
```

> As you can see, `else` statement is not mandatory

elif allows to avoid nesting if/else blocks

Multiple `if` / `else` statements can be *nested* to create an `else if` structure:

```
if condition1:
    instruction1
elif condition2:
    instruction2
else:
    instruction3
```

is equivalent to:

```
if condition1:
    instruction1
else:
    if condition2:
        instruction2
    else:
        instruction3
```

> Notice how the indentation stays at the first level with `elif`. Prefer the `elif` form (easier to read)!

Shorthand `if..else` notation

```
a = 5
b = 2

# One-line if statement
if a > b: print("a is greater than b")

# One-line if/else statement
print("A") if a > b else print("B")
```

Ternary operator, "inline" if/else statement

Pattern: `value_if_true if (condition) else value_if_false`

```
a = 5
b = 2

# Assign a value to 'bigger' based on a condition
bigger = a if a > b else b
print(bigger)
```

Shorthand if statements and ternary operators *should be used* when:

- You want to make a **quick assignment** based on a condition
- The condition and actions are **simple** and it **improves code readability**

> Remember: don't be "smart", be readable!

= is not ==

Remember, be careful to use the `==` operator to *compare* and `=` to assign !

```
# Assignment
x = 1
# Comparison
x == 1
if(x = 1): #Error!
  print("Oups!")
```

Loops (unconditional jumps)

- A *loop* is a control structure that allows you to **repeat some instructions a certain number of times**, or to *jump back* to previous instruction.
- `for` and `while` loops are the *fundamental loops* in every programming language, including Python.

For loops

We use range([start,] stop[,step]) function to **generate an iterable sequence of numbers (a 'range'):**

```
# From 0 (by default) to 5, step defaults to 1
for i in range(5):
  #Body of the for loop
  print(f'i={i}')
  print("Next value please")

# Use the list() function to convert a range to a list
type(range(1:5)
numbers = list(range(1:5))
type(numbers)
```

`i` is a variable, *local* to the *for loop*, that *sequentially* takes all the values generated by `range()`. The `print()` statements are, **the body of the loop** (indented), repeated *for each value*`.

Other examples

```
# From  2 to 8, step = 2
for k in range(2, 8, 2):
  print(f'k={k}')

# From  0 to 10, step = -1
for l in range(0, -10, -1):
  print(f'l={l}')
```

Practice: for loops and range()

Write *for loops* to produce the following sequences:

1. `2, 3, 4, 5`
2. `0, 2, 4, 6`
3. `0, -1, -2, -3`
4. `-10, -8, -6, -4, -2, 0`

> Each number will be printed on its own line with the `print()` function (no formating is required)

Correction: for loops and range()

```
# 1. `2, 3, 4, 5`
for i in range(2,6):
  print(i)

# 2. `0, 2, 4, 6`
for i in range(0, 8, 2):
  print(i)

# 3. `0, -1, -2, -3`
for i in range(0, -4, -1):
  print(i)

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

Going through a collection (list, string, dictionary)

A *for loop*, as we have already seen, is often used for going through iterables: **list**, tuple, **dictionary**, set or **string**:

```
# List
colors = ['orange', 'blue', 'green', 'orange']
for color in colors:
  print(f'{color}')

# String
word='hello'
for letter in word
  print(letter)

# Dictionary
my_animals = {"dog":['Medor', 'Max'], "cat":['Moustique', 'Plume']}

for (key, value) in my_animals.items():
  print(key, '=>', value)
```

Controlling the loop: `break` and `continue`

```
# What will be the result of this program ?

for x in range(6):
  if x == 1: continue
  if x == 3: break
  print(x)
else:
  print("Finally finished!")

print("The rest of the program")
```

- `break` : stop and quit the loop
- `continue` : stop the current iteration and jump to next iteration
- `else:` block: always executed at the end of the loop (optional)

The while loop

With the *while loop*, we can execute a set of statements **as long as a condition is true.**

```
while (condition):
  body of the loop
```

Example:

```
i = 1
while i < 6:
  print(i)
  i+=1
```

While loops and for loops

Any *loop* can be written as a *for loop* or as a *while loop*: **they are functionally equivalent!**

```
i=1
while i<12:
    print(i)
    i+=1

# Is equivalent to the for loop
for i in range(1, 12):
    print(i)
```

> *While loops* are often used when we *do not know in advance* when the loop will end (e.g. reading bytes from a file), whereas *for loops* are used when we know *beforehand* when it will end (e.g. iterating over a finite collection of items).

Practice : For and while loops

- Solve:
  - Problem 2
  - Problem 7

Working with dates, introduction to our first module

To manipulate *dates* and *datetimes* easily, we need to use a **module**.

A **module** (or library) is a **reusable** and **coherent unit of code**, created and maintained by other developers, giving some **specialized functionality**. It contains custom data types, variables, functions, etc.

Using the datetime module

To use a *module*, we need to import it in our code. To have the `datetime` *data type*, we have to import the corresponding *module* with the keyword `import`:

```
# From the 'datetime' module, import the datetime and timedelta data types
from datetime import datetime, timedelta

# Get current date and time
now = datetime.now()
# Format as a string
print("Today:", now.strftime("%Y-%m-%d %H:%M"))

# Date manipulation (add 7 days with timedelta() function)
next_week = now + timedelta(days=7)
print("In 7 days:", next_week.strftime("%Y-%m-%d"))
```

> The module `datetime` has already been installed on your machine with the Python interpreter!

Doing maths and using (pseudo)random number generators

```
import math
import random

print(math.sqrt(2), math.pi)
print(random.randint(1, 10))
print(random.randrange(1, 10))

# A list of 5 random integers between 1 and 100
print(random.sample(range(1,100),5))
```

Check methods available in random and math modules

> `datetime`, `math` and `random` are modules that are part of the *Python Standard Library* (PSL). They are pre-installed and shipped automatically whenever you install the Python interpreter on your machine

On naming things

- Name things *correctly* is important, because you think *through* words!
- Choosing a *good* word clarifies *intent*. An improperly named thing (variable, function, etc.) distorts the developer's mental representation of the program
- Try to find a *good name* for things
- When it's *hard* to find names, it means you don't fully understand (yet?) what you are trying to do.

> Naming things *well* is *hard* in programming (in real projects). It requires practice.

Naming conventions (style guide)

In this course, we will follow PEP 8, the style guide for Python Code:

- **Use 4 spaces** (or **one tab**) per indentation level. Don't mix both!
- **Choose** between single or doubled quotes for strings. Don't mix both!
- Variable names and function names should be lowercase, with words separated by underscores as necessary to improve readability (use `snake_case`)
- Constants are usually defined at the top of the file and **written in all capital letters with underscores separating words**. Examples include `MAX_OVERFLOW` and `TOTAL`.

Misc: Some useful *built-in* functions

- join(iterable), return a string which is the concatenation of the strings in iterable
- `time.sleep(n_seconds)`, from time package, to pause execution for a given amount of seconds

```
import time

n = input('Enter a positive number: ')
for i in range(0, n)
  print(i)

print(1, 2, 3, 4, sep='*', end='=')

vowels=['a','e','i','o','u','y']
'--'.join(vowels)

print("Start waiting...")
time.sleep(1)  # Pauses execution for 1 second
print("Done!")
```

> See all Python built-in functions and start to learn how to use (navigate) documentation

Let's take a look to our *Find the youngest person* program again

Find the youngest person program revisited

Now, you *should* understand *every* line of this program!

```
group = {
    "Alice": "1995-04-12",
    "Bob": "2002-11-23",
    "Charlie": "1988-08-05",
}

youngest_name = None
youngest_date = None

for name, date_str in group.items():
    if youngest_date is None or date_str > youngest_date:
        youngest_name = name
        youngest_date = date_str

print(f"The youngest person is {youngest_name}, born on {youngest_date}.")
```

Practice 1/3

1. **Write a Python program** `poem.my` to print the following string in a specific format (see the output below). Sample String : "Twinkle, twinkle, little star, How I wonder what you are! Up above the world so high, Like a diamond in the sky. Twinkle, twinkle, little star, How I wonder what you are"

```
Twinkle, twinkle, little star,
 How I wonder what you are!
  Up above the world so high,
  Like a diamond in the sky. Twinkle, twinkle, little star,
 How I wonder what you are
```

2. **Write a Python** program `current_datetime.py` to display the current date and time. Output Example:

```
Current date and time:
2026-09-07 14:09:12
```

Hint: use the strftime() method.

> From Prof. Karim ZKIK, PhD Materials

Practice 2/3

3. **Write a Python** program `geometry.py` that calculates and print the area of a circle based on a given radius. Sample output:

```
radius=1.1
Area= 3.8013271108436504
```

Practice 3/3

4. **Write a Python program** `input_to_datastructures.py` that accepts a sequence of comma-separated **integers** from the user and generates a list and a tuple of those numbers. We suppose that the input is *valid* (valid format). Sample output:

```
Enter a sequence of comma-separated numbers: 1,2,3,4
List: ['1', '2', '3', '4']
Tuple: ('1', '2', '3', '4')
```

5. **Enhance** the previous program by allowing the user to enter a sequence of comma-separated numbers (integers **or floats**). Sample output:

```
Enter a sequence of comma-separated numbers: 1,-2.3,3,4.5
List: ['1', '-2.3', '3', '4.5']
Tuple: ('1', '-2.3', '3', '4.5')
```

> *Hint*: use str.split() method, tuple() and list() functions.

> Start to **organize your sources properly** on your machine! (directories, naming convention, comments, etc.)

Practice! Problems and DataCamp labs

- **Complete** the following DataCamp Labs :
  - Introduction to Python (Course)
    - Chapter 1: Python Basics
    - Chapter 2: Python Lists
  - Introduction to Python (Practice)
  - Data Types in Python (Course):
    - Chapter 2: Dictionaries - The Root of Python
- View the problem sheet, try to solve problems starting with Problem 1. You should be able to solve all problems from 1 to 10.

Next Module

Go to next module

References

- Python official documentation, *everything* you need to know about the language. This is the *ultimate reference* to find and *check* information. For example, check the documentation on the introduction to lists
- List methods, official documentation
- String methods, official documentation
- PEP 8 – Style Guide for Python Code, use standard conventions to write your code. People **spend way more time reading code** than writing it (even more in the AI era...). Make your code *stylistically consistent* to help others (including yourself!) read and (re)use it!
- While vs For loops, an interesting and complete answer on the difference between the two constructs
- Always use [closed, open) intervals, by Fernando Hurtado Cardenas

Going further / Extra information

On semicolons in Python

- Python **does not require semicolons** to terminate statements.
- Semicolons **can be used** to delimit statements if you wish to put **multiple statements on the same line**.

```
x = 2
y = 3
#Equivalent to
x = 2; y = 3
```

Be careful when comparing floats 1/2

In Python, `==` is the **comparison operator**. It returns `True` if operands are the same value, `False` otherwise.

```
0.2 == 0.2
True # So far so good!
0.1 + 0.2 == 0.3
False # ... Wait... What?! Why?! Computers are broken!
```



Be careful when comparing floats 2/2

- Numbers, like everything else, is encoded in binary (base 2)
- Numbers are stored with a **finite precision**! We do not have *infinite* memory to represent them!
- In base 10 (decimal notation), `0.1` **is** finite. But, in base 2 (binary format), `0.1` is an **infinite sequence of digits!** $0.1_{10} = 0.0001100110011001\ldots_2$

```
>>> 0.1 + 0.2
0.30000000000000004
```

> Similarly, in base 10 (decimal notation), $1/3$ can not be represented with a finite sequence of numbers (0.33333333333...)!

Raw strings

A **raw string** is a string literal **prefixed** with an `r` or `R` (e.g., r"text"). It tells the Python interpreter to treat backslashes (`\`) as literal characters rather than as *escape* characters.

```
# Standard string: \n creates a new line
standard_str = "Line 1\nLine 2"
print(standard_str)

# Raw string: \n is treated as two literal characters '\' and 'n'
raw_str = r"Line 1\nLine 2"
print(raw_str)

# No need to escape every backslash like before
path = r"C:\Users\Name\Documents\notes.txt"
print(path)
```

Walrus Operator, another assignment operator

- The statement `n = 5` does *one thing*: **stores the value 5** in the variable `n`'s **memory location**. This assignment is NOT an expression.
- The statement `n := 5` (with Walrus Operator `:=`) does **two things**:
  - **stores the value 5** in the variable `n`'s **memory location**;
  - **evaluates to 5** (or "returns 5"). This assignment IS an *expression*.

```
1 == (n=1) #SyntaxError: invalid syntax.
1 == (n:=1) #True: 'n:=1' affects 1 to n AND evaluates to 1
```

Loop with index using enumerate() function

Eliminate manual indexing and write cleaner, more readable Python loops with the `enumerate()` function:

```
names = ["Lamport", "Von Neumann", "Abelson", "Van Rossum", "Ritchie", "Thompson"]

# Cleaner syntax (unpack index and value in a tuple)
for i, name in enumerate(names):
        print(name)
```