

# Fundamentos de Programação 2025-2026

Programming Fundamentals

Class #2

# Overview

- Python
- Values and Types
- Variables
- Input/output
  
- Conditional execution
  - If statement
  - If-else
  - If-elif-else
- Boolean expressions
  - The bool type
  - Relational operators
  - Logical operators
  - Properties
- Conditional expression

# PROBLEM

- **Coffee Price Calculator**

At a coffee shop, each cup of coffee costs €1.50. A customer orders several coffees, and the cashier wants to know how much they need to pay in total.

Write a program to help with this.

# Solution ?

- What do you propose as an agile solution for the problem of calculating **Coffee Price** ?
- How to do it?
- Suggestions ?

# The complete process

Includes the mental steps you take to obtain the algorithm.

- **Read & restate the problem**
  - Restate in one sentence: A cashier needs the total to charge a customer who ordered some coffees (each €1.50).
- **Identify inputs and outputs**
  - Input: number of coffees the customer wants.
  - Output: total amount to pay (currency, with two decimals).
- **List constraints and assumptions**
  - Price per coffee is fixed (€1.50).
  - Number of coffees is a whole number ( $\geq 0$ ).
  - Handle obvious edge cases (0 coffees, very large numbers, non-numeric input).
- **Work a manual example**
  - If number = 4  $\rightarrow$  total =  $4 \times 1.50 = \text{€}6.00$ .
  - Doing this by hand helps spot rounding/format issues.

# The complete process (cont.)

- Find the formula / core idea
  - $\text{total} = \text{number\_of\_coffees} * \text{price\_per\_coffee}.$
- Choose variables and data types
  - `n` (integer) for number of coffees.
  - `price` (real) = 1.50.
  - `total` (real) for result.
- Decide extra behaviour
  - How to treat invalid input? (e.g., re-ask or show an error)
  - How to format output? (e.g., two decimal places, € sign)
- Write step-by-step algorithm / pseudocode
  - Produce a straightforward sequence of operations

# The complete process (cont.)

- Translate algorithm to code
- Test
  - Try several inputs: 0, 1, 3, 1000, and an invalid input like ``-2`` or ``abc``.
  - Verify output correctness and formatting.
- Refine & document
  - Add input validation, friendly messages,
  - and comments so you and others understand each step.

# Possible Algorithm

## Algorithm: Coffee Price Calculator

- **Start**
- **Read** the number of coffees the customer wants (from user input).
- **Store** the fixed price of one coffee (€1.50)
- **Multiply** the number of coffees by the price of one coffee to get the total cost.
- **Display** the total cost with a clear message.
- **End**



# What we need to know to complete the process?

- What is Python
  - And its specificities
- Display a prompt
- Read the input
- Call it ``n``
- ...
- Run the program

# PYTHON BASICS

# Note

- For this course we will adopt Python...
- But the most important topics are applicable to other languages.

# Python

- Python is a **general-purpose programming language**, well known for its elegant syntax and readable code.
  - With Python it is possible to do everything, from GUI development, web applications, system administration tasks, data analysis, visualization, etc.
- Python is an **interpreted language**.
  - An interpreter parses and executes a program on a line-by-line basis.
  - This is slower than executing a compiled program.
- In Python, basic data structures and small utility functions are built-in
  - you don't need to define them.
- Moreover, Python has hundreds of extension libraries (modules)
  - available at <https://pypi.python.org/>

# Python in interactive mode

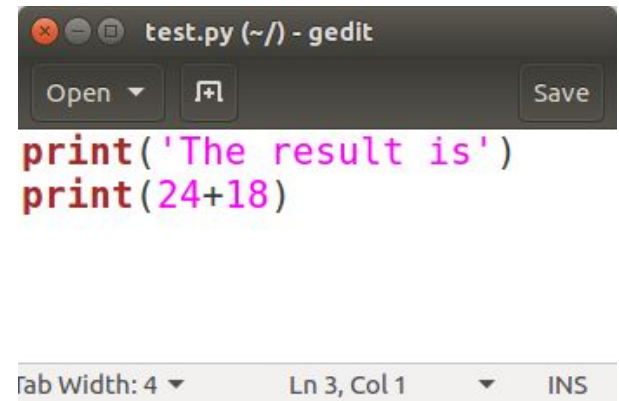
- There are two ways to use the interpreter: interactive mode and script mode.
- Execute python with **no argument** to run in **interactive mode**.
- Then, type Python statements and the interpreter displays the result:  

```
$ python3  
>>> 1 + 1  
2  
>>>
```
- The chevron, **>>>**, is **the prompt** the interpreter uses to indicate that it is ready.
- When you type an expression, the interpreter prints the result. Then, it shows the prompt again.

# Python in script mode

- Alternatively, you can store the statements in a file and use it the interpreter to execute it whenever you want
  - The file is called a **script or program**
- By convention, Python scripts have names that end in **.py**.
- To execute the script, just call the interpreter with the name of the file as argument. For example:

```
$ python3 test.py
The result is
42
```
- In **script mode**, Python does not show the prompt and does not print results automatically.
  - You need to call `print()` for that.

A screenshot of a gedit text editor window titled 'test.py (~/) - gedit'. The window contains two lines of Python code: `print('The result is')` and `print(24+18)`. The code is color-coded with red for the `print` function and magenta for the strings and numbers. Below the code, the status bar shows 'Tab Width: 4', 'Ln 3, Col 1', and 'INS'.

# Script mode programming

- Indentation
  - **Blocks of code are denoted by line indentation**, which is rigidly enforced.
  - The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.
- Statements in Python typically end with a new line.
  - However, a backslash ( \ ) in the end of the line indicates that the statement continues in the next line.
  - The semicolon ( ; ) allows multiple statements on a single line.
- A line containing only whitespace is known as a blank line and Python totally ignores it.
- To execute a script, pass it as argument to the interpreter:  
`python3 script.py`

In general programming implies handling data/information

# **HANDLING DATA**



# Values and types

- A **value** is a piece of data, such as a letter or a number:  
`33, 3.14, 'ola', 1+2j`
- Values belong to different **types** (or **classes**): int, float, str, complex.
- Use the **type** function to find the type of a value:  

```
>>> type('Hello, World!')  
<class 'str'>  
>>> type(17)  
<class 'int'>  
>>> type(3+5j)  
<class 'complex'>
```
- Types determine what you can do with values.
  - For instance, you cannot add an integer to a string:  

```
>>> 3 + 'cats'  
TypeError: unsupported operand type(s) for +: ...
```

# AI usage Example (to learn)

- What are data types?
- Why do we need this different types?
- How many different types do programming languages have?
- Prompt:
  - Explain **data types** in simple terms, like you're talking to someone who is just starting to learn programming. Focus on what it is, why we use it, and its main purpose.

# Copilot

## What is a Data Type?

A **data type** is a way for the computer to understand **what kind of information** you're working with. Think of it like labeling a box:

- If the box says "Books," you know it holds books.
- If it says "Snacks," you expect food inside.

In programming, we label our data so the computer knows how to handle it.

# Copilot

## Why Do We Use Data Types?

We use data types to:

- **Avoid confusion:** The computer needs to know if you're dealing with text, numbers, or something else.
- **Use memory efficiently:** Different types of data take up different amounts of space.
- **Prevent errors:** If you try to add two pieces of text like numbers, the computer can warn you.

# Copilot

## Main Purpose of Data Types

The main purpose is to **tell the computer how to store and work with your data**. It's like giving instructions:

"Hey computer, this is a number—do math with it."

"Hey computer, this is text—don't try to add it like a number."

# Copilot

## Common Data Types in Programming

- **Integer** – Whole numbers (e.g., 1, -42)
- **Float / Double** – Decimal numbers (e.g., 3.14, -0.001)
- **Boolean** – True or False values
- **Character** – A single letter or symbol (e.g., 'A', '7')
- **String** – A sequence of characters (e.g., "Hello world")
- **Array / List** – A collection of values (e.g., [1, 2, 3])
- **Object / Struct** – A group of related data under one name
- **Null / None / Undefined** – Represents no value or empty data
- **Date / Time** – Stores calendar dates and timestamps
- **Tuple** – An ordered, immutable group of values
- **Set** – A collection of unique values
- **Dictionary / Map** – Key-value pairs (e.g., {"name": "António"})

# Data Types (Python)

- Python has several *built-in data types*, including:
  - **Numeric types:** `int`, `float`, `complex`, e.g. `13`, `3.1`, `1-2j`
  - **Boolean:** `bool` (with values `True` and `False`).
  - **Strings:** `str`, e.g. `'Hello'`
  - **Lists:** `list`, e.g. `[1, 3, 1]`
  - **Tuples:** `tuple`, e.g. `(3, 'May', 1981)`
  - **Sets:** `set`, e.g. `{2, 3, 5}`
  - **Dictionaries:** `dict`, e.g. `{'eggs': 6, 'beer': 0.33}`
- You can also define new data types – called classes
  - We leave those for another course.

# Conversion between types

- Sometimes we need to convert values to a different type.
- We use type conversion functions: `str`, `int`, `float`, ...
- Just about any kind of value may be **converted to string** with `str`:

```
>>> str(1+2)
'3'
>>> str(1.0/2)
'0.5'
```

In fact, this **happens implicitly when you print values**.

- Numeric strings may be converted to `int` or `float`:

```
>>> 100 + int('33')
133
>>> float('0.12') / 10000
1.2e-05
```

- Converting a `float` to `int` *truncates* towards zero:

```
>>> int(2.78)
2
```



# Variables

- A **variable** is a name (identifier) that refers to a value.
- This is true, but as (future) Engineer I would like to know more:
  - What problem(s) originated variables?
  - Why use them?

# Learning more on variables using AI

- **Explain variables** used in Programming languages in simple terms, like you're talking to someone who is just starting to learn programming. Focus on **what they are, why we use them, the problem(s) that they solve, and its main purpose.**

# Copilot



## What Is a Variable?

A **variable** is like a **named container** where you store information. Imagine you have a jar labeled “cookies.” You can put cookies in it, take them out, or replace them with something else. In programming, a variable works the same way—it holds a value, and you give it a name so you can use it later.

Example:

Python

 Copiar

```
score = 10
```

Here, `score` is the name of the container, and `10` is the value inside.

# Copilot

## Why Do We Use Variables?

Because we don't want to repeat ourselves or hard-code values everywhere. Variables help us:

- **Store data** that we can use and change later
- **Make code flexible**—you can update the value without rewriting your whole program
- **Make code readable**—names like `price`, `username`, or `total` tell us what the data means

# Copilot

## What Problem Do Variables Solve?

Without variables, every time you need a value, you'd have to write it manually. That's:

- **Hard to manage**
- **Error-prone**
- **Impossible to scale**

Variables solve this by letting you **reuse and update data easily**. You can change one value, and the whole program adapts.

# Copilot

## **Main Purpose of Variables**

The main purpose is to:

- **Hold information** while your program runs
- **Let you work with that information**—do math, make decisions, display results
- **Keep your code clean, organized, and dynamic**

## **Real-Life Analogy**

Think of a variable like a name tag at a party.

You don't say "Hey, person wearing a red shirt with glasses who likes jazz"—you say "Hey António!"

In code, instead of repeating a value, you just use the variable name.

# Variable names

- Variable names **must begin with a letter** and may include both letters and digits.
  - **Some *keywords* cannot be used as variable names**
    - such as: `def`, `if`, `not`, `and`, etc.
- If you give a variable an illegal name, you get a **syntax error**:

```
>>> 76trombones = 'big parade'  
SyntaxError: invalid syntax
```

# Keywords

- The interpreter uses **keywords** to recognize the structure of the program.
- Keywords are reserved words:
  - they **cannot be used as variable names or any other identifier**.
- In Python3, the keywords are:

```
False class finally is          return
None      continue for         lambda      try
True      def      from        nonlocal while
and       del      global      not          with
as        elif     if          or          yield
assert    else     import      pass
break     except   in          raise
```



# Assigning values to variables

- An **assignment statement** assigns a value to a variable.

```
>>> n = 5
```

```
>>> pi = 3.14
```

- In an expression, a variable is substituted by its value.

```
>>> 2*pi*n
```

```
31.4000
```

# Reassignment



- You can assign a new value to an existing variable.

```
>>> n = 5
>>> print(n)
5
>>> n = n + 1 # Is this valid? YES!
>>> print(n)
6
```

[Play ▶](#)

- The variable forgets the old value and stores the new one until the next assignment!
- Therefore, a variable may take different values during the execution of a program and the order of operations is important!

Variable assignment is a fundamental operation  
in imperative programming languages!

# More on assignment

- There are special **augmented assignment operators**:

<code>n += 1</code>	equivalent to	<code>n = n + (1)</code>
<code>x -= pi</code>	equivalent to	<code>x = x - (pi)</code>
<code>n *= 1+p</code>	equivalent to	<code>n = n * (1+p)</code>
<code>x /= 2.2</code>	equivalent to	<code>x = x / (2.2)</code>
<code>n %= 3</code>	equivalent to	<code>n = n % (3)</code>

- Python allows **simultaneous assignment** like this:  
`name, age, height = "Maria", 21, 1.63`

# Operators

- Operators are special **symbols or words that represent computations.**

+   -   \*   /   \*\*   %   <=   or   and   ...

- The values combined by operators are called **operands.**
- For a given operator, operands must have compatible types.
- The result type depends on the operand types.

# Operators precedence

- When more than one operator appears in an expression, the **order of evaluation depends on the rules of precedence**
  - **P**arentheses, **E**xponents, **M**ultiplication/**D**ivision, **A**ddition/**S**ubtraction
    - mnemonic [PEMDAS](#).
- Use parentheses to make it obvious!
- The + operator performs concatenation in strings.
- The \* operator also works on strings;
  - it performs repetition. For example, 'Ah'\*3 is 'AhAhAh'.
- It is highly recommended to add notes to a program to explain in natural language what the program is doing. These notes are called comments, and they start with the # symbol.

# Arithmetic Operators Precedence

Descending precedence. Same color = same precedence.

Operator	Example	Meaning	Result
<b>+</b> (unary)	<b>+a</b>	<b>Unary Positive</b>	<b>a</b>
<b>-</b> (unary)	<b>-a</b>	<b>Unary Negation</b>	<b>a</b> with opposite sign
<b>**</b>	<b>a ** b</b>	<b>Exponentiation</b>	<b>a</b> raised to the power of <b>b</b>
<b>*</b>	<b>a * b</b>	<b>Multiplication</b>	Product of <b>a</b> and <b>b</b>
<b>/</b>	<b>a / b</b>	<b>Division</b>	Quotient when <b>a</b> is divided by <b>b</b> . The result is a float (or complex).
<b>//</b>	<b>a // b</b>	<b>Floor Division</b> (or <b>Integer Division</b> )	Quotient when <b>a</b> is divided by <b>b</b> , rounded to the next smallest whole number.
<b>%</b>	<b>a % b</b>	<b>Modulo</b>	Remainder when <b>a</b> is divided by <b>b</b> .
<b>+</b> (binary)	<b>a + b</b>	<b>Addition</b>	Sum of <b>a</b> and <b>b</b> .
<b>-</b> (binary)	<b>a - b</b>	<b>Subtraction</b>	<b>b</b> subtracted from <b>a</b> .

# Expressions and statements

- An **expression** is a combination of values, variables, and operators that results in a value.
- A **statement** is a unit of code that the Python interpreter can execute.
- Expressions *are* statements that have a resulting value.
- Other kinds of statement have no result, so they are not expressions.
- In script mode, an expression, all by itself, has no visible effect (unlike interactive mode).

# Getting input from the console

- The `input` function is used to get input from the console.
- It has an optional argument called the *prompt* and returns a string.

```
>>> name = input("What's your name? ")
What's your name? tim
>>> name
'tim'
```

- To get other types of values, you must convert!

```
>> age = int(input("Age? "))
Age? 22
>>> age
22
>>> type(age)
<class 'int'>
```



# Sending output to the console

- To output text to the screen, use the **print** function:

```
print("Hello World")
```

- To **print multiple values** (separated by blanks):

```
print("speed =", v)
```

- To write **multiple lines**, add the **'\n'** character:

```
print("Hello World\nThis is a message")
```

- The print function has some **optional keyword arguments**:

```
print(..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

- Use **sep=** to change how arguments are separated
- Use **end=** to change how the output is terminated.
- Use **file=** to send output to a different stream (e.g. file).

```
fh = open("data.txt", "w")  
print("Some text", file=fh)  
fh.close()
```

# Homework

- Write algorithms and pseudocode for the other 2 simple problems (Movie Ticket Counter, and Temperature Conversion).
- Write an algorithm for withdrawing money from an ATM.
- Do these [codecheck exercises](#).



# Simple **sequential** algorithms



- With variables, operators, input and output we can solve simple problems, for example:
  - Conversion of temperatures in degrees Fahrenheit to degrees Celsius
  - Conversion of Euros to Dollars
  - Calculate the price of several cups of coffee
  - ...

# Solution

START

DISPLAY "Enter temperature in Fahrenheit:"

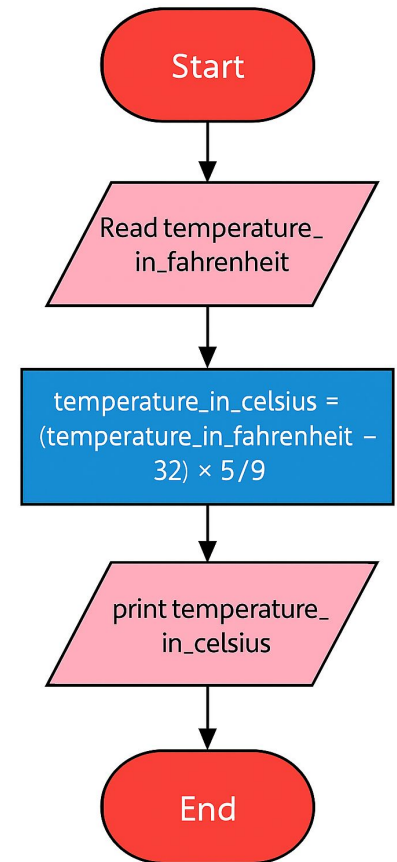
READ temperature\_in\_Fahrenheit

COMPUTE temperature\_in\_celsius =  
 $(\text{temperature\_in\_fahrenheit} - 32) \times 5 \div 9$

DISPLAY "Temperature in Celsius is: " +  
temperature\_in\_Celsius

END

[Flowchart Example: Temperature to Celsius Conversion | Fluxograma Template](#)



# Motivation problem

- We already have addressed the conversion from Fahrenheit to Celsius.
- Now we want you to solve a derived problem:
  - Convert the value of the water temperature given by a user in degrees Fahrenheit and **show a warning message if the water is boiling** (temperature above 100 degrees Celsius).

# Solution (in pseudocode)

START

DISPLAY "Enter the temperature in Fahrenheit :"

READ temperature\_in\_fahrenheit

COMPUTE  $\text{temperature\_in\_celsius} = (\text{temperature\_in\_fahrenheit} - 32) \times 5 \div 9$

IF temperature\_in\_celsius more than 100

    DISPLAY "The water is boiling!"

ELSE

    DISPLAY "The water is not boiling."

END

# **CONDITIONAL EXECUTION**

# Conditional execution (1): simple **if**

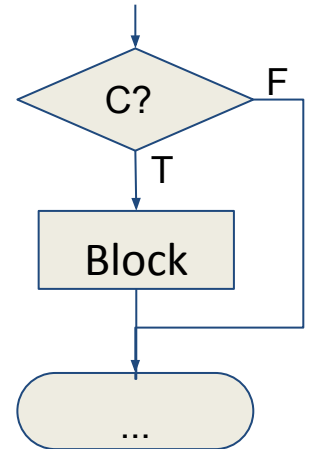
- **Conditional statements** allow the program to check conditions and change its behavior accordingly.

- The condition is evaluated...
  - If true, a block of statements is executed.
  - If not, execution continues after the block.

- The simplest form is the **if statement**:

```
if condition:  
    block_of_statements  
...
```

- The **condition** should be a boolean expression.  
(Actually, it may be of any type, as it is implicitly converted to bool, but this can be confusing and should be avoided.)
- The block must have one or more indented statements.





# Example

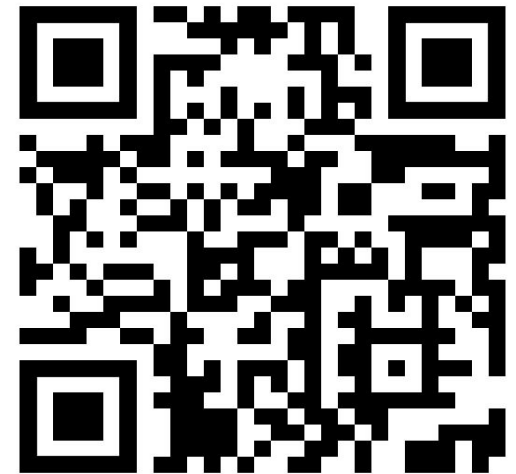
```
N = int(input("N? "))  
  
if N > 10:  
    print("A")  
  
if N % 2 == 0:  
    print("B")  
  
print("END")
```

[Play ▶](#)

- What is the output if  $N = 3$ ?
- What if  $N = 4$ ?
- And if  $N = 13$  or  $N = 14$ ?

Answer questions on:

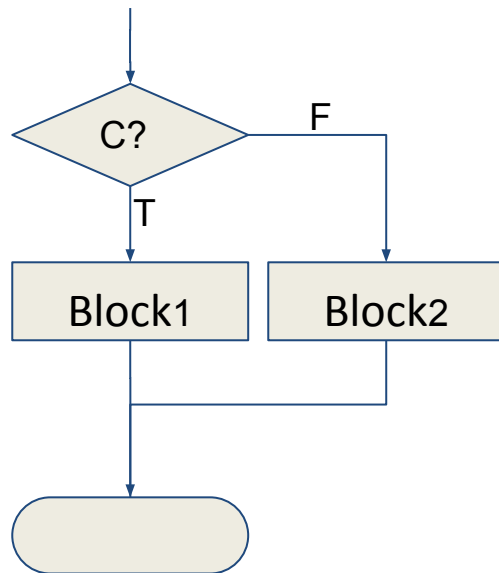
<https://forms.gle/cfjsNAHt8xov5VGP7>



[Edit](#) [Responses](#)

# Conditional execution (2): **if - else**

- A second form of the if statement allows **selection between two alternative paths**.
  - The condition determines which one gets executed.



```
x = 3
```

```
if x%2 == 0:
```

```
    R = 'even' # Block1
```

```
else:
```

```
    R = 'odd' # Block2
```

```
print(x, 'is', R)
```

[Play ▶](#)

# Conditional execution (3): **if - elif - else**

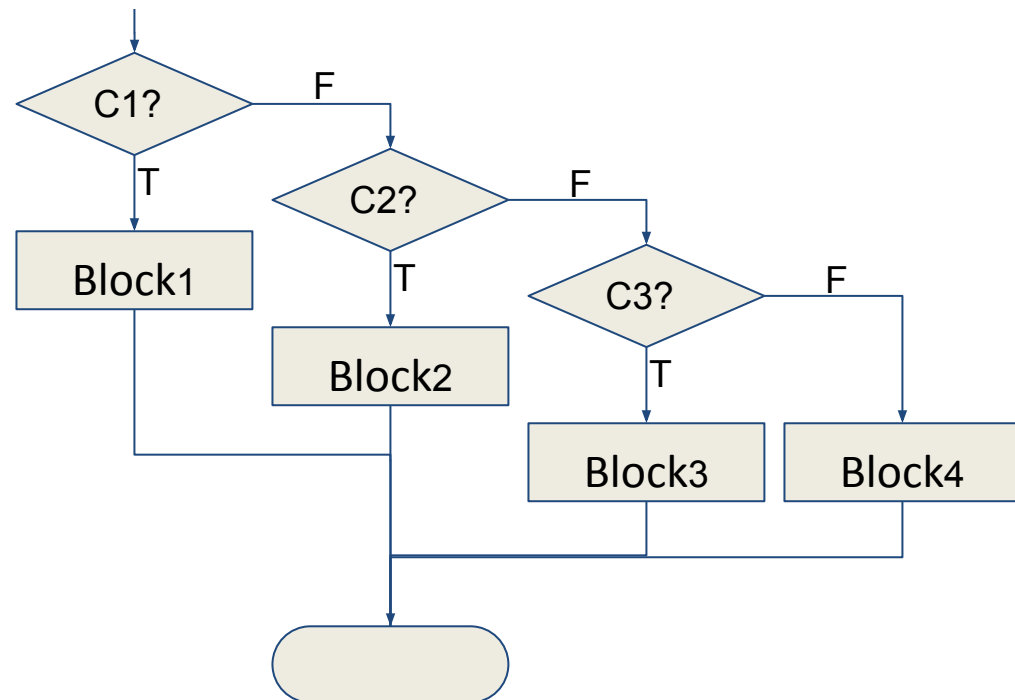
- Sometimes there are **more than two alternatives**, and we need more than two branches (**chained conditional**).

- Example:

```
x = 12
```

```
if x < 10: #C1
    mark = 'Poor'
elif x < 13: #C2
    mark = 'Fair'
elif x < 17: #C3
    mark = 'Good'
else:
    mark = 'Excellent'
```

```
print(mark)
```



[Play ▶](#)

# Conditional statement semantics

- Which conditions select each block of statements?

```
if C1:
    Block1      ← Block1 is executed iff C1
elif C2:
    Block2      ← Block2 is executed iff  $\neg C1 \wedge C2$ 
elif C3:
    Block3      ← Block3 is executed iff  $\neg C1 \wedge \neg C2 \wedge C3$ 
else:
    Block4      ← Block4 is executed iff  $\neg C1 \wedge \neg C2 \wedge \neg C3$ 

Rest           ← is always executed
```

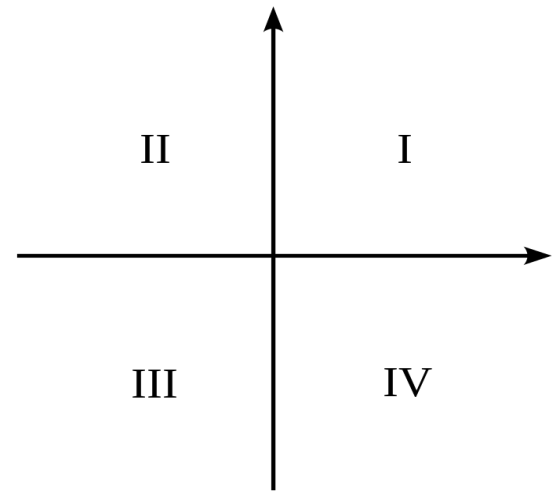
$\neg C1$  means not C1

# Nested conditional statements

- Conditional statements may be nested within each other.
- Application Example – Determine quadrant in 2 dim plane

```
if y > 0:
    if x > 0:
        quadrant = 1
    else:
        quadrant = 2
else:
    if x < 0:
        quadrant = 3
    else:
        quadrant = 4
```

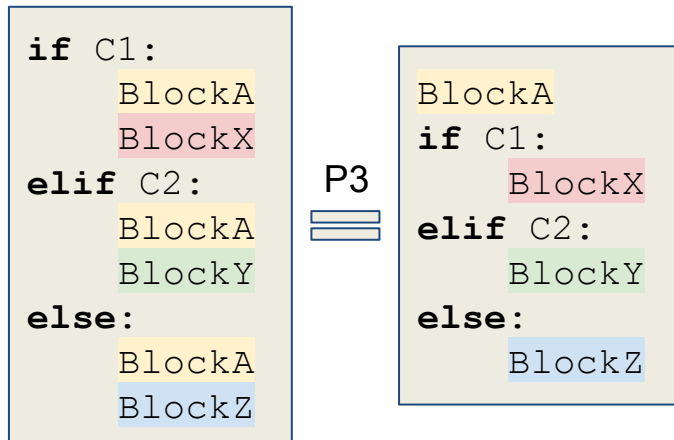
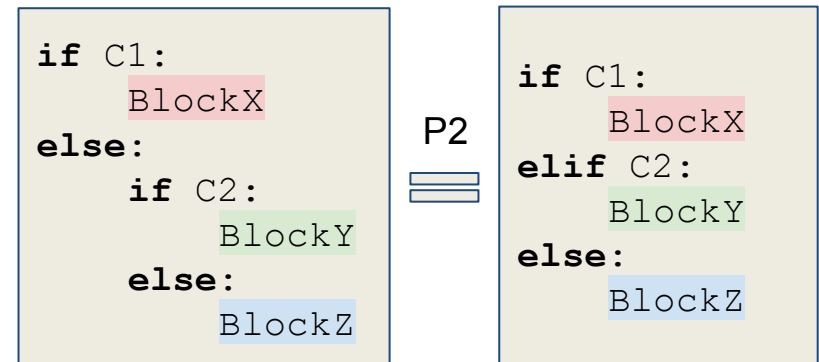
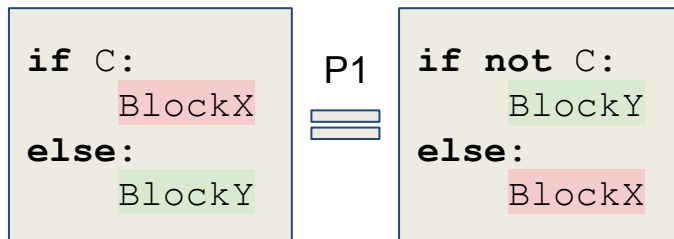
[Play ▶](#)



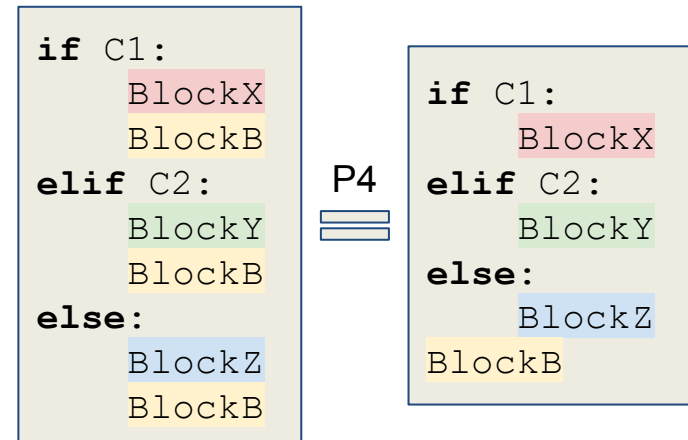
- Although the indentation makes the structure apparent, **deeply nested conditionals become difficult to read.**
- Try to simplify nested conditional statements
  - Using, for example, equivalence properties.

# Program equivalence properties

For *well-behaved* blocks of statements, the following properties apply.



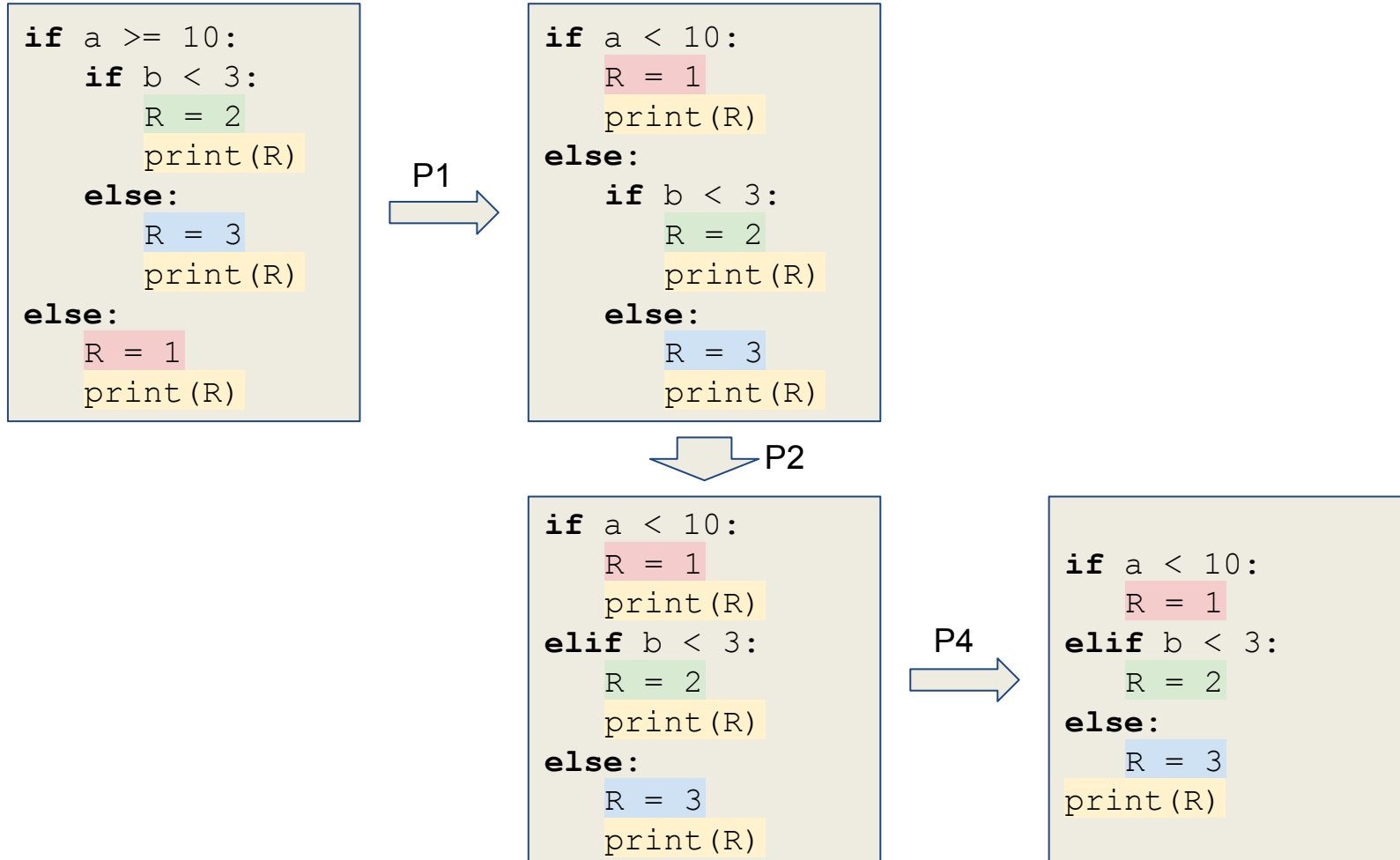
(If C1, C2 have no side effects.)



(If C1, C2 have no side effects.)

# Example: code simplification

- Applying equivalence properties may simplify the code.



# Conditional expression

- Python also includes a conditional expression, based on a **ternary operator**:

`expression1 if condition else expression2`

- Uses keywords `if` and `else`, but it **is an expression**!
- The **condition** is evaluated first.
  - If true, then `expression1` is evaluated and is the result.
  - If false, then `expression2` is evaluated and is the result.

```
n = int(input("number? "))
msg = "odd" if n%2!=0 else "even"
print(n, "is", msg)
```



# BOOLEAN EXPRESSIONS

# Boolean expressions

- A **boolean expression** is an expression that is either true or false.

```
>>> n = 5          # this IS NOT a boolean expression!
>>> n == 5         # this IS a boolean expression!
True
>>> 6 == n         # this is another boolean expression.
False
```

- True and False are special values that belong to the type `bool`.

# Boolean expressions

- Boolean values may be **stored in variables**.

```
>>> isEven = n%2==0
```

- May be **converted** to string.

```
>>> str(isEven)
'False'
```

- Or to integer.

```
>>> int(False) # 0
>>> int(True)  # 1
```

## Null and empty values convert to False:

```
>>> bool(0)          # False
>>> bool(0.0)        # False
>>> bool('')         # False
>>> bool([])         # False
```

## Other values convert to True:

```
>>> bool(1)          # True
>>> bool('False')    # True (surprise!)
>>> bool([False])    # True (surprise?)
```

# Relational and logical operators

- **Relational operators** produce boolean results:

```
x == y      # x is equal to y
x != y      # x is not equal to y
x > y       # x is greater than y
x < y       # x is less than y
x >= y      # x is greater than or equal to y
x <= y      # x is less than or equal to y
x < y < z   # x is less than y and y is less than z (cool!)
```

- There are three **logical operators**: and, or, not.


```
x >= 0 and x < 10      # x is between 0 and 10 (exclusive)
0 <= x and x < 10      # same thing
x == 0 or not isEven and y/x > 1
```

- How do you check if X is greater than Y and Z?

a) `X > Y and Z`      b) `X > Y and X > Z`      c) `Y < X > Z`

a)  WRONG

b)  CORRECT

c)  Correct, but confusing

# Properties

- Remember these properties:

$x == y$	$\Leftrightarrow$	$\text{not } (x \neq y)$	$\Leftrightarrow$	$y == x$
$x \neq y$	$\Leftrightarrow$	$\text{not } (x == y)$	$\Leftrightarrow$	$y \neq x$
$x > y$	$\Leftrightarrow$	$\text{not } (x \leq y)$	$\Leftrightarrow$	$y < x$
$x \leq y$	$\Leftrightarrow$	$\text{not } (x > y)$	$\Leftrightarrow$	$y \geq x$

- And these (where A, B, C are boolean):

$\text{not } (\text{not } A)$	$\Leftrightarrow$	$A$
$\text{not } (A \text{ and } B)$	$\Leftrightarrow$	$(\text{not } A) \text{ or } (\text{not } B)$
$\text{not } (A \text{ or } B)$	$\Leftrightarrow$	$(\text{not } A) \text{ and } (\text{not } B)$
$A \text{ or } B$	$\Leftrightarrow$	$B \text{ or } A$
$A \text{ and } B$	$\Leftrightarrow$	$B \text{ and } A$
$A \text{ or } (B \text{ and } C)$	$\Leftrightarrow$	$(A \text{ or } B) \text{ and } (A \text{ or } C)$
$A \text{ and } (B \text{ or } C)$	$\Leftrightarrow$	$(A \text{ and } B) \text{ or } (A \text{ and } C)$

# Precedence rules

- Arithmetic > relational > not > and > or.
- Example (starting from the lowest priority operation):

$x \leq 1 + 2 * y ** 3 \text{ or } n \neq 0 \text{ and not } 1/n \leq y$

$(\underline{x \leq 1 + 2 * y ** 3}) \text{ or } (\underline{n \neq 0 \text{ and not } 1/n \leq y})$

$(x \leq (\underline{1 + 2 * y ** 3})) \text{ or } ((\underline{n \neq 0}) \text{ and } (\underline{\text{not } 1/n \leq y}))$

$(x \leq (1 + (\underline{2 * y ** 3}))) \text{ or } ((n \neq 0) \text{ and } (\text{not } (\underline{1/n \leq y})))$

$(x \leq (1 + (2 * (\underline{y ** 3})))) \text{ or } ((n \neq 0) \text{ and } (\text{not } ((\underline{1/n}) \leq y)))$

- Homework: Try starting from the highest priority operation.

# Short-circuit evaluation

- Operators **and** and **or** only evaluate the second operand if needed!

```
A and B    # if A is false then A, otherwise B
A or B     # if A is true then A, otherwise B
```

- This is called **short-circuit evaluation**.
- It can be very useful. For example, these 2 conditions are equivalent for every  $n \neq 0$ .

A)  $1/n > 2$  and  $n \neq 0$

B)  $n \neq 0$  and  $1/n > 2$

But, for  $n = 0$ :

- A produces a **ZeroDivisionError**, whereas
  - B is False (because  $1/n$  is not evaluated). B is probably preferable.
- The order of the operands is important!**

# Exercises

- [Review exercises](#) (= aula02 ex 1)

