Programmering af Mobile Robotter

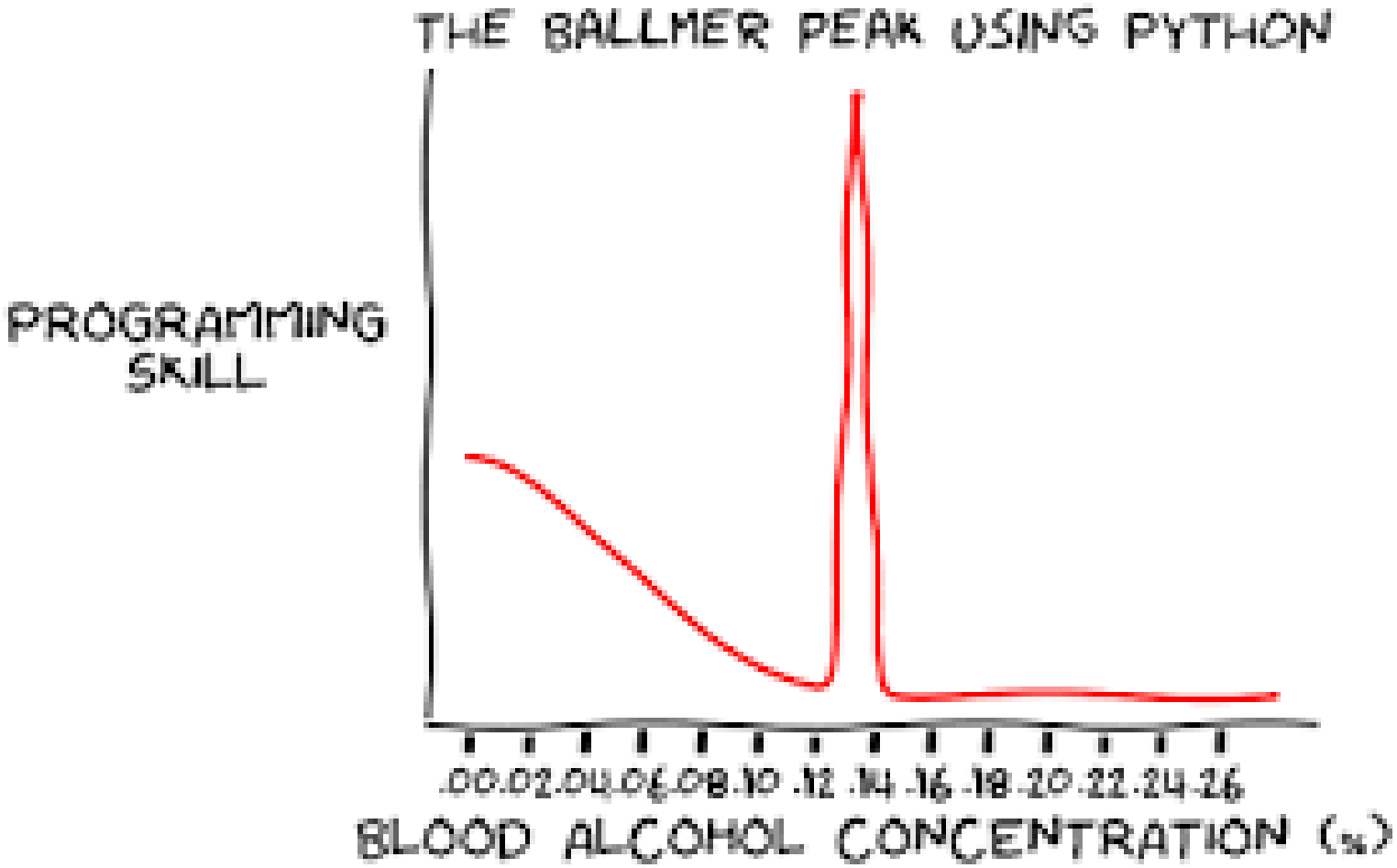
RB1-PMR – Module 2: Basic programming concepts using Python

Jes Hundevadt Jepsen | jegj@mmmi.sdu.dk
SDU Drone Center, MMMI, University of Southern Denmark, Odense, Denmark

Agenda

- Recap of last module
- Introduction to programming
- **Basic concepts in programming** (using Python)
  - Operators (Arithmetic, conditional, and Logical)
  - Built-in **data types** and **functions**
  - **Conditional statements** and **loops**
  - **Functions** and error handling
  - Troubleshooting and Debugging (basic)
- Extra Credit Activity 1: Calculator
- Assignments

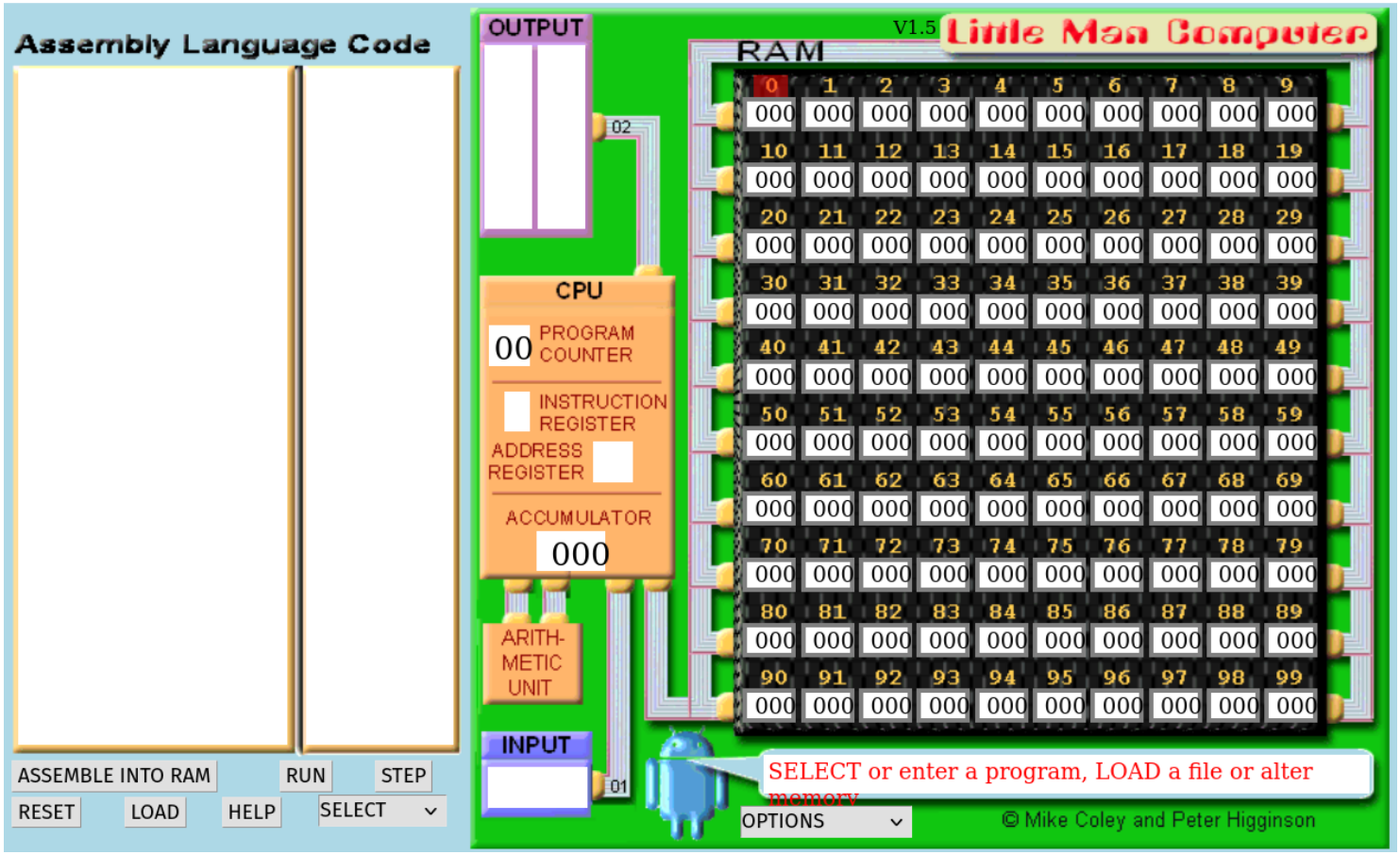
Recap (Friday)



Recap

- Introduction to the course and microcontrollers
- Number systems
  - Binary <-> Decimal <-> Hexadecimal
- **Micro-architecture**
  - Von Neumann and Harvard architecture
  - Key components
- **Little Man Computer** (assignment)
  - Fetch, decode, and execute cycle
  - (Solution will be available on ItsLearning)

  - **How was it?**



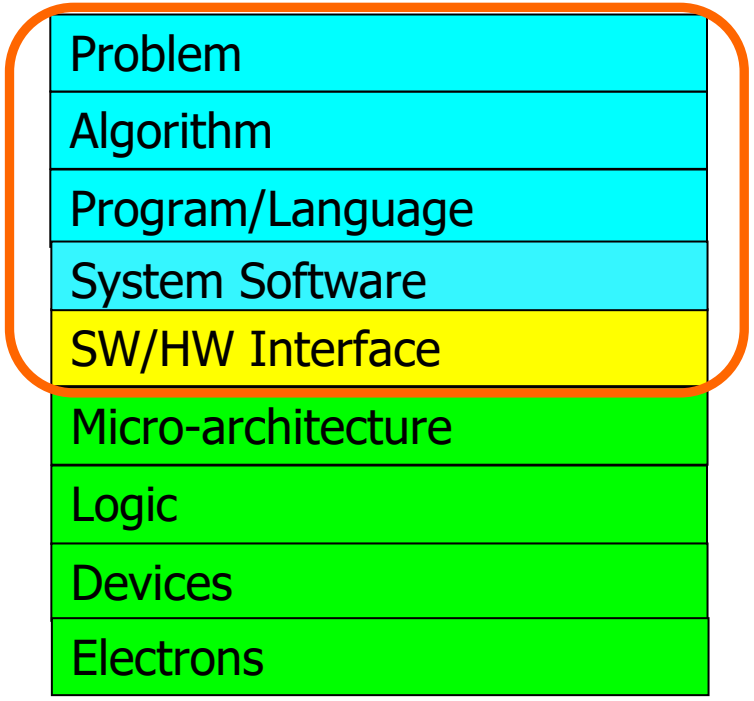
Programming

In-Class Q/A:
“What is programming?”

Programming (or **coding**)

- Definition[1]:

*"The composition of **sequences of instructions**, called programs, that computers can follow to **perform tasks**."*

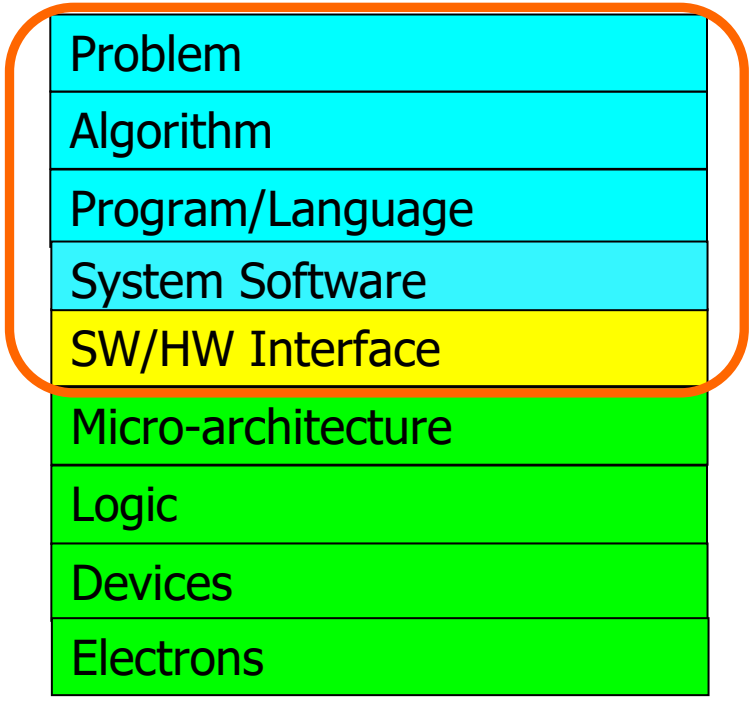


[1] https://en.wikipedia.org/wiki/Computer_programming

Programming (or **coding**)

- Definition[1]:

*"The composition of **sequences of instructions**, called programs, that ~~computers~~ microcontrollers can follow to **perform tasks**."*



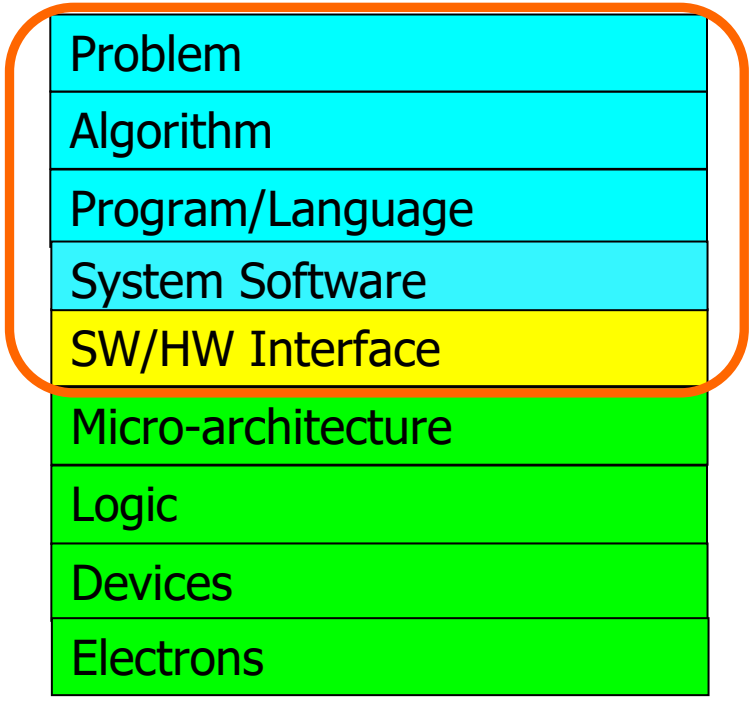
[1] https://en.wikipedia.org/wiki/Computer_programming

Programming (or **coding**)

- Definition[1]:

*"The composition of **sequences of instructions**, called programs, that ~~computers~~ microcontrollers can follow to **perform tasks**."*

- Best practice involves:
    1. **Problem Definition:** Understanding and defining the problem that needs to be solved.



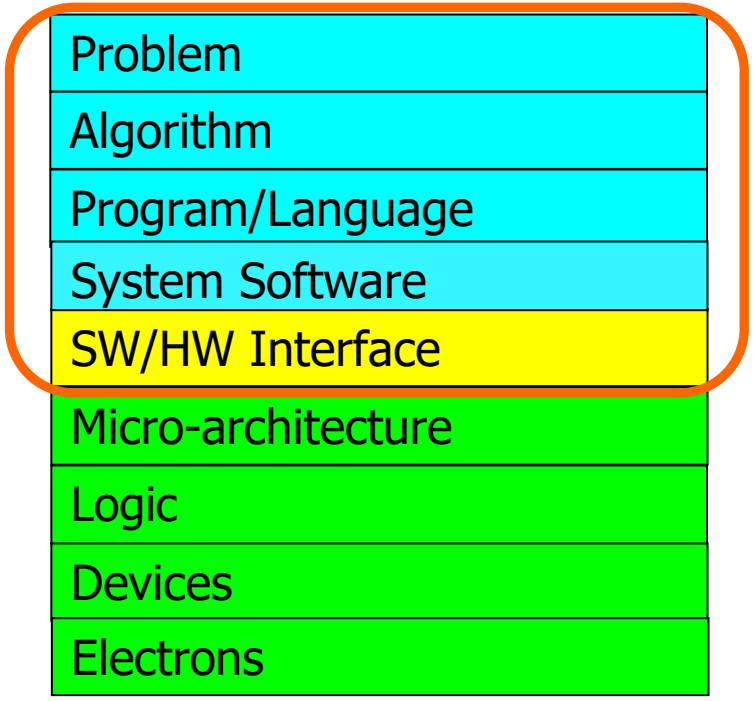
[1] https://en.wikipedia.org/wiki/Computer_programming

Programming (or **coding**)

- Definition[1]:

*"The composition of **sequences of instructions**, called programs, that ~~computers~~ microcontrollers can follow to **perform tasks**."*

- Best practice involves:
  1. **Problem Definition:** Understanding and defining the problem that needs to be solved.
  2. **Algorithm Design:** Creating a step-by-step procedure or set of rules to solve the problem, eg. using
     a. **Pseudocode:** Express an algorithm or process in plain language, without the strict syntax of a programming language.
     b. **Diagrams:** Flowcharts, state diagrams, etc.



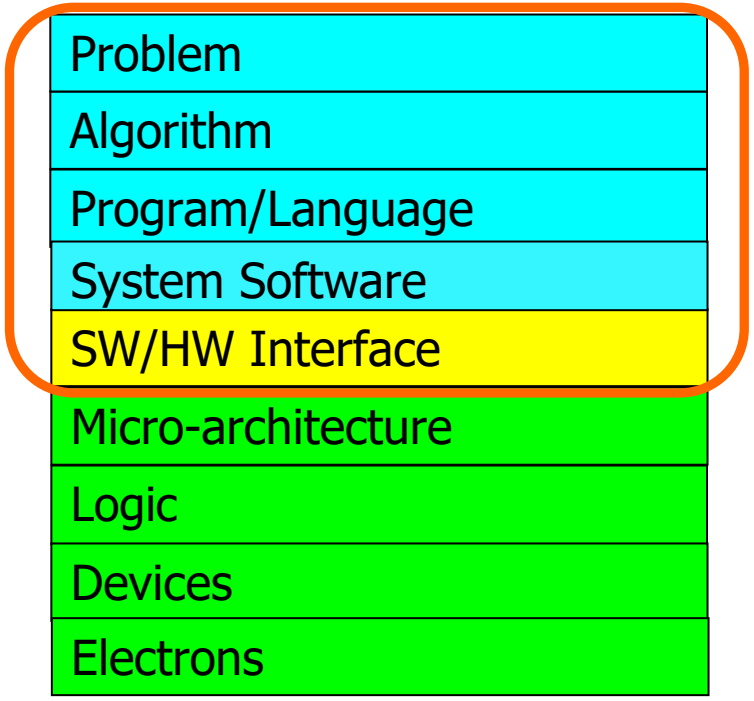
[1] https://en.wikipedia.org/wiki/Computer_programming

Programming (or **coding**)

- Definition[1]:

*"The composition of **sequences of instructions**, called programs, that ~~computers~~ microcontrollers can follow to **perform tasks**."*

- Best practice involves:
  1. **Problem Definition:** Understanding and defining the problem that needs to be solved.
  2. **Algorithm Design:** Creating a step-by-step procedure or set of rules to solve the problem.
  3. **Programming:** Writing the algorithm in a programming language.



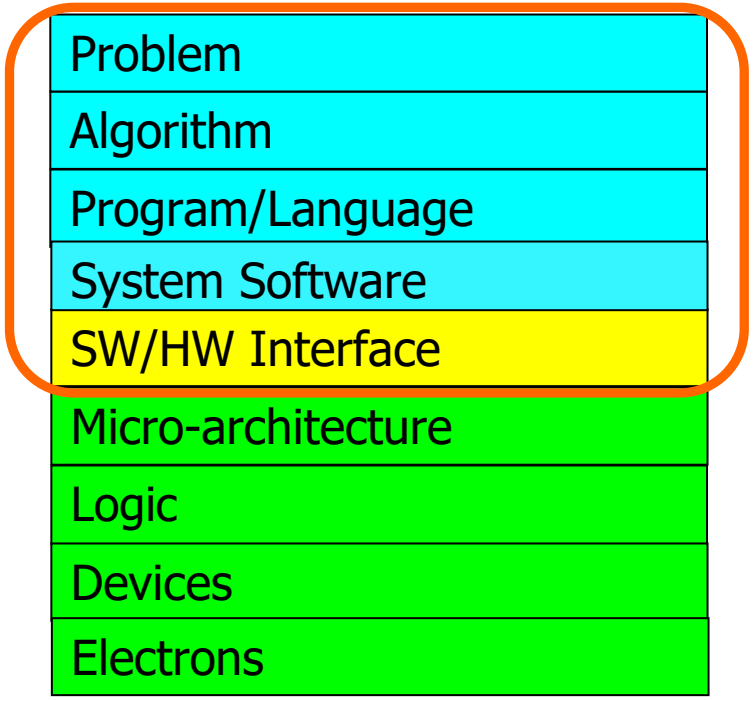
[1] https://en.wikipedia.org/wiki/Computer_programming

Programming (or **coding**)

- Definition[1]:

*"The composition of **sequences of instructions**, called programs, that ~~computers~~ microcontrollers can follow to **perform tasks**."*

- Best practice involves:
  1. **Problem Definition:** Understanding and defining the problem that needs to be solved.
  2. **Algorithm Design:** Creating a step-by-step procedure or set of rules to solve the problem.
  3. **Programming:** Writing the algorithm in a programming language.
  4. **Testing:** Running the code to check for and fix any errors or bugs.



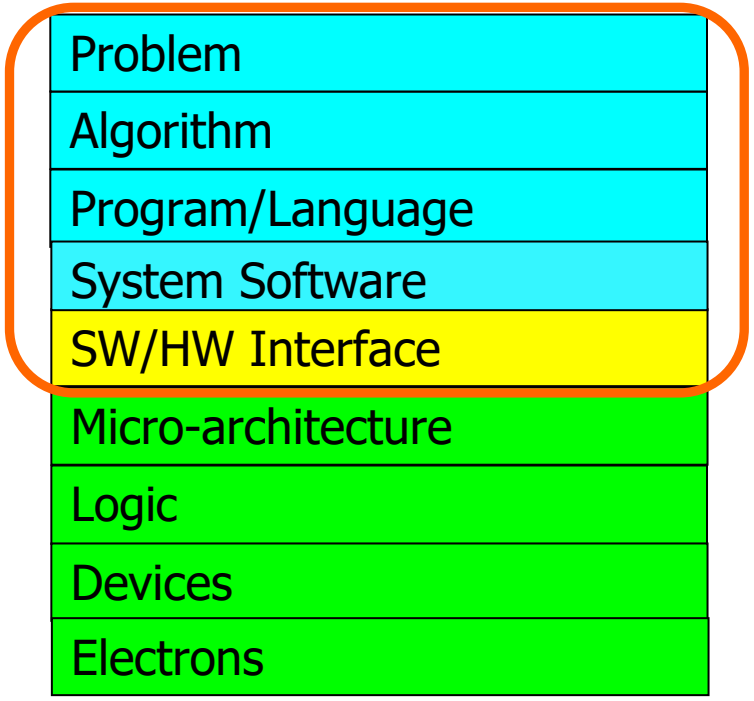
[1] https://en.wikipedia.org/wiki/Computer_programming

Programming (or **coding**)

- Definition[1]:

*"The composition of **sequences of instructions**, called programs, that ~~computers~~ microcontrollers can follow to **perform tasks**."*

- Best practice involves:
  1. **Problem Definition:** Understanding and defining the problem that needs to be solved.
  2. **Algorithm Design:** Creating a step-by-step procedure or set of rules to solve the problem.
  3. **Programming:** Writing the algorithm in a programming language.
  4. **Testing:** Running the code to check for and fix any errors or bugs.
  5. **Debugging:** Identifying and correcting mistakes in the code.



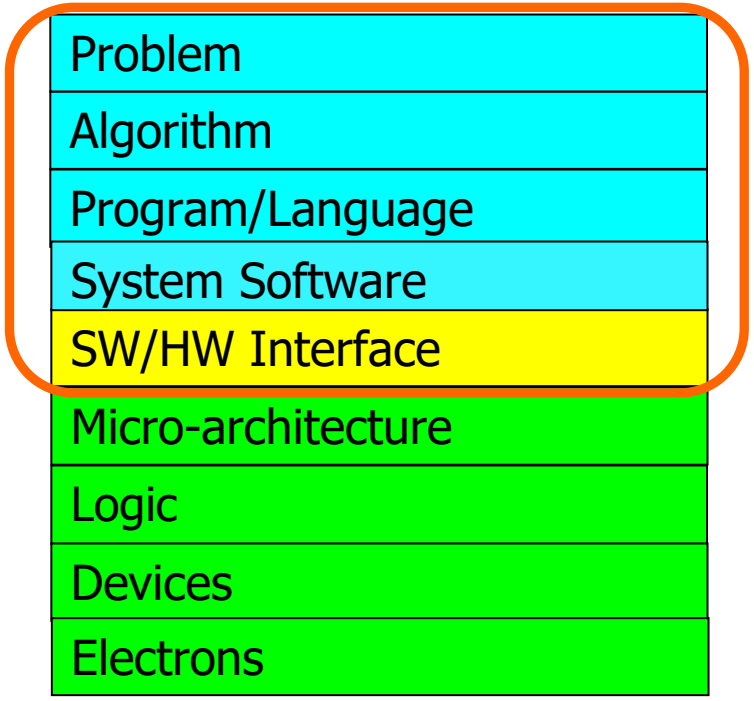
[1] https://en.wikipedia.org/wiki/Computer_programming

Programming (or **coding**)

- Definition[1]:

*"The composition of **sequences of instructions**, called programs, that ~~computers~~ microcontrollers can follow to **perform tasks**."*

- Best practice involves:
  1. **Problem Definition:** Understanding and defining the problem that needs to be solved.
  2. **Algorithm Design:** Creating a step-by-step procedure or set of rules to solve the problem.
  3. **Programming:** Writing the algorithm in a programming language.
  4. **Testing:** Running the code to check for and fix any errors or bugs.
  5. **Debugging:** Identifying and correcting mistakes in the code.
  6. **Maintenance:** Updating and modifying the code to improve performance or adapt to new requirements.



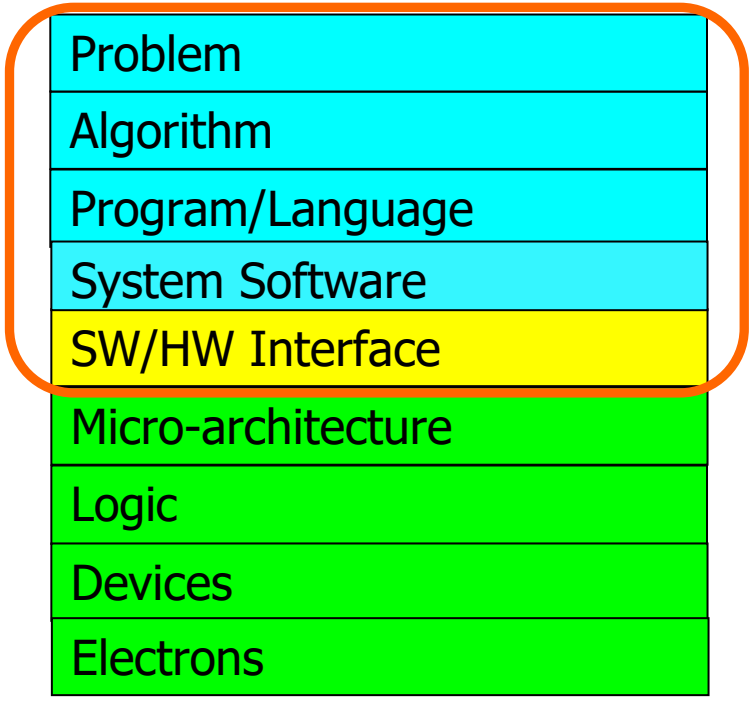
[1] https://en.wikipedia.org/wiki/Computer_programming

Programming (or **coding**)

- Definition[1]:

*"The composition of **sequences of instructions**, called programs, that ~~computers~~ microcontrollers can follow to **perform tasks**."*

- Best practice involves:
  1. **Problem Definition:** Understanding and defining the problem that needs to be solved.
  2. **Algorithm Design:** Creating a step-by-step procedure or set of rules to solve the problem.
  3. **Programming:** Writing the algorithm in a programming language.
  4. **Testing:** Running the code to check for and fix any errors or bugs.
  5. **Debugging:** Identifying and correcting mistakes in the code.
  6. **Maintenance:** Updating and modifying the code to improve performance or adapt to new requirements.
     a. Repeat the process… **(iterative process)**



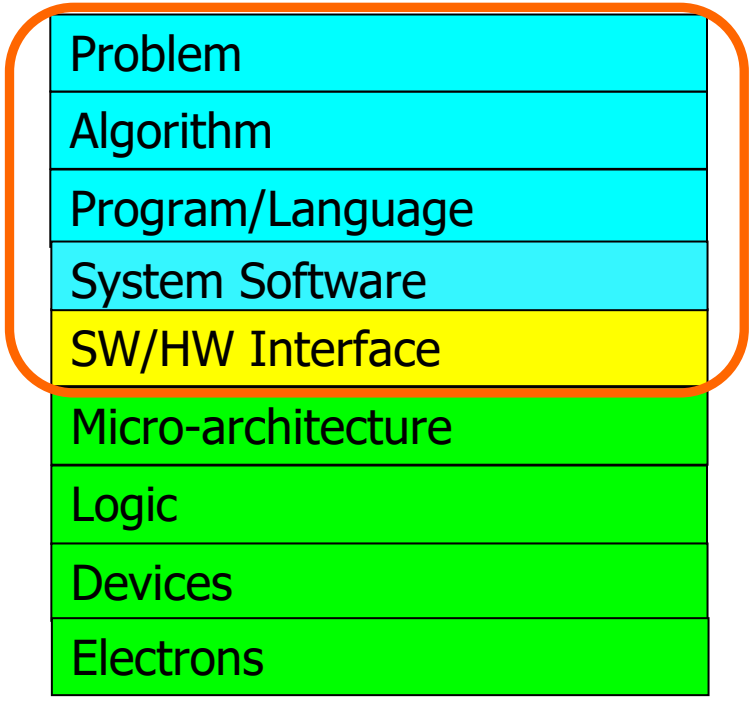
[1] https://en.wikipedia.org/wiki/Computer_programming

Programming (or **coding**)

- Definition[1]:

*"The composition of **sequences of instructions**, called programs, that ~~computers~~ microcontrollers can follow to **perform tasks**."*

- (Best) Practice involves:
  1. **practice, practice, practice, etc.**
  2. **…and more practice…**



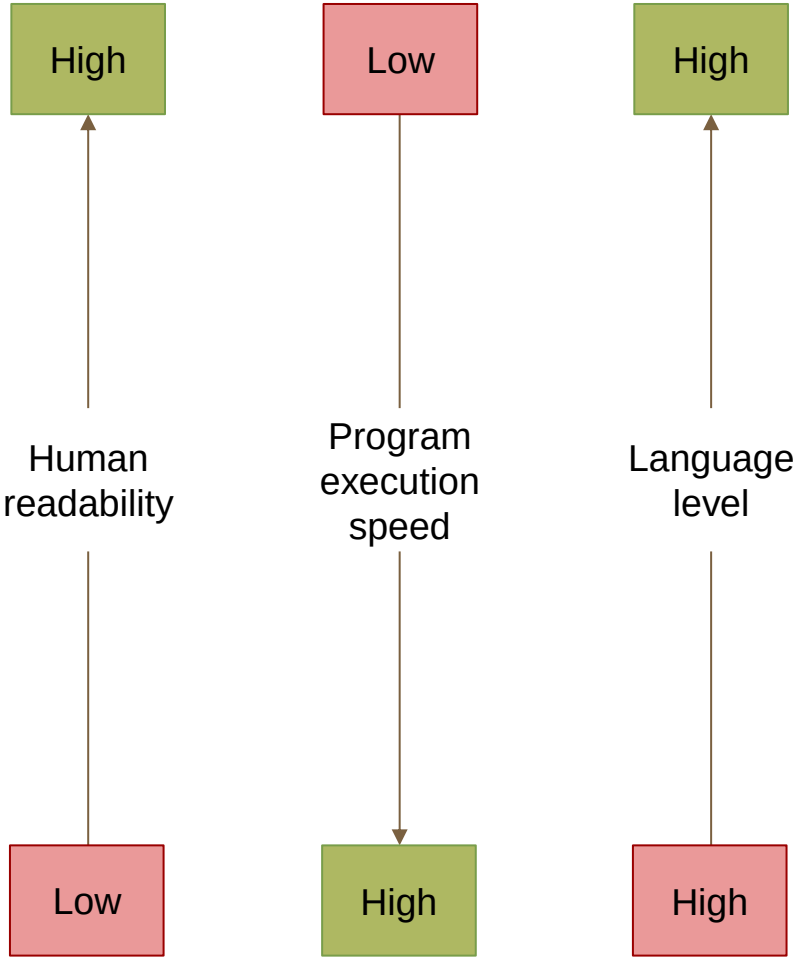
[1] https://en.wikipedia.org/wiki/Computer_programming

Levels of Programming

In-Class Q/A:
"What is your definition/understanding of high-level *and low-level programming?"*
Mid-level programming?

Levels of Programming languages

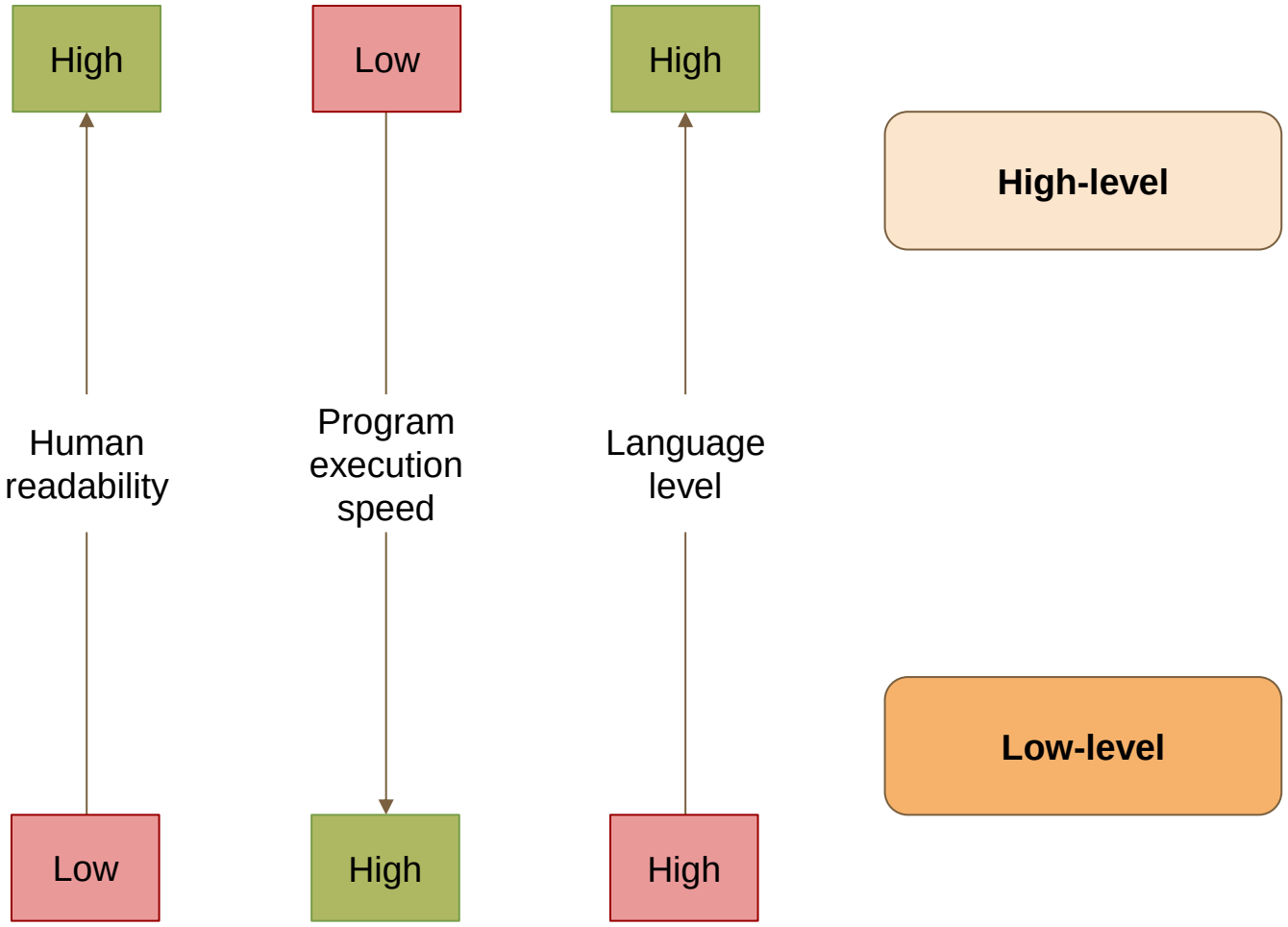
- **High-level**
  - Easy to read/write and learn
  - (Typically) Lower performance due to overhead
  - Limited control over hardware
  - Good for prototyping and quick development
  - Advanced tools (eg. error handling, data management)



Levels of Programming languages

- **High-level**
  - Easy to read/write and learn
  - (Typically) Lower performance due to overhead
  - Limited control over hardware
  - Good for prototyping and quick development
  - Advanced tools (eg. error handling, data management)

- **Low-level**
  - Difficult to read/write and learn
  - High performance due to direct control
  - Full control over hardware
  - Time-consuming to develop and maintain
  - Minimal error handling…

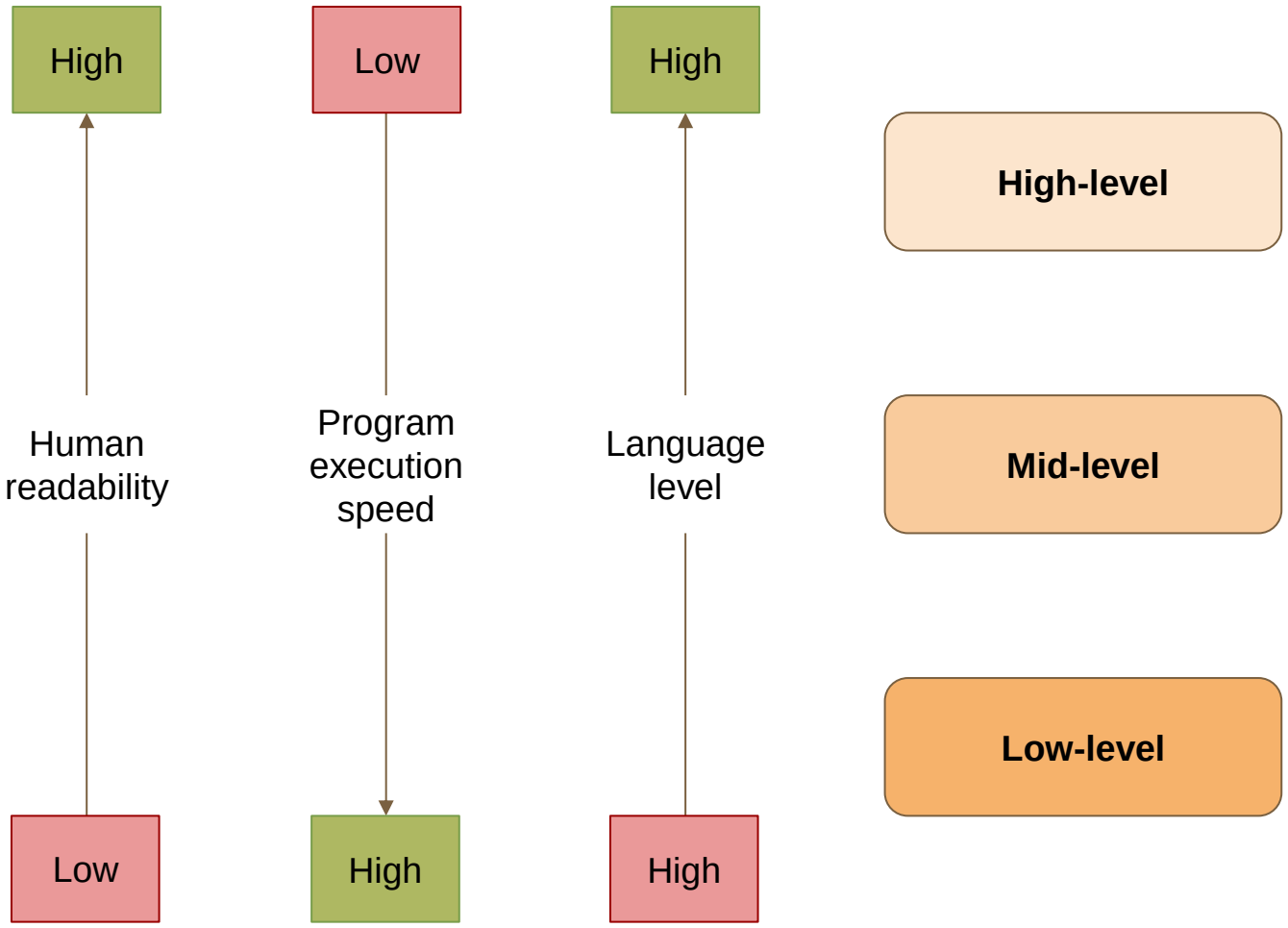


Levels of Programming languages

- **High-level**
  - Easy to read/write and learn
  - (Typically) Lower performance due to overhead
  - Limited control over hardware
  - Good for prototyping and quick development
  - Advanced tools (eg. error handling, data management)

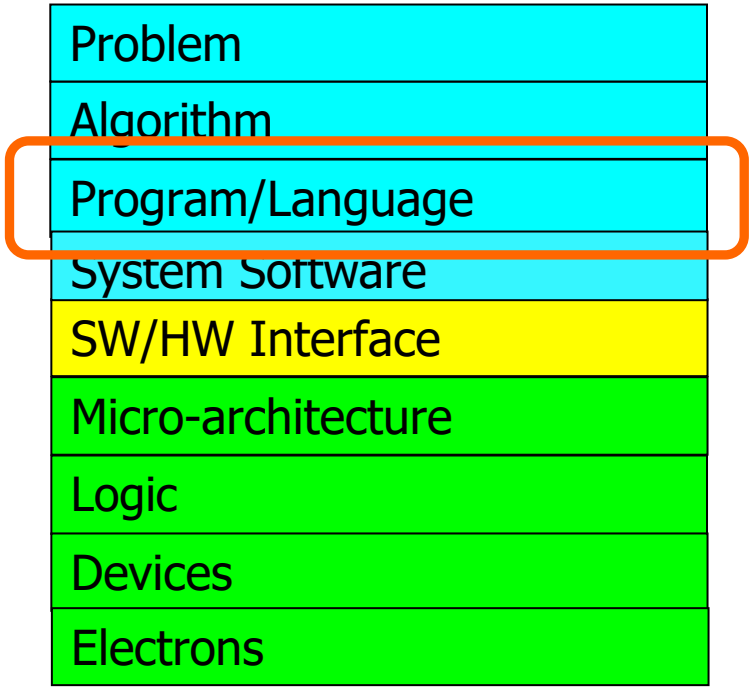
- **Mid-level**
  - A balance between high-level ease of use and low-level control over hardware

- **Low-level**
  - Difficult to read/write and learn
  - High performance due to direct control
  - Full control over hardware
  - Time-consuming to develop and maintain
  - Minimal error handling



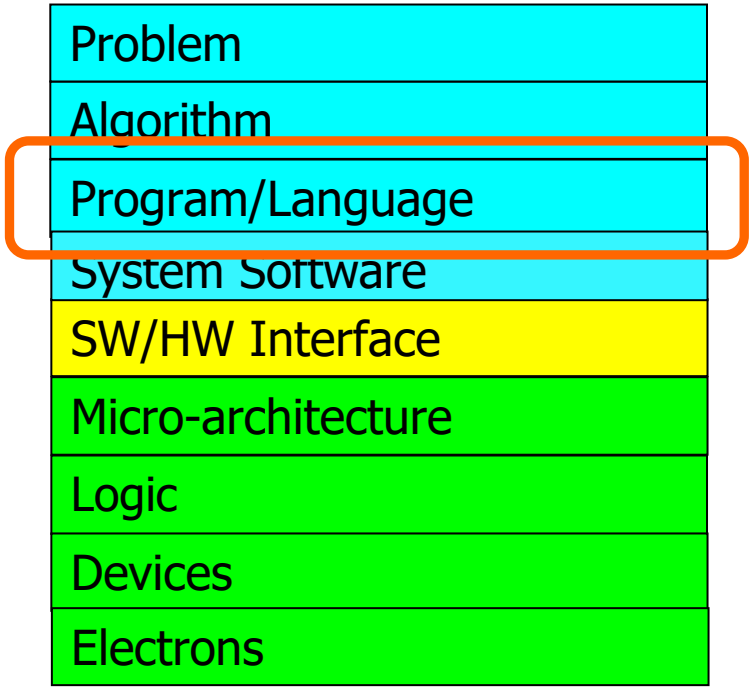
Levels of Programming languages

- **High-level**
  - JavaScript, Python, Ruby, etc.
- **Mid-level**
  - C, C++, Rust, etc.
- **Low-level**
  - Assembly Language, Machine code



Levels of Programming languages

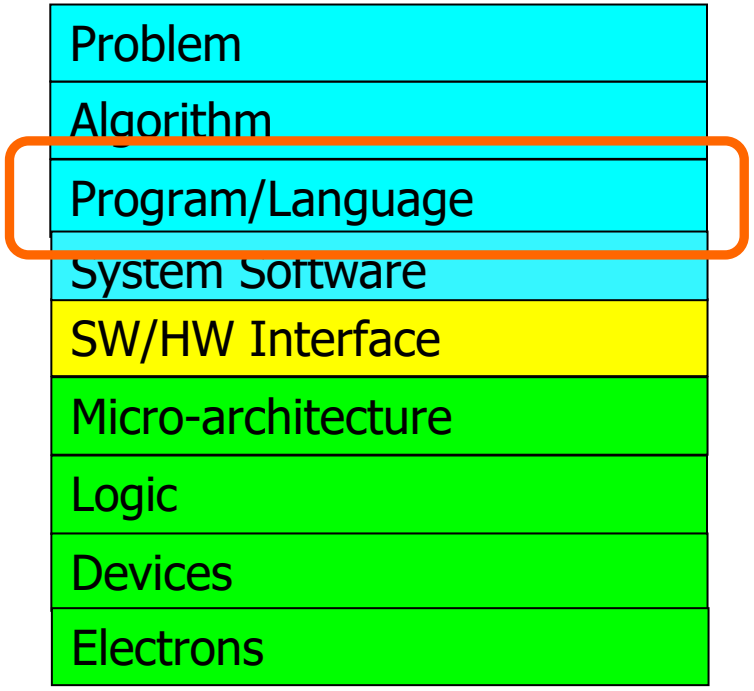
- **High-level**
  - JavaScript, Python, Ruby, etc.
- **Mid-level**
  - C, C++, Rust, etc.
- **Low-level**
  - Assembly Language, Machine code



Definition changes over time?
Or from person-to-person?

Levels of Programming languages

- **High-level? Meta-level?**
  - CoPilot? ChatGBT?
- **Mid-level? High-level?**
  - JavaScript, Python, Ruby?
- **Low-level? Mid-level??**
  - C, C++, Rust?
- **VERY Low-level? Low-level?**
  - Assembly Language, Machine code?



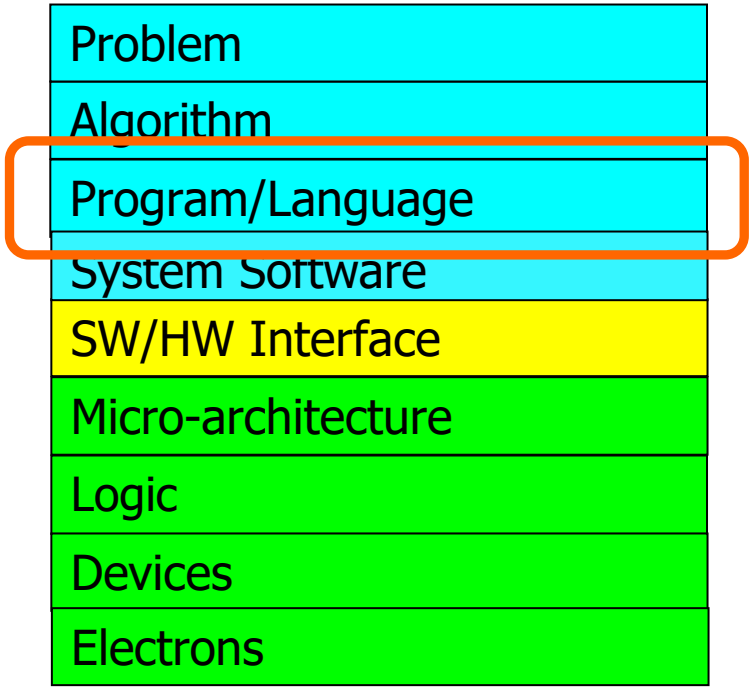
Definition changes over time?
Or from person-to-person?

Difference between programming languages?

Interpreted and **Compiled** languages?

- **Compiled Languages:** The code/program is translated from the source code into machine code (binary code) by a compiler before execution (executable file).

- **Interpreted Languages:** The code/program is executed line-by-line using an interpreter.



Interpreted and **Compiled** languages?

- **Compiled Languages:** The code/program is translated from the source code into machine code (binary code) by a compiler before execution (executable file).
  - = need to compile the entire program before running it.
    - Fast execution (the translation step is already done before run-time)
    - (Most) **Errors are detected** by the compiler before
      - Not runtime errors (eg. dividing by zero)
  - **Example languages:** C, C++, Rust, Java, etc.

- **Interpreted Languages:** Code is translated and executed line-by-line using an interpreter at runtime.

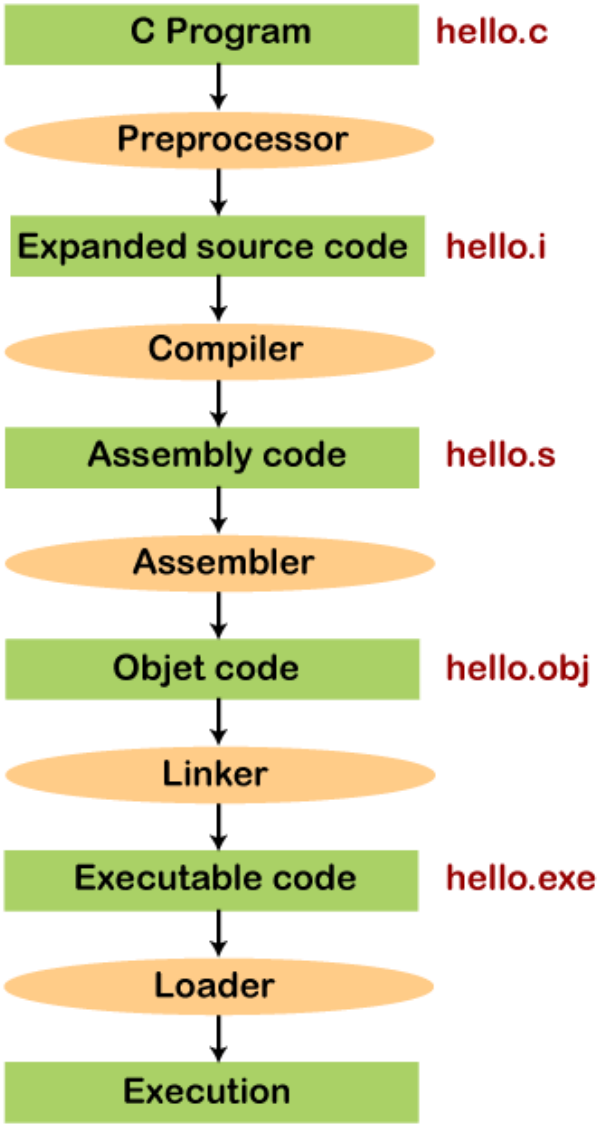
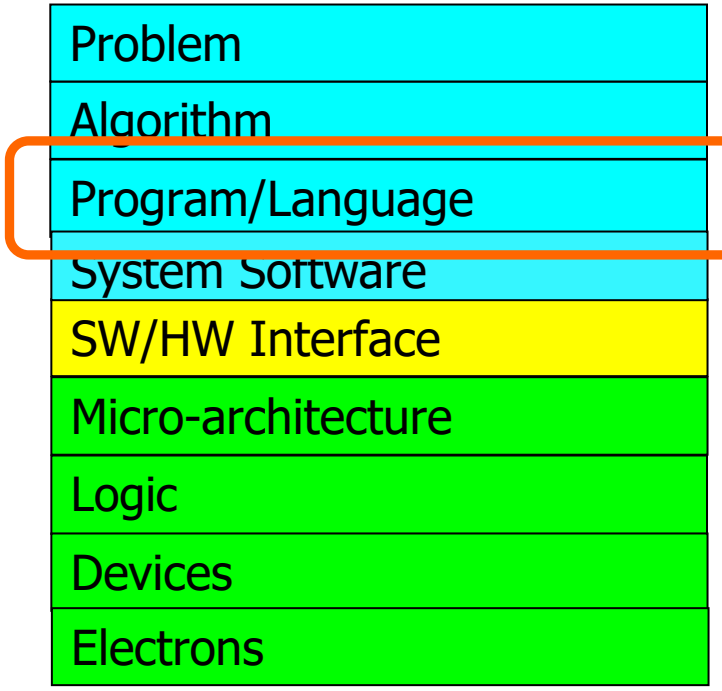


Illustration: https://medium.com/@elodieriou/the-compilation-process-in-c-b4325a4dd953

Interpreted and **Compiled** languages?

- **Compiled Languages:** The code/program is translated from the source code into machine code (binary code) by a compiler before execution (executable file).

- **Interpreted Languages:** Code is translated and executed line-by-line using an interpreter at runtime.
  - = no need to compile the entire program before running it
    - = fast prototyping!
    - ***"Move fast, break things"***
      - Errors are detected at runtime… **Good? Bad?**
  - Generally slower, since code is translated during execution
  - **Example languages:** Python, JavaScript, Ruby, etc.



Programming Language Syntax

*"...the **rules** that define the combinations of symbols that are considered to be correctly **structured statements or expressions** in that language."*

- **…so how you**
  - write commands,
  - declarations,
  - expressions,
  - structure of the source code,
  - etc.

Different programming language
=
Different syntax

Illustration: https://garbagevalue.com/definitions/syntax-programming-language

Programming Language Syntax (Examples)

Examples: Hello World		
High-level	Mid-level	Low-level
Python	**C++**	**Assembly**

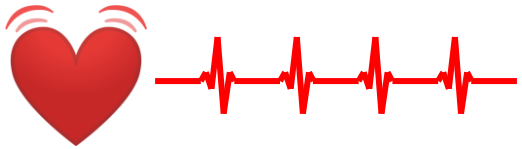
```python
print("Hello, World!") # Output string
```

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;   // Outp
    return 0;
}
```

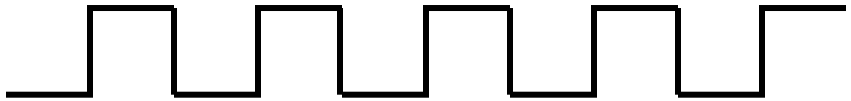
```assembly
section .data
    hello db 'Hello, World!', 0x0A  ; The string to be printed

section .text
    global _start

_start:
    ; Output string to stdout
    mov eax, 4          ; syscall number for sys_write
    mov ebx, 1          ; file descriptor 1 is stdout
    mov ecx, hello      ; pointer to the string
    mov edx, 13         ; length of the string
    int 0x80            ; call kernel

    ; Exit the program
    mov eax, 1          ; syscall number for sys_exit
    xor ebx, ebx        ; return 0 status
    int 0x80            ; call kernel
```

Sequential execution (for most languages)

→ Pipeline code execution (Module 1)



CPU Clock

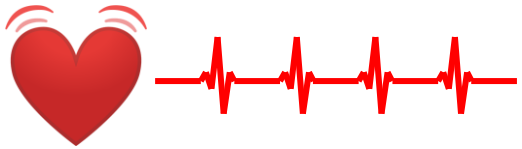


Program/Code Memory

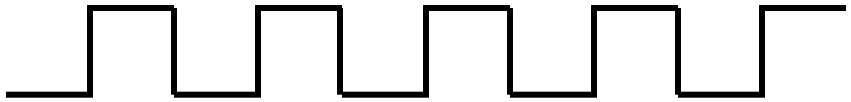
	Address	Memory	Fetch / Decode	Execution
PC = 0	0	Instruction 0	Instruction 0	
PC = 1	1	Instruction 1	Instruction 1	Instruction 0
PC = 2	2	Instruction 2	Instruction 2	Instruction 1
PC = 3	3	Instruction 3	Instruction 3	Instruction 2
PC = 4	4	Instruction 4	Instruction 4	Instruction 3
	5	Instruction 5		
	6	Instruction 6		

Illustration: Datateknik course, W1 (T540050101, by Ali Sahafi)

Sequential execution (for most languages)
→ Pipeline code execution (Module 1)



CPU Clock



(Typically) **The same for the program execution**

- Each statement is executed one at the time
  - Starting from the top of the source code
  - …and then line-by-line down to the bottom…

- Next statement does not begin before the previous

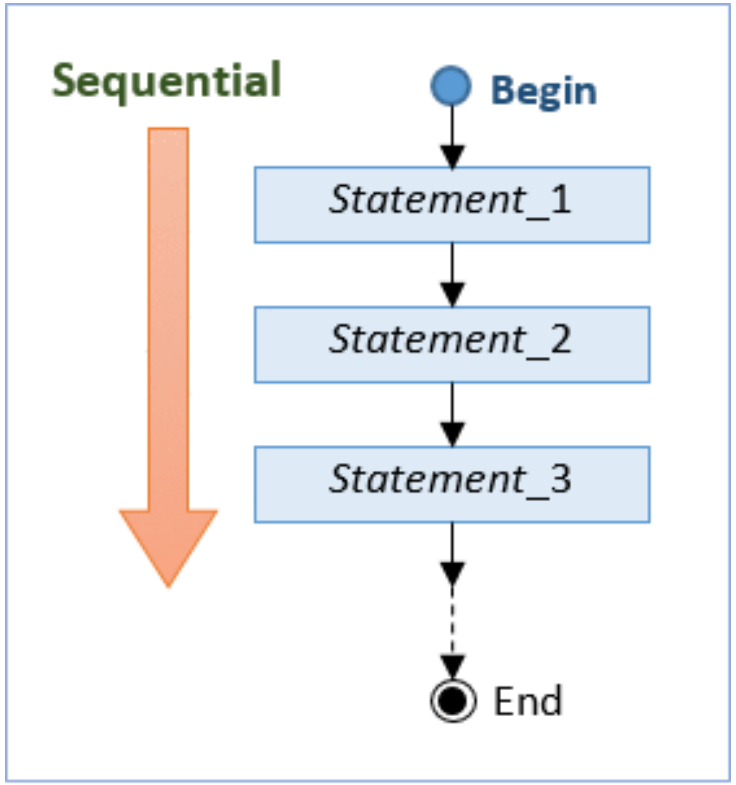
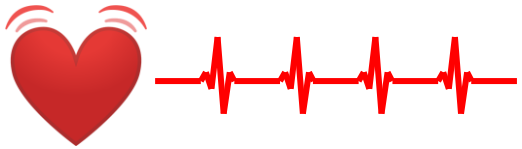
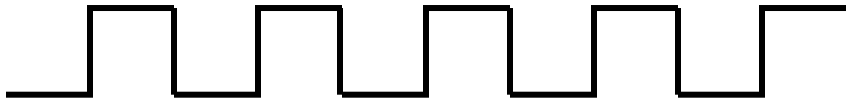


Illustration: https://www3.ntu.edu.sg/home/ehchua/Programming/java/J1a_Introduction.html

Sequential execution (for most languages)
→ Pipeline code execution (Module 1)



CPU Clock



(Typically) **The same for the program execution**

- Each statement is executed one at the time
  - Starting from the top of the source code
  - …and then line-by-line down to the bottom…
- Next statement does not begin before the previous
- Since the instructions are followed in order…
  - … it's easy to predict the output based on the input
  - …and debug, if there is an error.

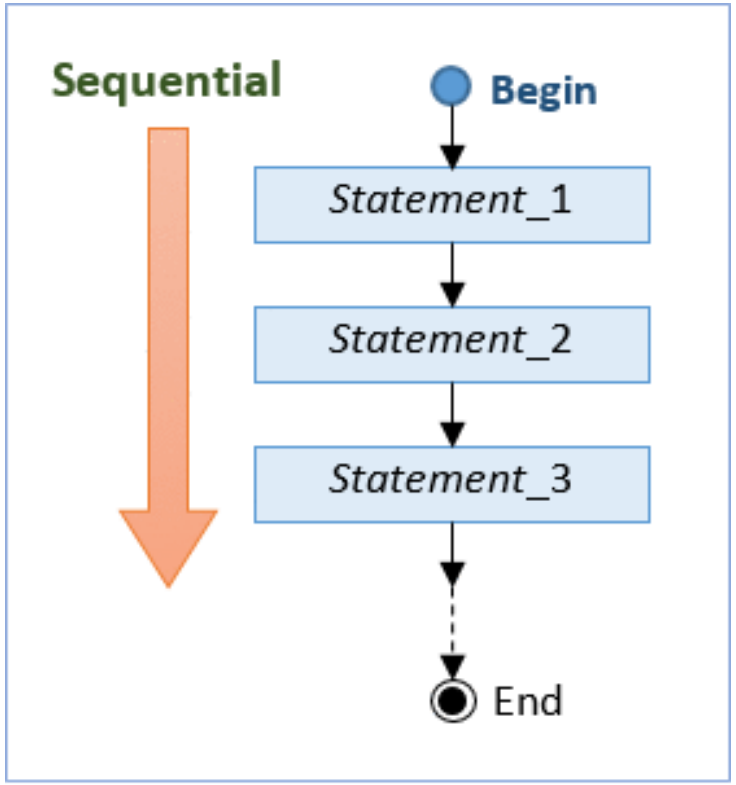
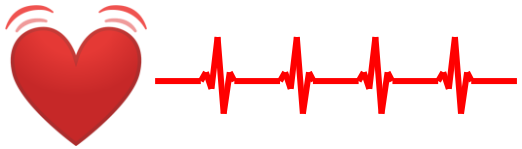
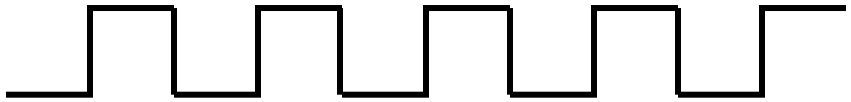


Illustration: https://www3.ntu.edu.sg/home/ehchua/Programming/java/J1a_Introduction.html

Sequential execution (for most languages)
→ Pipeline code execution (Module 1)



CPU Clock



(Typically) **The same for the program execution**

- Each statement is executed one at the time
  - Starting from the top of the source code
  - …and then line-by-line down to the bottom…
- Next statement does not begin before the previous
- Since the instructions are followed in order…
  - … it's easy to predict the output based on the input
  - …and debug, if there is an error.
- **…and not entirely true…**
  - Control flows / branching is a exception (more about that later…)

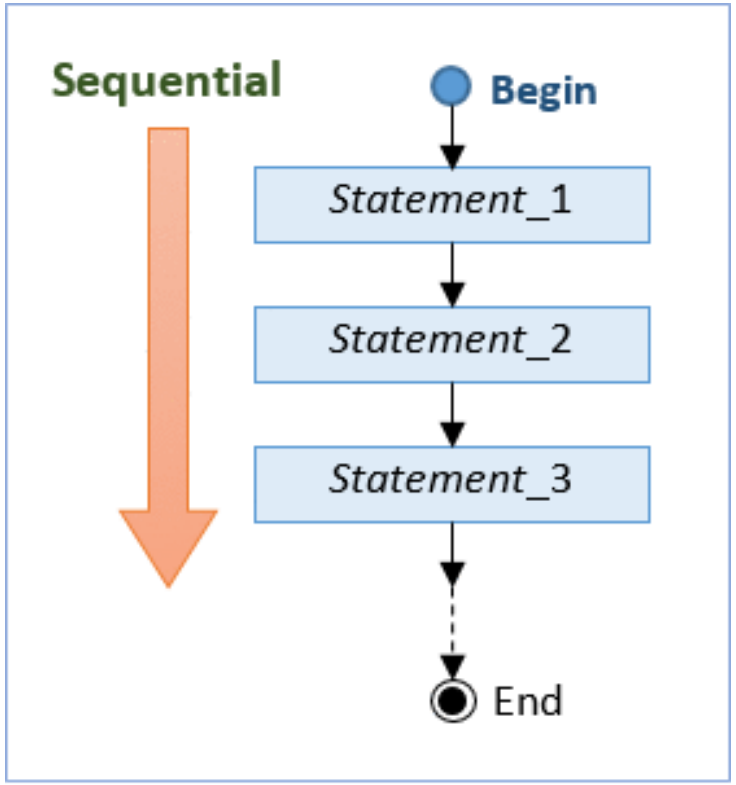


Illustration: https://www3.ntu.edu.sg/home/ehchua/Programming/java/J1a_Introduction.html

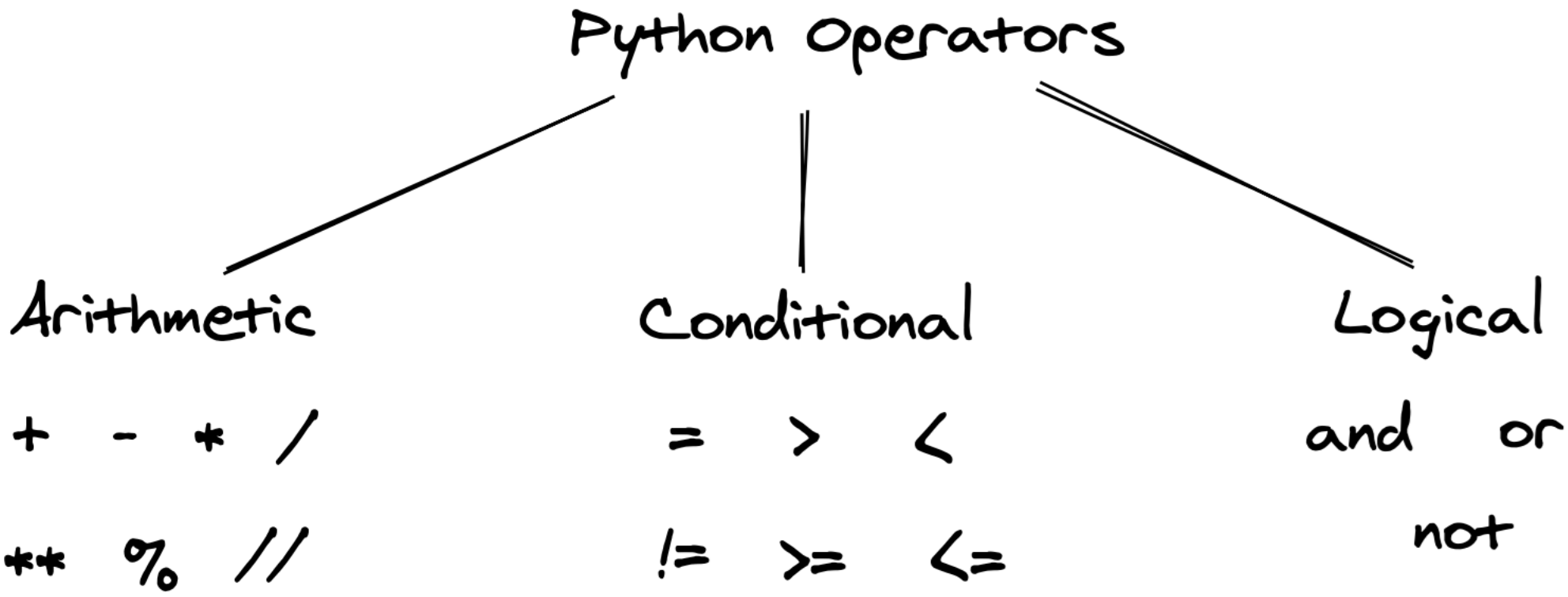


Illustration: https://jovian.ai/learn/data-analysis-with-python-zero-to-pandas

Arithmetic Operators

- Used with numeric values to perform **basic mathematical operations**

Operator	Purpose	Example	Result
+	**Addition**	2 + 3	5
-	**Subtraction**	3 - 2	1
*	**Multiplication**	8 * 12	96
/	**Division**	100 / 7	14.28571…
//	**Floor Division**	100 // 7	14
%	**Modulus/Remainder**	100 % 7	2
**	**Exponent**	5 ** 3	125

Conditional Operators

- Used comparing numbers and variables (returns either **True** or **False**)
  - Commonly in **conditional statements** and **loops**.

Operator	Purpose	Example	Result
==	**Check if operands are equal**	2 == 3	False
!=	**Check if operands are not equal**	3 != 2	True

Conditional Operators

- Used comparing numbers and variables (returns either **True** or **False**)
  - Commonly in **conditional statements** and **loops**.

Operator	Purpose	Example	Result
==	**Check if operands are equal**	2 == 3	False
!=	**Check if operands are not equal**	3 != 2	True
>	**Check if left operand is greater than right operand**	8 > 12	False
<	**Check if left operand is less than right operand**	8 < 12	True

Conditional Operators

- Used comparing numbers and variables (returns either **True** or **False**)
  - Commonly in **conditional statements** and **loops**.

Operator	Purpose	Example	Result
==	**Check if operands are equal**	2 == 3	False
!=	**Check if operands are not equal**	3 != 2	True
>	**Check if left operand is greater than right operand**	8 > 12	False
<	**Check if left operand is less than right operand**	8 < 12	True
>=	**Check if left operand is greater than or equal to right operand**	8 >= 8	True
<=	**Check if left operand is less than or equal to right operand**	8 <= 8	True

Logical Operators

- Used for combining conditions with logical operators (returns either True or False)
  - Commonly in conditional statements and loops.

a	b	a **and** b	a **or** b	not a	not b
True	**True**	True	True	False	False
True	**False**	False	True	False	True
False	**True**	False	True	True	False
False	**False**	False	False	True	True

- **And:** Returns True if both operands are True
- **Or:** Returns True if either of the operands are True
- **Not:** Returns True if the operand is False

Logical Operators

- Used for combining conditions with logical operators (returns either True or False)
  - Commonly in conditional statements and loops and **combined.**

a	b	a **and** b	a **or** b	not a **and** b	not (a **and** b)
True	**True**	True	True	False	False
True	**False**	False	True	False	True
False	**True**	False	True	True	True
False	**False**	False	False	False	True

- **And:** Returns True if both operands are True
- **Or:** Returns True if either of the operands are True
- **Not:** Returns True if the operand is False

In-class assignment (10 min)

Combining conditions with logical operators

```
my_favorite_number = 3
```

```
1) my_favorite_number > 0 and my_favorite_number <= 3
2) my_favorite_number > 0 or my_favorite_number <= 3
3) my_favorite_number == 0 or not my_favorite_number <= 3
4) not my_favorite_number > 0 or not my_favorite_number <= 3
5) not (my_favorite_number > 0 or not my_favorite_number <= 3)
```

a	**b**	a **and** b	a **or** b	**not** a	**not** b	not a **and** b	not (a **and** b)
True	**True**	True	True	False	False	False	False
True	**False**	False	True	False	True	False	True
False	**True**	False	True	True	False	True	True
False	**False**	False	False	True	True	False	True

Hand up (True) | Hand down (False)

```
my_favorite_number = 3
```

1) `my_favorite_number > 0 and my_favorite_number <= 3`
   a) ?
2) `my_favorite_number > 0 or my_favorite_number <= 3`
   a) ?
3) `my_favorite_number == 0 or not my_favorite_number <= 3`
   a) ?
4) `not my_favorite_number > 0 or not my_favorite_number <= 3`
   a) ?
5) `not (my_favorite_number > 0 or not my_favorite_number <= 3)`
   a) ?

Hand up (True) | Hand down (False)

```
my_favorite_number = 3
```

1) `my_favorite_number > 0 and my_favorite_number <= 3`
   a) `True and True = **True**`
2) `my_favorite_number > 0 or my_favorite_number <= 3`
   a) `?`
3) `my_favorite_number == 0 or not my_favorite_number <= 3`
   a) `?`
4) `not my_favorite_number > 0 or not my_favorite_number <= 3`
   a) `?`
5) `not (my_favorite_number > 0 or not my_favorite_number <= 3)`
   a) `?`

Hand up (True) | Hand down (False)

```
my_favorite_number = 3
```

1) `my_favorite_number > 0 and my_favorite_number <= 3`
   a) `True and True = True`
2) `my_favorite_number > 0 or my_favorite_number <= 3`
   a) `True or True =` **`True`**
3) `my_favorite_number == 0 or not my_favorite_number <= 3`
   a) `?`
4) `not my_favorite_number > 0 or not my_favorite_number <= 3`
   a) `?`
5) `not (my_favorite_number > 0 or not my_favorite_number <= 3)`
   a) `?`

Hand up (True) | Hand down (False)

```
my_favorite_number = 3
```

1) `my_favorite_number > 0 and my_favorite_number <= 3`
   a) `True and True = True`
2) `my_favorite_number > 0 or my_favorite_number <= 3`
   a) `True or True = True`
3) `my_favorite_number == 0 or not my_favorite_number <= 3`
   a) `False or not True =` **`False`**
4) `not my_favorite_number > 0 or not my_favorite_number <= 3`
   a) `?`
5) `not (my_favorite_number > 0 or not my_favorite_number <= 3)`
   a) `?`

Hand up (True) | Hand down (False)

```
my_favorite_number = 3
```

1) `my_favorite_number > 0 and my_favorite_number <= 3`
   a) `True and True = True`
2) `my_favorite_number > 0 or my_favorite_number <= 3`
   a) `True or True = True`
3) `my_favorite_number == 0 or not my_favorite_number <= 3`
   a) `False or not True = False`
4) `not my_favorite_number > 0 or not my_favorite_number <= 3`
   a) `not True or not True =` **`False`**
5) `not (my_favorite_number > 0 or not my_favorite_number <= 3)`
   a) `?`

Hand up (True) | Hand down (False)

```
my_favorite_number = 3
```

1) `my_favorite_number > 0 and my_favorite_number <= 3`
   a) `True and True = True`
2) `my_favorite_number > 0 or my_favorite_number <= 3`
   a) `True or True = True`
3) `my_favorite_number == 0 or not my_favorite_number <= 3`
   a) `False or not True = False`
4) `not my_favorite_number > 0 or not my_favorite_number <= 3`
   a) `not True or not True = False`
5) `not (my_favorite_number > 0 or not my_favorite_number <= 3)`
   a) `not (True or not True) = not True =` **`False`**

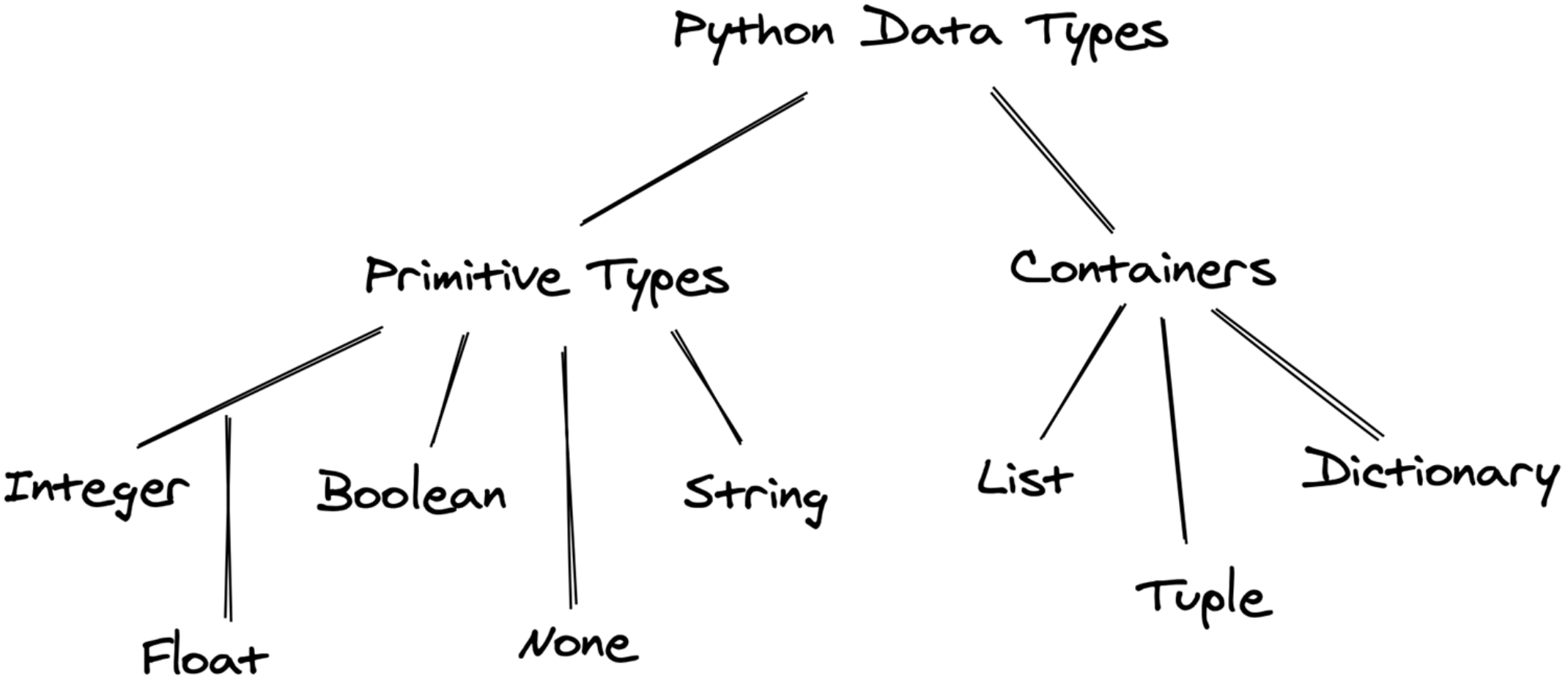


Illustration: https://jovian.ai/learn/data-analysis-with-python-zero-to-pandas

Data Type

"A classification that specifies the kind of value an object can have and what operations can be performed on that object."

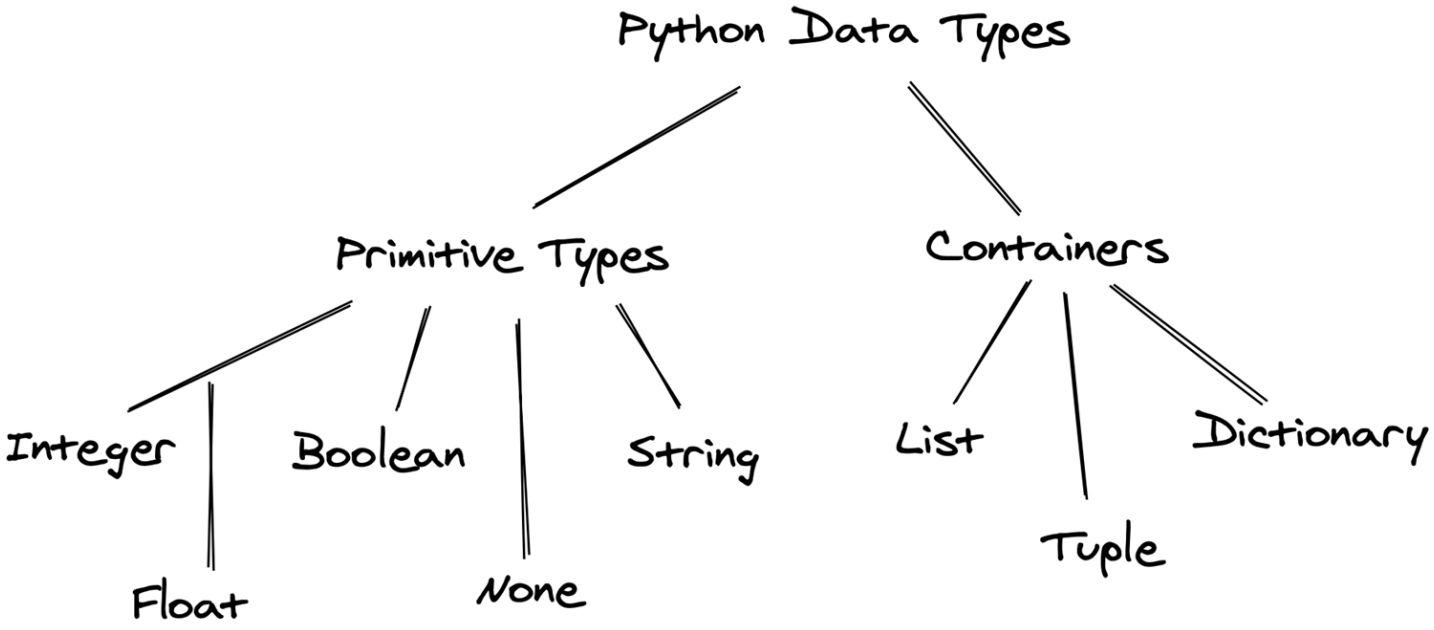


Illustration: https://jovian.ai/learn/data-analysis-with-python-zero-to-pandas

Data Type

"A classification that specifies the kind of value an object can have and what operations can be performed on that object."

- **Primitive Data Types:** the most basic data types and are usually built into the language.

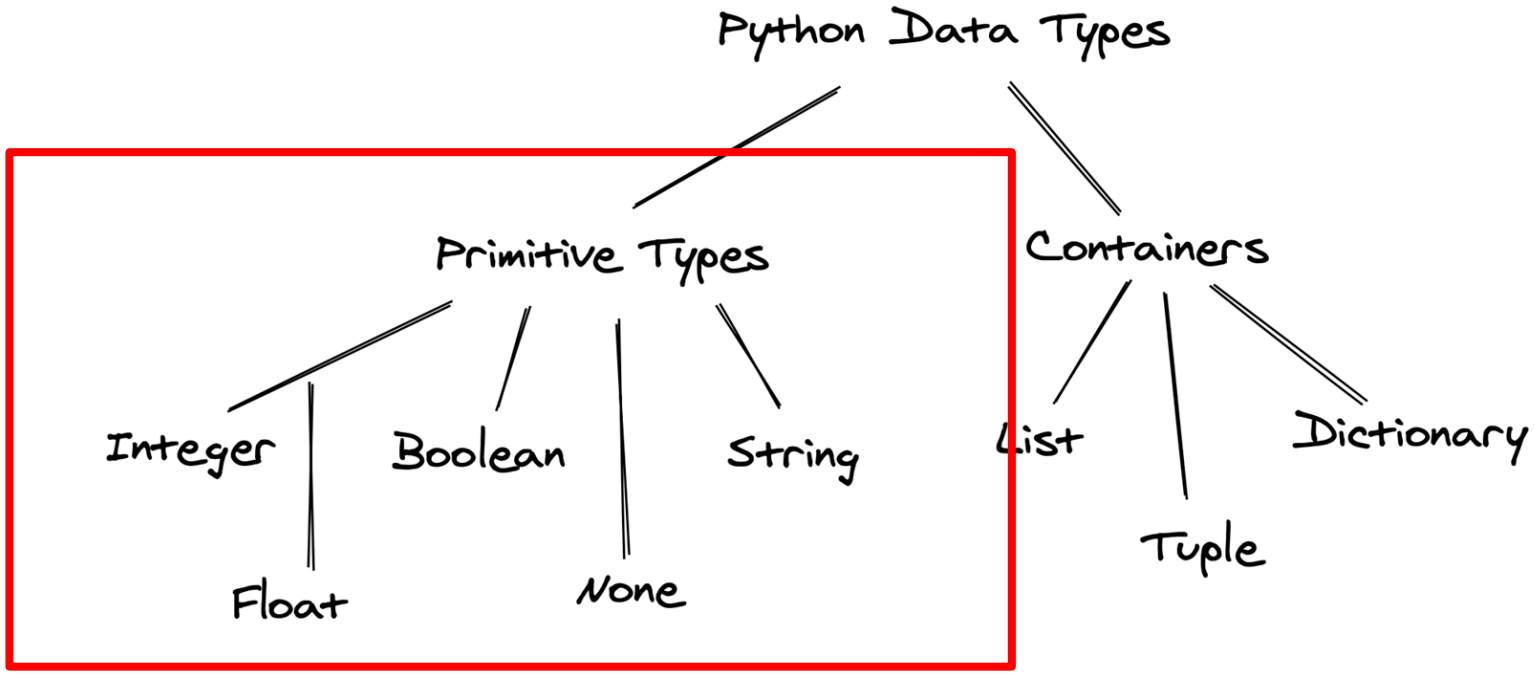


Illustration: https://jovian.ai/learn/data-analysis-with-python-zero-to-pandas

Data Type

"A classification that specifies the kind of value an object can have and what operations can be performed on that object."

- **Primitive Data Types:** the most basic data types and are usually built into the language.
  - **Integer (int):** Represents whole numbers without decimal points (e.g., 1, -3, 42).

Data Type

"A classification that specifies the kind of value an object can have and what operations can be performed on that object."

- **Primitive Data Types:** the most basic data types and are usually built into the language.
  - **Integer (int):** Represents whole numbers without decimal points (e.g., 1, -3, 42).
  - **Floating Point (float, double):** Represents numbers with decimal points (e.g., 3.14, -0.001, 2.0).

Data Type

"A classification that specifies the kind of value an object can have and what operations can be performed on that object."

- **Primitive Data Types:** the most basic data types and are usually built into the language.
  - **Integer (int):** Represents whole numbers without decimal points (e.g., 1, -3, 42).
  - **Floating Point (float, double):** Represents numbers with decimal points (e.g., 3.14, -0.001, 2.0).
  - **Character (char):** Represents a single character (e.g., 'a', 'Z', '5')

Data Type

"A classification that specifies the kind of value an object can have and what operations can be performed on that object."

- **Primitive Data Types:** the most basic data types and are usually built into the language.
  - **Integer (int):** Represents whole numbers without decimal points (e.g., 1, -3, 42).
  - **Floating Point (float, double):** Represents numbers with decimal points (e.g., 3.14, -0.001, 2.0).
  - **Character (char):** Represents a single character (e.g., 'a', 'Z', '5')
  - **Boolean (bool):** Represents true or false values.

Data Type

"A classification that specifies the kind of value an object can have and what operations can be performed on that object."

- **Primitive Data Types:** the most basic data types and are usually built into the language.
  - **Integer (int):** Represents whole numbers without decimal points (e.g., 1, -3, 42).
  - **Floating Point (float, double):** Represents numbers with decimal points (e.g., 3.14, -0.001, 2.0).
  - **Character (char):** Represents a single character (e.g., 'a', 'Z', '5')
  - **Boolean (bool):** Represents true or false values.
  - **Null (None):** Represents the absence of a value or a null type.

Data Type

"A classification that specifies the kind of value an object can have and what operations can be performed on that object."

- **Primitive Data Types:** the most basic data types and are usually built into the language.
  - **Integer (int):** Represents whole numbers without decimal points (e.g., 1, -3, 42).
  - **Floating Point (float, double):** Represents numbers with decimal points (e.g., 3.14, -0.001, 2.0).
  - **Character (char):** Represents a single character (e.g., 'a', 'Z', '5')
  - **Boolean (bool):** Represents true or false values.
  - **Null (None):** Represents the absence of a value or a null type.
  - **(String (str):** Represents a sequence of characters.)

Data Type

"A classification that specifies the kind of value an object can have and what operations can be performed on that object."

- **Examples** (for different languages)

Data Type	Python	Java	C++	Size
Integer	int	byte (8-bit), short (16-bit), int (32-bit), long (64-bit)	char (8-bit), short (16-bit), int (32-bit), long (64-bit)	**8-bit to 64-bit**
Float	float	float (32-bit), double (64-bit)	float (32-bit), double (64-bit)	**32-bit (single-precision), 64-bit (double-precision)**
Char	Str	char	char	**8-bit**
String	str	String	std::string	**Variable size (depends on length)**
Boolean	bool	boolean	bool	**1 byte (typically)**
None	None	null	nullptr	

Data Type

"A classification that specifies the kind of value an object can have and what operations can be performed on that object."

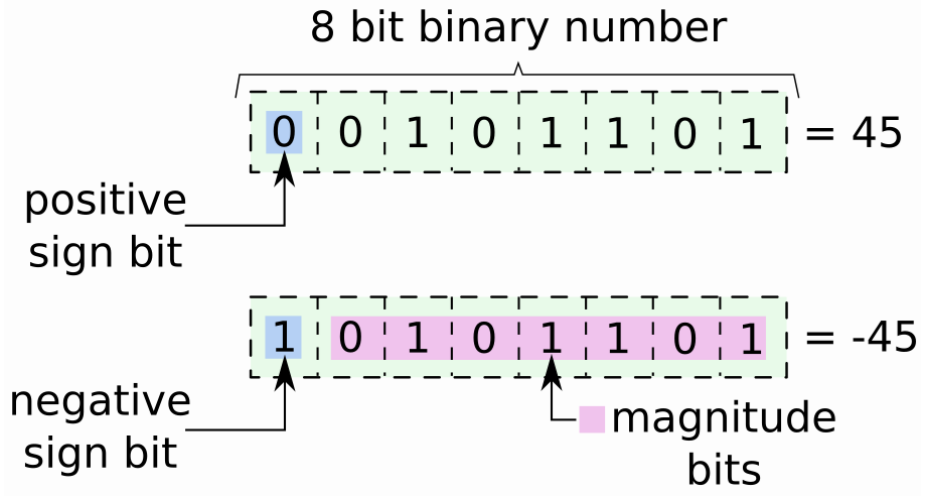
- **Examples** (for different languages)

Data Type	Python	Java	C++	Size
Integer	int	byte (8-bit), short (16-bit), int (32-bit), long (64-bit)	char (8-bit), short (16-bit), int (32-bit), long (64-bit), long long (64-bit)	**8-bit to 64-bit**

Integer Type	Signed Range (default)	Unsigned Range
8-bit	-128 to 127	0 to 255
16-bit	-32,768 to 32,767	0 to 65,535
32-bit	-2,147,483,648 to 2,147,483,647	0 to 4,294,967,295
64-bit	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615
Python (int)	Unbounded (limited by available memory)	

- Why?

Signed Int8



Data Type

"A classification that specifies the kind of value an object can have and what operations can be performed on that object."

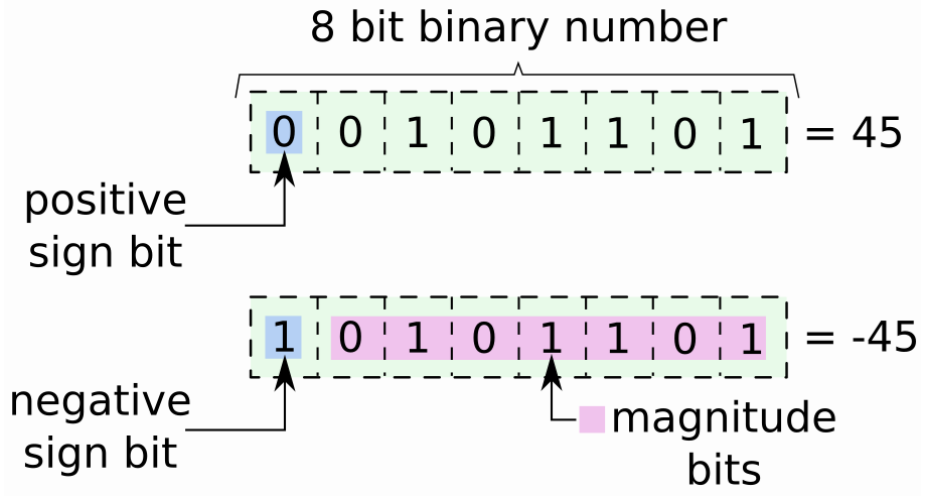
- **Examples** (for different languages)

Data Type	Python	Java	C++	Size
Integer	int	byte (8-bit), short (16-bit), int (32-bit), long (64-bit)	char (8-bit), short (16-bit), int (32-bit), long (64-bit), long long (64-bit)	**8-bit to 64-bit**

Integer Type	Signed Range (default)	Unsigned Range
8-bit	-128 to 127	0 to 255
16-bit	-32,768 to 32,767	0 to 65,535
32-bit	-2,147,483,648 to 2,147,483,647	0 to 4,294,967,295
64-bit	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615
Python (int)	Unbounded (limited by available memory)	

- Why?
  - Optimizing memory usage
  - Range extension

Signed Int8



Variables

"A symbolic name that is a reference or pointer to an object."

- **Characteristics**
  - **Dynamic Typing** (in Python): The type of the variable is determined at runtime and can change during execution.
  - **Assignment:** Variables are assigned values using the " = " operator.
  - **Naming Rules**: Variable names must start with a letter or underscore, followed by letters, digits, or underscores (e.g., `my_var`, `_var`, `var1`).

Python	C++

`age = 25`
`height = 5.9`
`name = "Alice"`
`is_student = True`
	`int age = 25;`
	`float height = 5.9f;`
	`std::string name = "Alice";`
	`bool is_student = true;`

Variables are assigned without specifying a type.	Variables must be declared with a type before assignment.

Containers

*"A symbolic name that is a reference or pointer an object that holds **a collection of other objects**."*

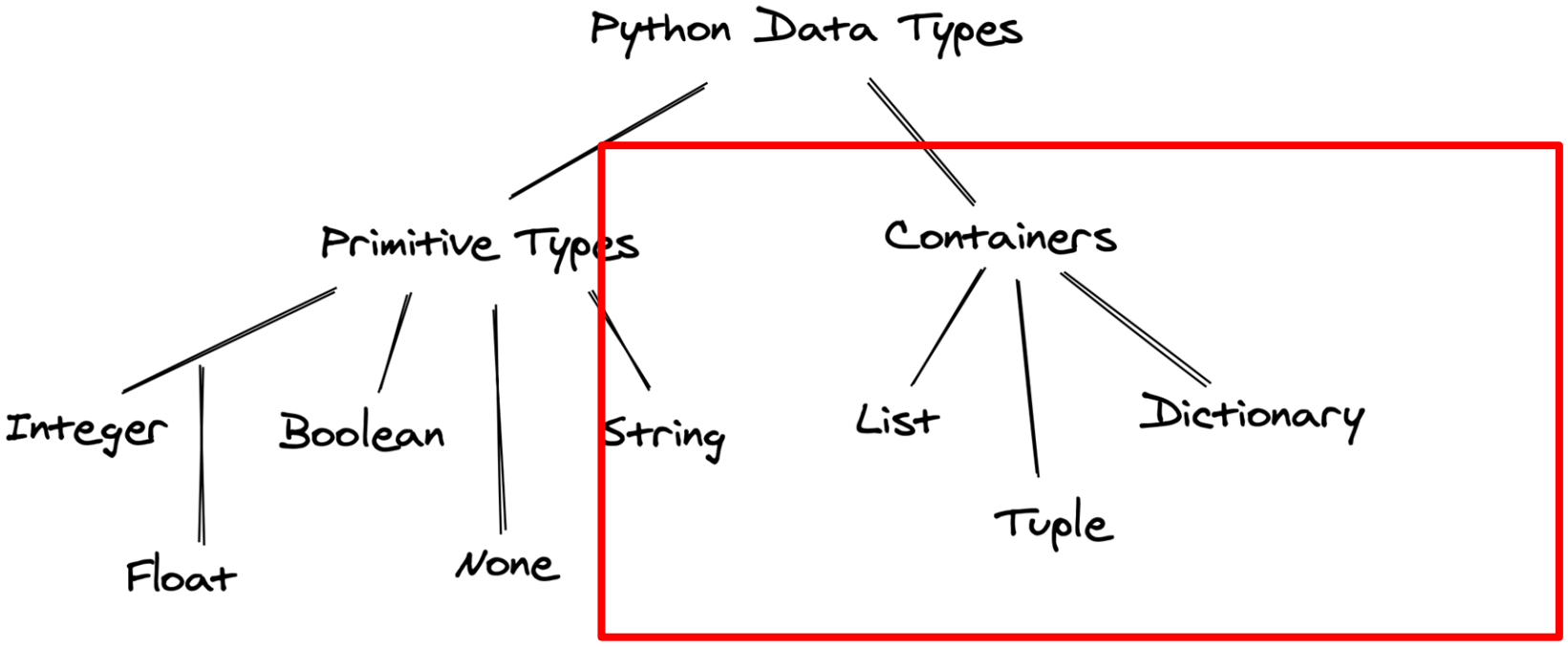


Illustration: https://jovian.ai/learn/data-analysis-with-python-zero-to-pandas

Containers

*"A symbolic name that is a reference or pointer an object that holds **a collection of other objects.**"*

Container Type	Description	Example

List	Ordered, mutable, allows duplicate, indexable, Heterogeneous	`my_list = [1, 2, 3, 4]`
		`my_list.append(5)  # Adds 5 to the list`
		`my_list[0] = 10  # Changes the first element to 10`
		`print(my_list)  # Output: [10, 2, 3, 4, 5]`

Tuple	Ordered, immutable, allows duplicate, indexable, Heterogeneous	`my_tuple = (1, 2, 3, 4)`
		`print(my_tuple[1])  # Output: 2`
		`my_tuple[0] = 10 # This would raise a TypeError`

Dictionary	Unordered, mutable, keys are unique, not indexable, heterogeneous	`my_dict = {'a': 1, 'b': 2}`
		`my_dict['c'] = 3  # Adds a new key-value pair`
		`my_dict['a'] = 10  # Changes the value associated with key 'a'`
		`print(my_dict)  # Output: {'a': 10, 'b': 2, 'c': 3}`

Ordered: Elements maintain their insertion order / **Unordered:** Elements do not maintain their order.
Immutable: Once created, cannot be modified / **Mutable:** Can be modified after creation
Heterogeneous: Can store elements of different data types / **Homogeneous**: Can only store characters data types
Illustration: https://jovian.ai/learn/data-analysis-with-python-zero-to-pandas

Containers

*"A symbolic name that is a reference or pointer an object that holds **a collection of other objects.**"*

Container Type	Description	Example

Set	Unordered collection of unique elements (no duplicates), mutable, not indexable, heterogeneous	`unique_numbers = {1, 2, 3, 3, 2}`
		`unique_numbers.add(4)  # Adds 4 to the set`
		`unique_numbers.remove(1)  # Removes 1 from the set`
		`print(unique_numbers)  # Output: {2, 3, 4}`

String	Ordered, immutable sequence of characters (homogeneous), supports slicing, Indexable	`greeting = "hello"`
		`print(greeting[1])  # Output: 'e'`
		`greeting[1] = 's'  # Error`

Ordered: Elements maintain their insertion order / **Unordered:** Elements do not maintain their order.
Immutable: Once created, cannot be modified / **Mutable:** Can be modified after creation
Heterogeneous: Can store elements of different data types / **Homogeneous**: Can only store characters data types

Built-in functions (Python):

- Predefined functions always available and do not require importing any modules.

- These functions perform various operations such as:
  - mathematical calculations, type conversions, input/output operations, and data manipulation.

Built-in Functions (Python)			
A abs() aiter() all() anext() any() ascii() **B** bin() bool() breakpoint() bytearray() bytes() **C** callable() chr() classmethod() compile() complex() **D** delattr() dict() dir() divmod()	**E** enumerate() eval() exec() **F** filter() float() format() frozenset() **G** getattr() globals() **H** hasattr() hash() help() hex() **I** id() input() int() isinstance() issubclass() iter()	**L** len() list() locals() **M** map() max() memoryview() min() **N** next() **O** object() oct() open() ord() **P** pow() print() property()	**R** range() repr() reversed() round() **S** set() setattr() slice() sorted() staticmethod() str() sum() super() **T** tuple() type() **V** vars() **Z** zip() _ __import__()

Built-in functions (Python):

- Predefined functions always available and do not require importing any modules.

- These functions perform various operations such as:
  - mathematical calculations, type conversions, input/output operations, and data manipulation.

Built-in Functions (Python)			
A abs() aiter() all() anext() any() ascii() **B** bin() bool() breakpoint() bytearray() bytes() **C** callable() chr() classmethod() compile() complex() **D** delattr() dict() dir() divmod()	**E** enumerate() eval() exec() **F** filter() float() format() frozenset() **G** getattr() globals() **H** hasattr() hash() help() hex() **I** id() input() int() isinstance() issubclass() iter()	**L** len() list() locals() **M** map() max() memoryview() min() **N** next() **O** object() oct() open() ord() **P** pow() print() property()	**R** range() repr() reversed() round() **S** set() setattr() slice() sorted() staticmethod() str() sum() super() **T** tuple() type() **V** vars() **Z** zip() _ __import__()

Worth mentioning (especially for debugging…)

- **Type:** Get the data type of a variable with the `type()` function.

- **Casting (in Python):** Setting the Specific Data Type

    - Why?

Worth mentioning (especially for debugging…)

- **Type:** Get the data type of a variable with the `type()` function.

- **Casting (in Python):** Setting the Specific Data Type

  - Why?
    - …some operations require the operands to be of the same type.
    - …convert between container (Homogeneous).
    - …handling user inputs, eg. a calculator?

Worth mentioning (especially for debugging…)

- **Type:** Get the data type of a variable with the `type()` function.

- **Casting (in Python):** Setting the Specific Data Type
  - If you want to specify the data type of a variable, this can be done with casting.
    - `int()` - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)
    - `float()` - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
    - `str()` - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Example: Casting and type using **Python**

String to Integer

```
s = '456'
print(type(s)) # Output: <class 'str'>
int_s = int(s)
print(int_s) # Output: 456
print(type(int_s)) # Output: <class 'int'>
```

Float to String

```
num = 3.14
str_num = str(num)
print(str_num)  # Output: '3.14'
print(type(str_num)) # Output: <class 'str'>
```

Float to Integer

```
f = 10.7
int_f = int(f)
print(int_f)  # Output: 10
print(type(int_f))  # Output: <class 'int'>
```

Worth mentioning (especially for debugging…)

- **Type:** Get the data type of a variable with the `type()` function.

- **Casting (comparison):** Setting the Specific Data Type
  - If you want to specify the data type of a variable, this can be done with casting.

Examples: Float to Integer (comparison between programming languages)

Python

```python
float_number = 12.34
int_number = int(float_number)
print(int_number)  # Output: 12
```

C++

```cpp
#include <iostream>
using namespace std;

int main() {
    float float_number = 12.34;
    int int_number = static_cast<int>(float_number);
    cout << int_number << endl;  // Output: 12
    return 0;
}
```

Java

```java
public class Main {
    public static void main(String[] args) {
        double floatNumber = 12.34;
        int intNumber = (int) floatNumber;
        System.out.println(intNumber);  // Output: 12
    }
}
```

Length: Get the number of items in an object using the `len()` function.

Object Type	Example	Example	Output	Description
String	`my_string = "Hello, World!"`	`type(my_string)` `len(my_string)`	`<class 'str'>` `13`	Number of characters in the string
List	`my_list = [1, 2, 3, 4, 5]`	`type(my_list)` `len(my_list)`	`<class 'list'>` `5`	Number of elements in the list
Tuple	`my_tuple = (10, 20, 30, 40)`	`type(my_tuple)` `len(my_tuple)`	`<class 'tuple'>` `4`	Number of items in the tuple
Dictionary	`my_dict = {'a': 1, 'b': 2, 'c': 3}`	`type(my_dict)` `len(my_dict)`	`<class 'dict'>` `3`	Number of key-value pairs in the dictionary
Set	`my_set = {1, 2, 3, 4, 5}`	`type(my_set)` `len(my_set)`	`<class 'set'>` `5`	Number of unique elements in the set

Nested List	`nested_list = [ [1, 2, 3], [4, 5], [6] ]`	`type(nested_list)`
		`len(nested_list)`
		`len(nested_list[1])`
			`<class 'list'>`
			`3`
			`2`
				Number of sublists in the nested list

Range: Generate a sequence of numbers using the `range()` function, especially within a for loop.

List the numbers from 0 to 5

```
numbers = list(range(5))
print(numbers) # Output: [0, 1, 2, 3, 4]
```

List the numbers based on the length of the Tuple

```
my_tuple = (10, 20, 30, 40)
numbers = list(range(len(my_tuple)))
print(numbers) # Output: [0, 1, 2, 3]
```

Task: Python Operators and Data Types (40 min / 10.00)

(Assuming that you have the ***hello_world.ipynb*** up-and-running)

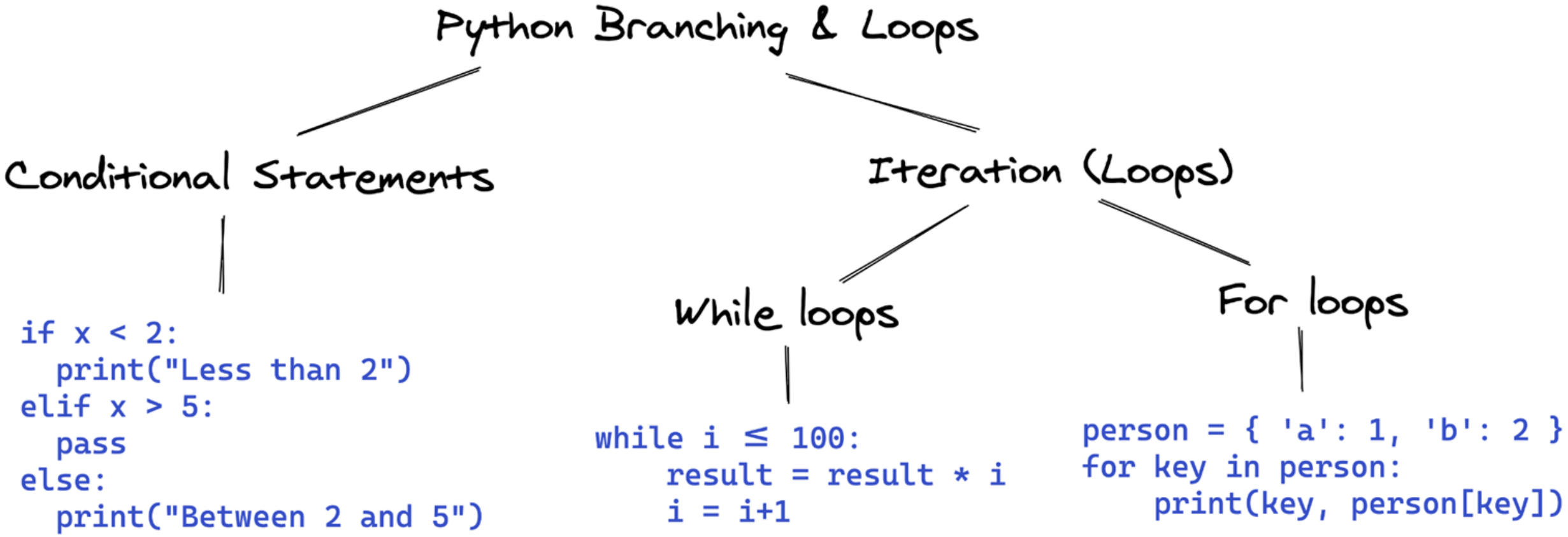
1. Go to ItsLearning, and download the ***pmr_02_exercises.ipynb***

2. Run it in the same environment as you teste the ***hello_world.ipynb***

3. Get as far as you can with:
   a. **Assignment #1:** Performing Arithmetic Operations using Python
   b. **Assignment #2:** Variables and Data Types in Python

(if you finished #1 and #2 within the timeframe,

start answering the questions in the bottom of the file)



Source: https://jovian.ai/learn/data-analysis-with-python-zero-to-pandas

Branching

“A fundamental concept in programming that allows you to control the flow of your program based on conditions.”

= moving the Program Counter to a new address

…causing the CPU to begin executing a different instruction sequence.

Branching

“A fundamental concept in programming that allows you to control the flow of your program based on conditions.”

= moving the Program Counter to a new address

…causing the CPU to begin executing a different instruction sequence

- ***Conditional*** *branch (conditional statements):*
  - may or may not cause branching depending on some condition.

- ***Unconditional*** *branch (iterations / loops):*
  - always results in branching

Conditional statements

“A fundamental construct in programming that allows you to execute specific blocks of code based on whether a condition is `True` or `False`“

Usage:

- **Condition**: An expression that evaluates to `True` or `False`.
- **Control Statements**: `if`, `else`, `elif`.

Conditional statements

"A fundamental construct in programming that allows you to execute specific blocks of code based on whether a condition is `True` or `False`"

Syntax and usage:

- **Condition**: An expression that evaluates to `True` or `False`.
- **Control Statements**: `if` `else` `elif`

Example: If ... Else syntax

Python

```
if condition1:
    # Code to execute if condition1 is true
elif condition2:
    # Code to execute if condition2 is true
else:
    # Code to execute if none of the above conditions are true
```

C++

```
if (condition1) {
    // Code to execute if condition1 is true
} else if (condition2) {
    // Code to execute if condition2 is true
} else {
    // Code to execute if none of the above conditions are true
}
```

Uses **indentation** to indicate a block of code.	Uses **curly brackets** to indicate a block of code.

Conditional statements

"A fundamental construct in programming that allows you to execute specific blocks of code based on whether a condition is `True` or `False`"

Example: If ... Else

```
age = 20

if age < 18:
    print("You are a minor.")
elif age >= 18 and age < 65:
    print("You are an adult.")
else:
    print("You are a senior citizen.")
```

Conditional statement If ... Else

```
age = 20
has_license = True

if age >= 18 and has_license:
    print("You are allowed to drive.") # Result
else:
    print("You are not allowed to drive.")
```

Combining conditions with Logical Operators

```
age = 20
has_license = False

if age >= 18:
    if has_license:
        print("You are allowed to drive.")
    else:
        print("You need a license to drive.")
# Result
else:
    print("You are not allowed to drive.")
```

Nested If ... Else

Conditional statements (Shorthand If .. Else, a bit C/C++-like syntax)

Example: Shorthand If ... Else

```
a = 5
b = 4

# Short Hand If
if a > b: print("a is greater than b")

# Short Hand If ... Else
print("A") if a > b else print("B")
print("A") if a > b else print("=") if a == b else print("B")
```

Conditional statements

- **If ... Else** versus **Match**?
  a. **Match:** Introduced in Python 3.10, same purpose, more modern/readable syntax.
     i. (Similar to switch-cases in C++)

Example: Comparison

```
# Define the animal type
animal = "dog"

# Categorize the animal
if animal == "dog":
    print("This is a mammal.")
elif animal == "eagle":
    print("This is a bird.")
elif animal == "shark":
    print("This is a fish.")
elif animal == "frog":
    print("This is an amphibian.")
elif animal == "crocodile":
    print("This is a reptile.")
else:
    print("Unknown animal type.")
```

```
# Define the animal type
animal = "dog"

# Categorize the animal using match
match animal:
    case "dog":
        print("This is a mammal.")
    case "eagle":
        print("This is a bird.")
    case "shark":
        print("This is a fish.")
    case "frog":
        print("This is an amphibian.")
    case "crocodile":
        print("This is a reptile.")
    case _:
        print("Unknown animal type.")
```

Loops

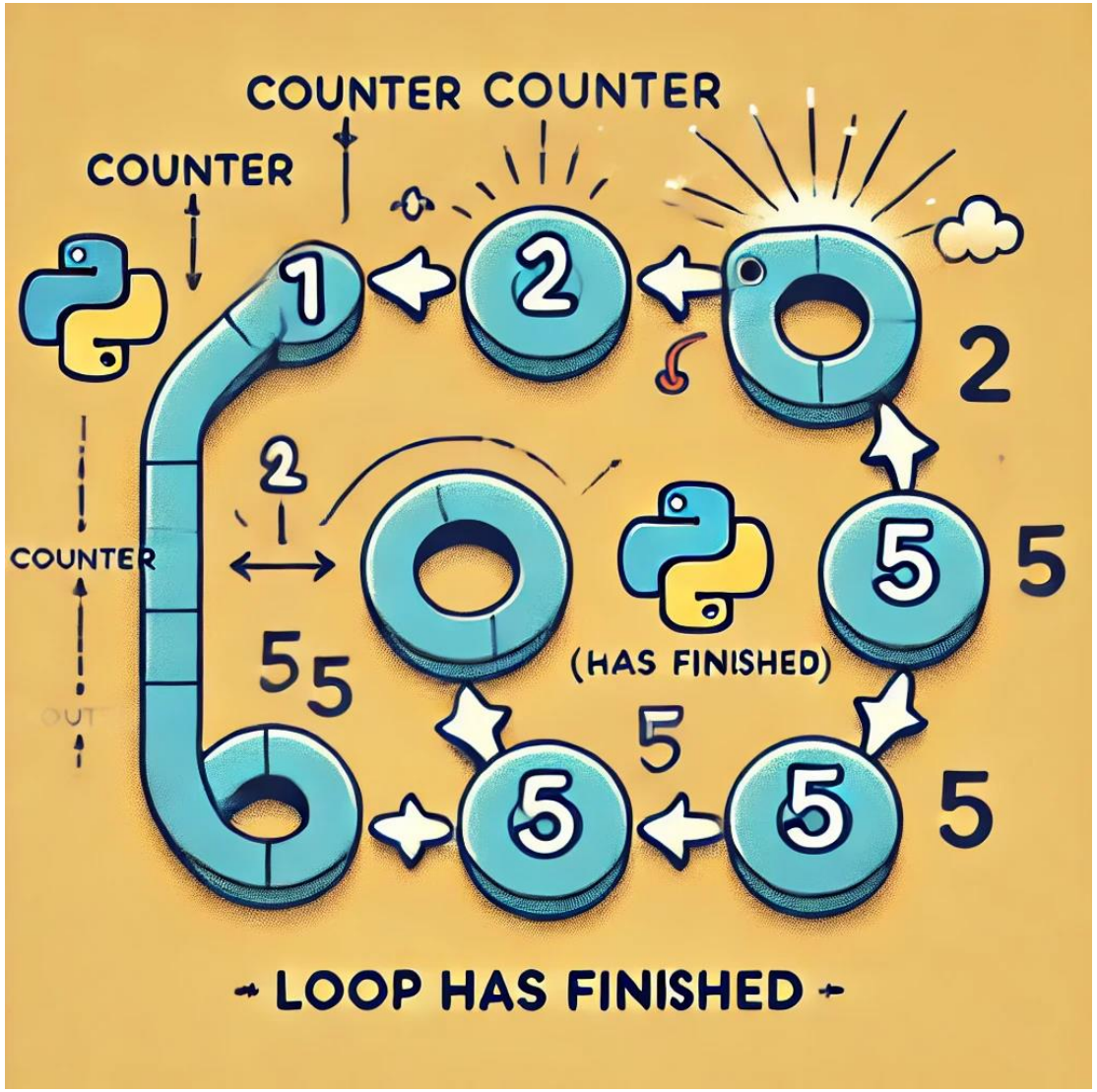


Illustration: generated using OpenAI's DALL·E tool

Loops

"A fundamental concept in programming that allow you to execute a block of code multiple times."

- ***For loop:*** iterating over a sequence or container (such as a list, tuple, set, or string).

- ***While loop:*** repeatedly executes a block of code as long as a given condition is true.

Example: Comparison in the programming syntax	
Python	**C++**

`for condition:`
`  # Code to execute if condition is true`
	`for (condition) {`
	`  // Code to execute if condition is true`
	`}`

`while condition:`
`  # Code to execute if condition is true`
	`while (condition) {`
	`  // Code to execute if condition is true`
	`}`

Uses indentation to indicate a block of code.	Uses curly brackets to indicate a block of code.

Loops

"A fundamental concept in programming that allow you to execute a block of code multiple times."

- ***For loop:*** iterating over a sequence or container (such as a list, tuple, set, or string).
- ***While loop:*** repeatedly executes a block of code as long as a given condition is true.
- **Control Statements**: `break`, `continue`, `return`, and ~~`goto`~~ (in some languages).

Example: Comparison in the programming syntax	
Python	**C++**

`for condition:`
`  # Code to execute if condition is true`
	`for (condition) {`
	`  // Code to execute if condition is true`
	`}`

`while condition:`
`  # Code to execute if condition is true`
	`while (condition) {`
	`  // Code to execute if condition is true`
	`}`

Uses indentation to indicate a block of code.	Uses curly brackets to indicate a block of code.

Loops (key concepts)

- ***For loop:*** iterating over a sequence or container (such as a list, tuple, set, or string).

Example: Syntax	
Python	**Example**

`for condition:`
`  # Code to execute if condition is true`
	`fruits = ["apple", "banana", "cherry"]`
	
	`for fruit in range(len(fruits)):`
	`  print(fruits[fruit])`

	`fruits = ["apple", "banana", "cherry"]`
	
	`for fruit in fruits:`
	`  print(fruit)`
	
	`# Both examples prints:`
	`# apple`
	`# banana`
	`# cherry`

Loops (key concepts)

- ***Nested For loop:*** iterating over a sequence/container of sequences/containers.

Example: Iterating Over a List of Lists

```
for outer_variable in outer_sequence:
    for inner_variable in inner_sequence:
        # Code to execute
```

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

for row in matrix:
    for element in row:
        print(element, end=' ')
    print()  # For new line after each row

# Prints
# 1 2 3
# 4 5 6
# 7 8 9
```

Loops (key concepts)

- ***Nested For loop:*** iterating over a sequence/container of sequences/containers.

Example: Iterating Over a List of Lists

```
for outer_variable in outer_sequence:
    for inner_variable in inner_sequence:
        # Code to execute
```

Outer Loop: The first loop iterates through the outer sequence (e.g., rows in a matrix, or the first list in a pair generation).

Inner Loop: The nested loop iterates through the inner sequence (e.g., elements within a row, or the second list in a pair generation).

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

for row in matrix:
    for element in row:
        print(element, end=' ')
    print()  # For new line after each row

# Prints
# 1 2 3
# 4 5 6
# 7 8 9
```

Loops (key concepts)

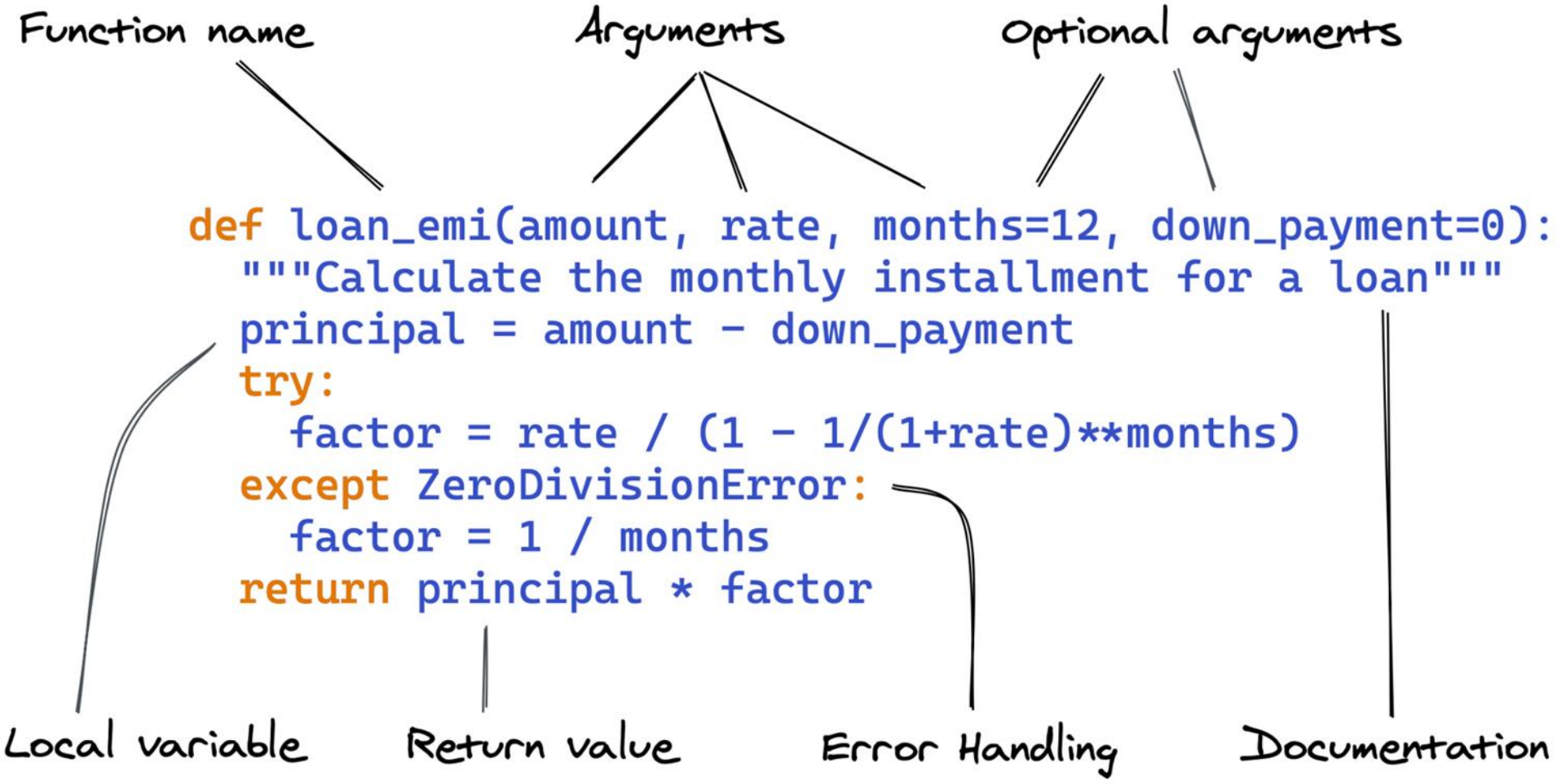
- ***While loop:*** repeatedly executes a block of code as long as a given condition is true

Example: Syntax

Python	Example

`while condition:`
`  # Code to execute if condition is true`
	`i = 0`
	
	`while True: # Count forever....`
	`  print(i)`
	`  i += 1`

`ready = get_status()`
 	
`while not ready: # Keep looping until ready....`
`  ready = get_status()`
 	
`# Continue when ready!`
	`i = 0`
	
	`while i < 5: # Count 0 . . . 4`
	`  print(i)`
	`  i += 1`



Source: https://jovian.ai/learn/data-analysis-with-python-zero-to-pandas

Function

“A block of reusable code that performs a specific task”

Syntax and usage:

- ***Redundancy:*** Reusing code blocks
- ***Readability:*** Organizes your code by moving parts out of your main loop

Syntax: Python	Example: Define a function

`def function_name(parameters):`
`  # Function body`
`  return value  # (optional) return statement`
	`# Print “Hello, World!”`
	`print("Hello, World!")`

Start by using the `def` keyword, followed by the function name and parentheses. Inside the parentheses, you can list any parameters the function will accept. Uses indentation to indicate a block of code.	

Function

“A block of reusable code that performs a specific task”

Syntax and usage:

- ***Redundancy:*** Reusing code blocks
- ***Readability:*** Organizes your code by moving parts out of your main loop

Syntax: Python	Example: Define a function

`def function_name(parameters):`
`  # Function body`
`  return value  # (optional) return statement`
	`def hello():`
	`  print("Hello, World!")`
	
	`# Calling the function`
	`hello()`

Start by using the `def` keyword, followed by the function name and parentheses. Inside the parentheses, you can list any parameters the function will accept. Uses indentation to indicate a block of code.	

Function

“A block of reusable code that performs a specific task”

Syntax and usage:

- ***Redundancy:*** Reusing code blocks
- ***Readability:*** Organizes your code by moving parts out of your main loop

Syntax: Python	Example: Implement argument

`def function_name(parameters):`
`  # Function body`
`  return value  # (optional) return statement`
	`def hello(name):`
	`  print(f"Hello, {name}!")`
	
	`# Calling the function`
	`hello("Jes") # Output: Hello, Jes!`

Start by using the `def` keyword, followed by the function name and parentheses. Inside the parentheses, you can list any parameters the function will accept. Uses indentation to indicate a block of code.	

Function

“A block of reusable code that performs a specific task”

Syntax and usage:

- ***Redundancy:*** Reusing code blocks
- ***Readability:*** Organizes your code by moving parts out of your main loop

Syntax: Python	Example: Implement default argument

`def function_name(parameters):`
`  # Function body`
`  return value  # (optional) return statement`
	`def hello(name="World"):`
	`  print(f"Hello, {name}!")`
	
	`# Calling the function`
	`hello() # Output: Hello, World!`
	`hello("Jes") # Output: Hello, Jes!`

Start by using the `def` keyword, followed by the function name and parentheses. Inside the parentheses, you can list any parameters the function will accept. Uses indentation to indicate a block of code.	

Function

“A block of reusable code that performs a specific task”

Syntax and usage:

- ***Redundancy:*** Reusing code blocks
- ***Readability:*** Organizes your code by moving parts out of your main loop

Syntax: Python	Example: Implement default argument

`def function_name(parameters):`
`  # Function body`
`  return value  # (optional) return statement`
	`def hello(name="World"):`
	`  """`
	`  This function prints a personalized greeting message.`
	
	`  :param name: The name of the person to greet. Default: "World".`
	`  """`
	`  print(f"Hello, {name}!")`
	
	`# Calling the function`
	`hello("Jes") # Output: Hello, Jes!`

Start by using the `def` keyword, followed by the function name and parentheses. Inside the parentheses, you can list any parameters the function will accept. Uses indentation to indicate a block of code.	

Function

“A block of reusable code that performs a specific task”

Syntax and usage:

- ***Redundancy:*** Reusing code blocks
- ***Readability:*** Organizes your code by moving parts out of your main loop

Syntax: Python	Example: Handle Multiple Greetings (improving)

`def function_name(parameters):`
`  # Function body`
`  return value  # (optional) return statement`
	`def hello(*names):`
	`  """`
	`  This function prints a greeting message for each name provided.`
	
	`  :param names: A list of names to greet.`
	`  """`
	`  for name in names:`
	`  print(f"Hello, {name}!")`
	
	`# Calling the function`
	`hello("Jes", "Bo", "Hans")`
	`# Output: Hello, Jes! Hello, Bo! Hello, Hans!`

Start by using the `def` keyword, followed by the function name and parentheses. Inside the parentheses, you can list any parameters the function will accept. Uses indentation to indicate a block of code.	

Scope

"The scope of a variable refers to the region of the code where the variable is recognized."

Syntax and usage:

- ***Global scope:*** A variable defined in the main body of the Python code is a global variable. It can be accessed anywhere in the code, both inside and outside functions.

- ***Local scope:*** A variable defined inside a function is a local variable. It can only be accessed within that function and not outside.

Example: Global scope

```
x = 10  # Global variable

def print_global():
    print(x)  # Accessing the global variable

print_global()  # Output: 10
print(x)  # Output: 10
```

Example: Local scope

```
def print_local():
    x = 5  # Local variable
    print(x)  # Accessing the local variable

print_local()  # Output: 5
print(x)  # Error: NameError: name 'x' is not defined
```

Scope

“The scope of a variable refers to the region of the code where the variable is recognized.”

Syntax and usage:

- ***Global Keyword:*** If you need to modify a global variable inside a function, you can use the global keyword. This keyword tells Python that you are referring to the global variable, not a local one.

Example: Local scope

```
x = 10  # Global variable

def print_global():
    global x  # Accessing the global variable
    print(x)
    x += 1 # Modifying the global variable

print_global()  # Output: 10
print(x)  # Output: 11
```

Troubleshooting and Debugging

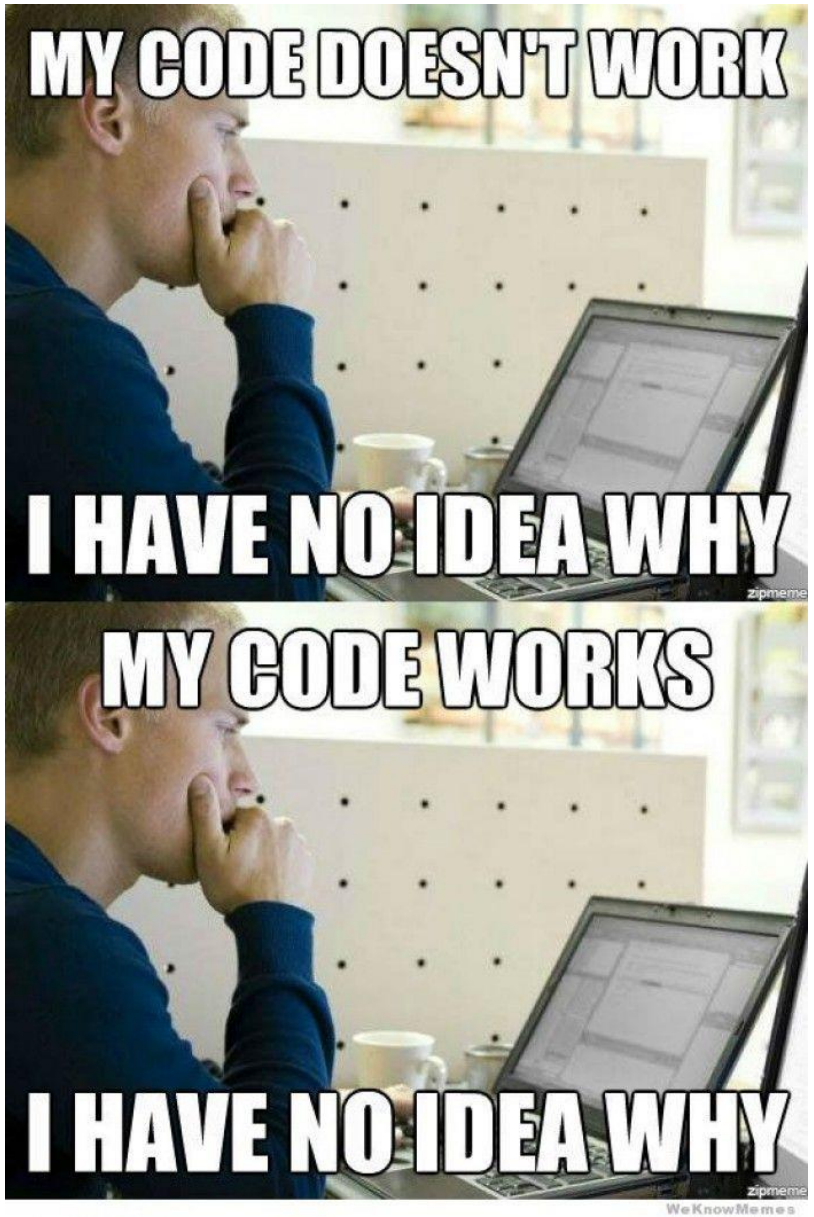
Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

Best practise:

"Try again, ask a friend, ask an adult…"

- …when you get an error, look at the:
  - **Error Type**: The type of error (e.g., SyntaxError, TypeError, ValueError).
  - **Traceback**: A list showing the sequence of calls that led to the error.
  - **Error Message**: A description of what caused the error (e.g., "division by zero").



Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

Common errors (in Python)**:**

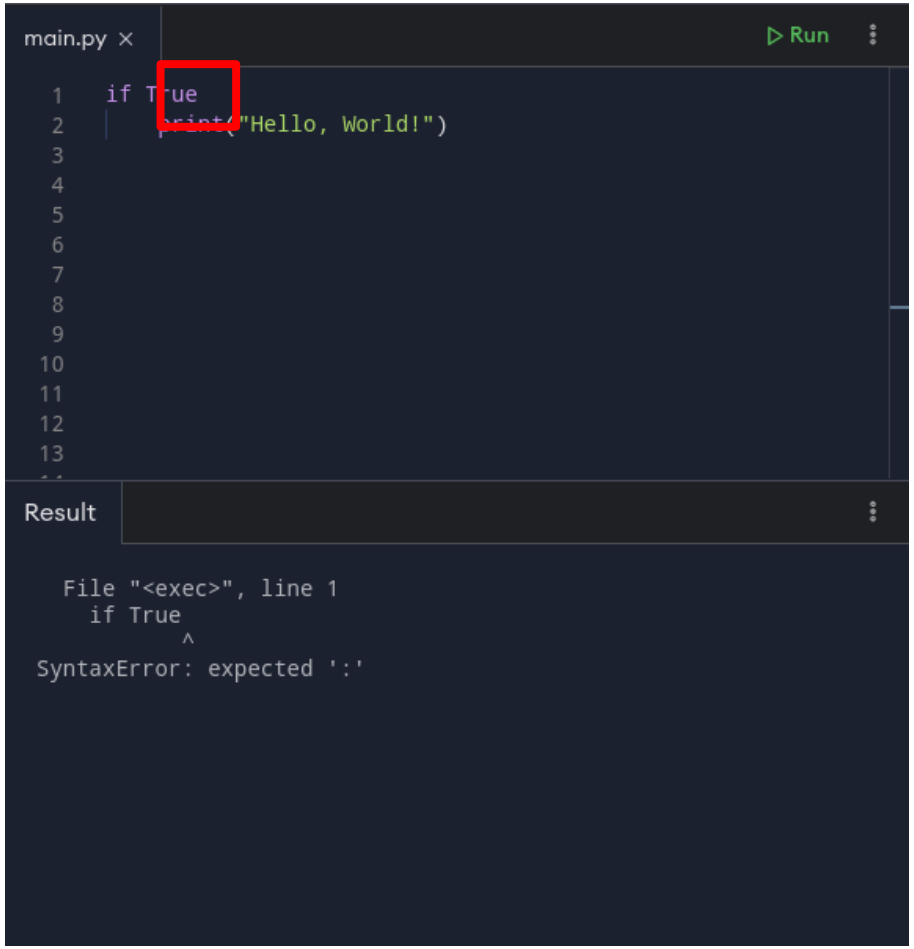
- **Syntax Errors:** Occur when the interpreter encounters code that doesn't conform to the language's syntax rules.
- **Indentation Errors:** Misaligned or inconsistent indentation can lead to errors.
- **Name Errors:** A variable or function is used before it's defined or misspelled.
- **Type Errors:** Occur when an operation or function is applied to an object of inappropriate type (e.g., adding a string to an integer).
- **Index/Key Errors:** When trying to access an index that is out of range in a list or string, or trying to access a dictionary with a key that does not exist.
- **Value Errors:** When a function receives an argument of the correct type but an inappropriate value.

Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

Common errors (in Python)**:**

- **Syntax Errors:** Occur when the interpreter encounters code that doesn't conform to the language's syntax rules.
- **Indentation Errors:** Misaligned or inconsistent indentation can lead to errors.
- **Name Errors:** A variable or function is used before it's defined or misspelled.
- **Type Errors:** Occur when an operation or function is applied to an object of inappropriate type (e.g., adding a string to an integer).
- **Index/Key Errors:** When trying to access an index that is out of range in a list or string, or trying to access a dictionary with a key that does not exist.
- **Value Errors:** When a function receives an argument of the correct type but an inappropriate value.



Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

Common errors (in Python)**:**

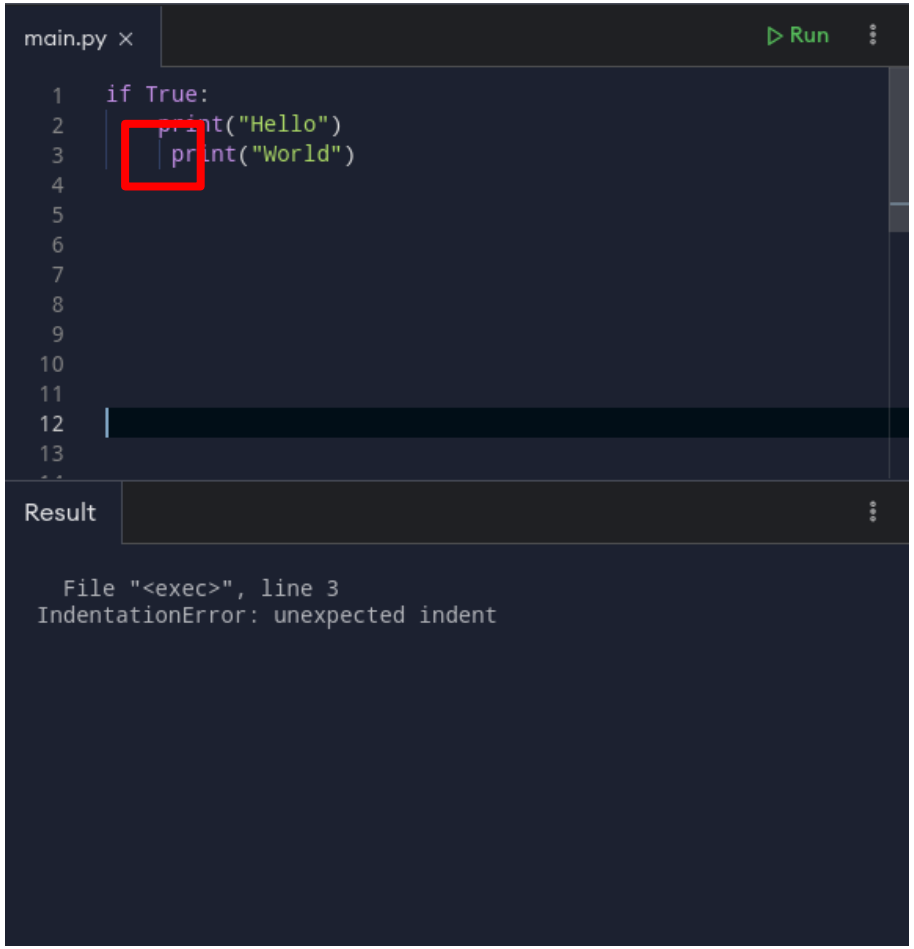
- **Syntax Errors:** Occur when the interpreter encounters code that doesn't conform to the language's syntax rules.
- **Indentation Errors:** Misaligned or inconsistent indentation can lead to errors.
- **Name Errors:** A variable or function is used before it's defined or misspelled.
- **Type Errors:** Occur when an operation or function is applied to an object of inappropriate type (e.g., adding a string to an integer).
- **Index/Key Errors:** When trying to access an index that is out of range in a list or string, or trying to access a dictionary with a key that does not exist.
- **Value Errors:** When a function receives an argument of the correct type but an inappropriate value.

Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

Common errors (in Python)**:**

- **Syntax Errors:** Occur when the interpreter encounters code that doesn't conform to the language's syntax rules.
- **Indentation Errors:** Misaligned or inconsistent indentation can lead to errors.
- **Name Errors:** A variable or function is used before it's defined or misspelled.
- **Type Errors:** Occur when an operation or function is applied to an object of inappropriate type (e.g., adding a string to an integer).
- **Index/Key Errors:** When trying to access an index that is out of range in a list or string, or trying to access a dictionary with a key that does not exist.
- **Value Errors:** When a function receives an argument of the correct type but an inappropriate value.

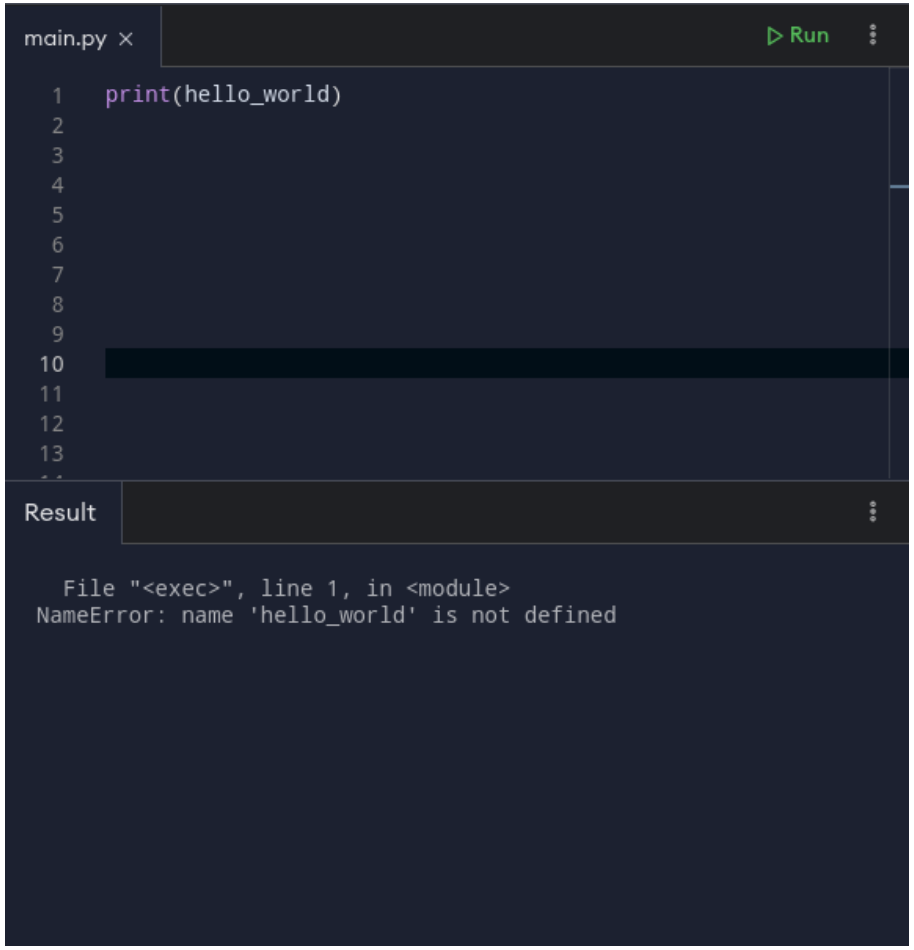


Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

Common errors (in Python)**:**

- **Syntax Errors:** Occur when the interpreter encounters code that doesn't conform to the language's syntax rules.
- **Indentation Errors:** Misaligned or inconsistent indentation can lead to errors.
- **Name Errors:** A variable or function is used before it's defined or misspelled.
- **Type Errors:** Occur when an operation or function is applied to an object of inappropriate type (e.g., adding a string to an integer).
- **Index/Key Errors:** When trying to access an index that is out of range in a list or string, or trying to access a dictionary with a key that does not exist.
- **Value Errors:** When a function receives an argument of the correct type but an inappropriate value.



Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

Common errors (in Python)**:**

- **Syntax Errors:** Occur when the interpreter encounters code that doesn't conform to the language's syntax rules.
- **Indentation Errors:** Misaligned or inconsistent indentation can lead to errors.
- **Name Errors:** A variable or function is used before it's defined or misspelled.
- **Type Errors:** Occur when an operation or function is applied to an object of inappropriate type (e.g., adding a string to an integer).
  - **How to fix this?**
- **Index/Key Errors:** When trying to access an index that is out of range in a list or string, or trying to access a dictionary with a key that does not exist.
- **Value Errors:** When a function receives an argument of the correct type but an inappropriate value.

Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

Common errors (in Python)**:**

- **Syntax Errors:** Occur when the interpreter encounters code that doesn't conform to the language's syntax rules.
- **Indentation Errors:** Misaligned or inconsistent indentation can lead to errors.
- **Name Errors:** A variable or function is used before it's defined or misspelled.
- **Type Errors:** Occur when an operation or function is applied to an object of inappropriate type (e.g., adding a string to an integer).
  - How to fix this? **Casting…**
- **Index/Key Errors:** When trying to access an index that is out of range in a list or string, or trying to access a dictionary with a key that does not exist.
- **Value Errors:** When a function receives an argument of the correct type but an inappropriate value.

Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

(Most) Common errors (in Python)**:**

- **Syntax Errors:** Occur when the interpreter encounters code that doesn't conform to the language's syntax rules.
- **Indentation Errors:** Misaligned or inconsistent indentation can lead to errors.
- **Name Errors:** A variable or function is used before it's defined or misspelled.
- **Type Errors:** Occur when an operation or function is applied to an object of inappropriate type (e.g., adding a string to an integer).
- **Index/Key Errors:** When trying to access an index that is out of range in a list or string, or trying to access a dictionary with a key that does not exist.
    - **How to fix this?**
- **Value Errors:** When a function receives an argument of the correct type but an inappropriate value.

Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

(Most) Common errors (in Python)**:**

- **Syntax Errors:** Occur when the interpreter encounters code that doesn't conform to the language's syntax rules.
- **Indentation Errors:** Misaligned or inconsistent indentation can lead to errors.
- **Name Errors:** A variable or function is used before it's defined or misspelled.
- **Type Errors:** Occur when an operation or function is applied to an object of inappropriate type (e.g., adding a string to an integer).
- **Index/Key Errors:** When trying to access an index that is out of range in a list or string, or trying to access a dictionary with a key that does not exist.
  - How to fix this? **len()...**
- **Value Errors:** When a function receives an argument of the correct type but an inappropriate value.

Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

(Most) Common errors (in Python)**:**

- **Syntax Errors:** Occur when the interpreter encounters code that doesn't conform to the language's syntax rules.
- **Indentation Errors:** Misaligned or inconsistent indentation can lead to errors.
- **Name Errors:** A variable or function is used before it's defined or misspelled.
- **Type Errors:** Occur when an operation or function is applied to an object of inappropriate type (e.g., adding a string to an integer).
- **Index/Key Errors:** When trying to access an index that is out of range in a list or string, or trying to access a dictionary with a key that does not exist.
- **Value Errors:** When a function receives an argument of the correct type but an inappropriate value.

Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

Debugging Techniques

- **Print Statements:** Inserting print() statements in your code can help track the flow of execution and check the values of variables at different points.

Example: Using `print()` for debugging

```
print("I'm here")

...

print("Now I'm here...")

...

print("Counter: {}".format(counter) )
print("Counter: {}".format(type(counter)) )
```

Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

Debugging Techniques

- **Print Statements:** Inserting print() statements in your code can help track the flow of execution and check the values of variables at different points.
- **Try-Except Blocks:** Use try-except blocks to catch and handle exceptions, which can help prevent your program from crashing and allow you to manage errors gracefully.

Example: Try-Except Block

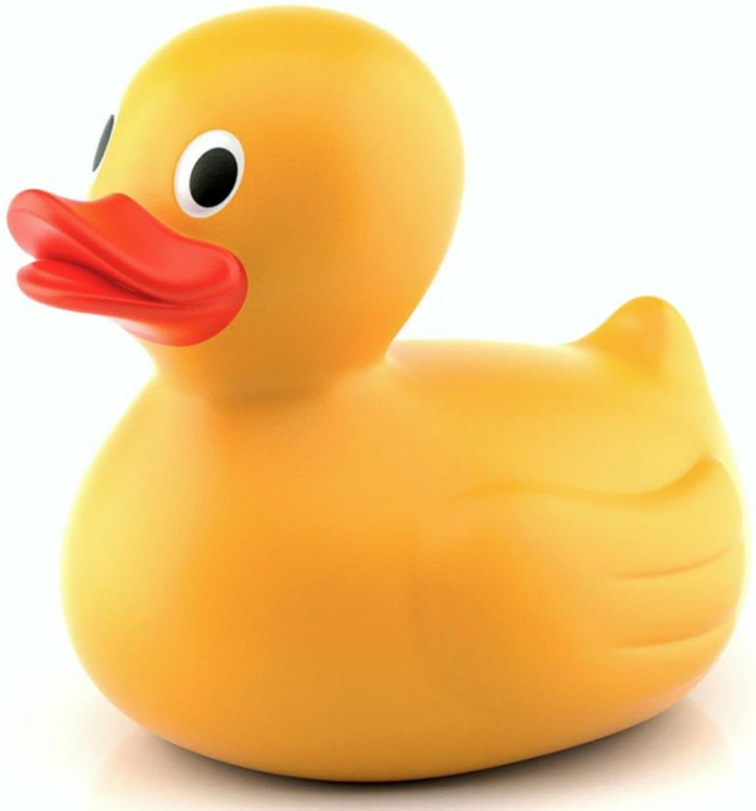
```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error: {e}")
    # Prints --> Error: division by zero
```

Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

Debugging Techniques

- **Print Statements:** Inserting print() statements in your code can help track the flow of execution and check the values of variables at different points.
- **Try-Except Blocks:** Use try-except blocks to catch and handle exceptions, which can help prevent your program from crashing and allow you to manage errors gracefully.
- **Rubberdugging / Rubber Duck Debugging:** Explaining your code, often to an inanimate object like a rubber duck, can help clarify your thinking and reveal bugs you might have missed.

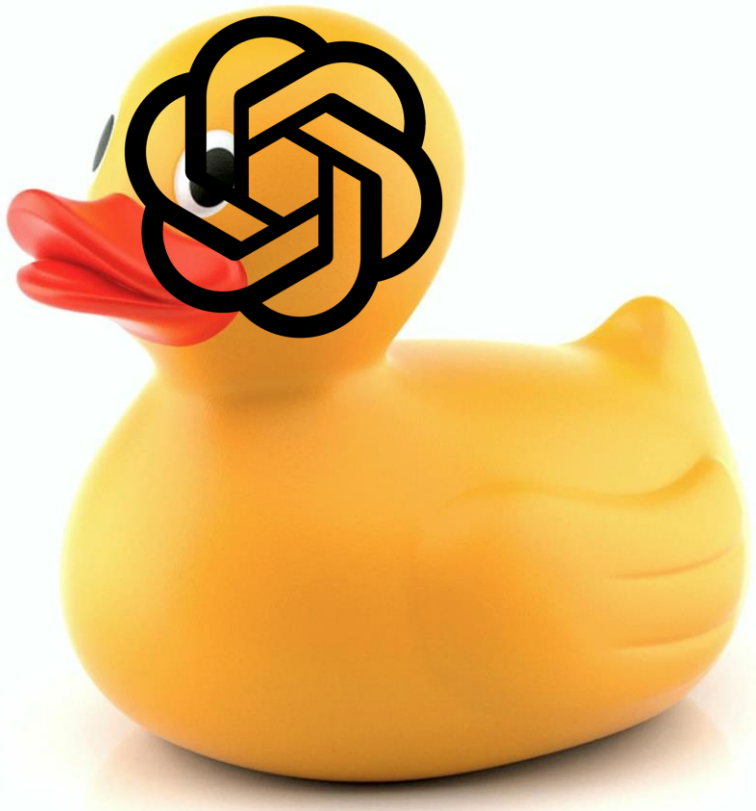


Troubleshooting and Debugging

"...identifying, diagnosing, and resolving errors or bugs that arise when writing or running the code."

Debugging Techniques

- **Print Statements:** Inserting print() statements in your code can help track the flow of execution and check the values of variables at different points.
- **Try-Except Blocks:** Use try-except blocks to catch and handle exceptions, which can help prevent your program from crashing and allow you to manage errors gracefully.
- **Rubberdugging / Rubber Duck Debugging:** Explaining your code, often to an inanimate object like a rubber duck, can help clarify your thinking and reveal bugs you might have missed.
- **Artificial Intelligence…**



Extra Credit Activity 1: Calculator

- Jupyter Notebooks / IPython Notebooks
- Demo

Assignments (***pmr_02_exercises.ipynb***)

- **If not** finished:
  - Assignment #1: Performing Arithmetic Operations using Python
  - Assignment #2: Variables and Data Types in Python
- **else:**
  - Assignment #3: Branching using Conditional Statements and Loops in Python
  - Assignment #4: Functions and scope

…and then “Extra Credit Activity 1: Calculator”