
1878
Western
UNIVERSITY · CANADA

AI-in-Action Heroes

SUMMER ACADEMY COURSE OFFERED BY AISE PROGRAM

The Objectives of the course

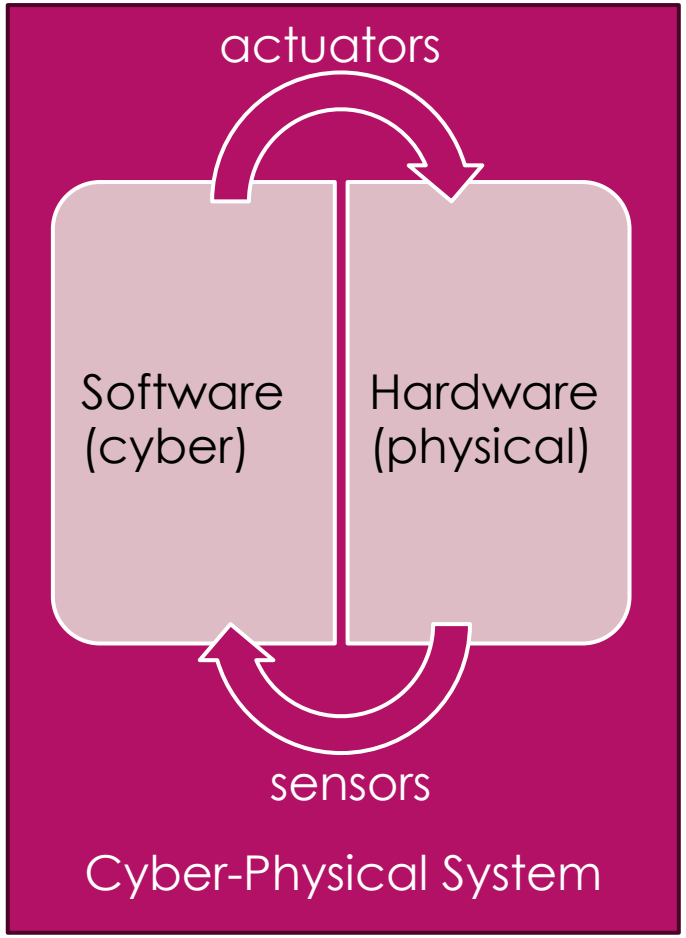
- Get a glance of the big picture and the different aspects and components

Software program

- Allow us to give instructions to a computer in order to solve a problem.
- Instructions written in a language that the computer can understand
- Needs and environment to get executed.

Cyber-Physical System

- Hardware aspects
  - Sensors: collect input from the surroundings and send it to the program.
  - Actuators: take commands form the program and cause the device to take action.
- Software aspects
  - OS, Libraries, …, etc.
  - Your programs



What is AI?

What is artificial intelligence (AI)?

Artificial intelligence leverages computers and machines to mimic the problem-solving and decision-making capabilities of the human mind

Source: https://www.ibm.com/topics/artificial-intelligence

Putting AI in Action

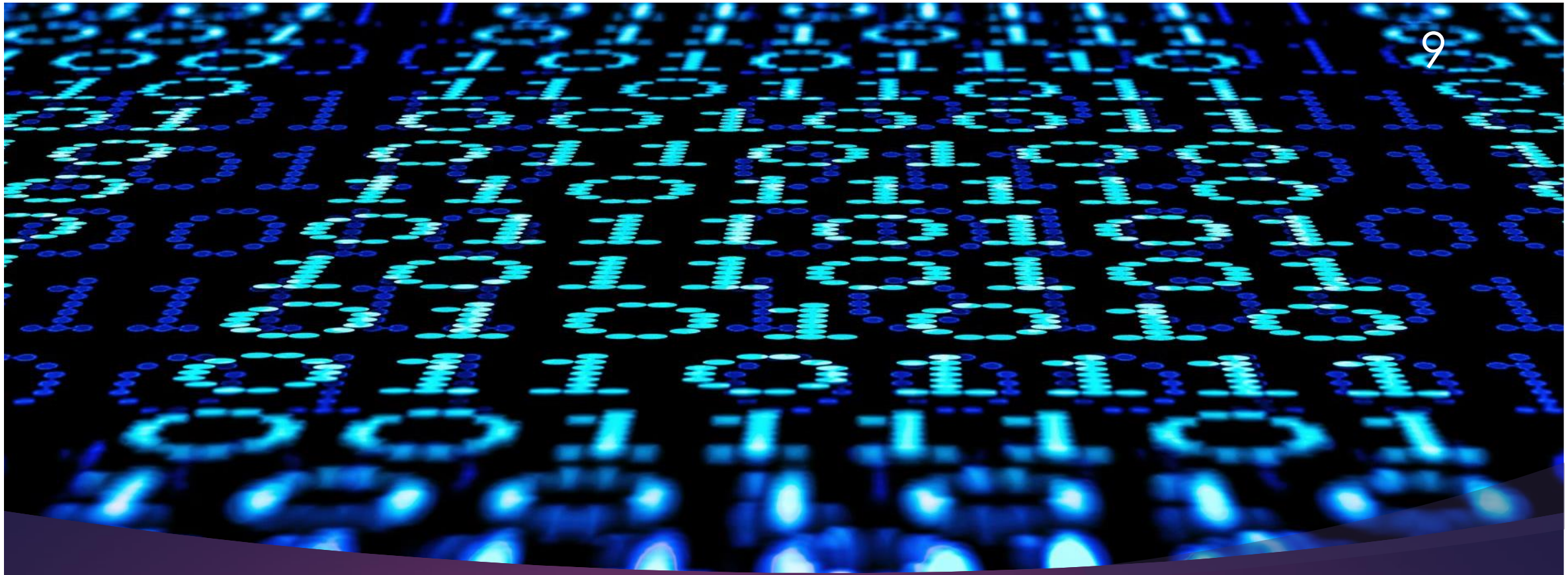
AI applications in the context of different engineering disciplines

- For example, if we are talking about robots
  - Building a robot is one thing.
  - Building a robot with AI capabilities is another thing.
  - Preparing the robot to utilize the capabilities for an AI system is another thing.
  - Putting these AI capabilities in action to fulfill a specific functionality is another thing.

Putting AI in Action

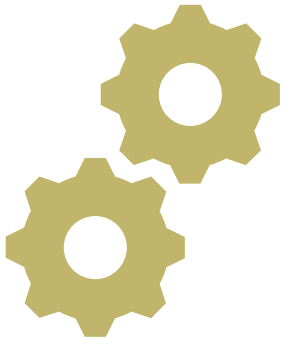
AI applications in the context of different engineering disciplines

- For example, if we are talking about robots
  - Building a robot is one thing.
  - Building a robot with AI capabilities is another thing.
  - **Preparing the robot to utilize the capabilities for an AI system.**
  - **Putting these AI capabilities in action to fulfill a specific functionality is another thing.**



Introduction to Programming

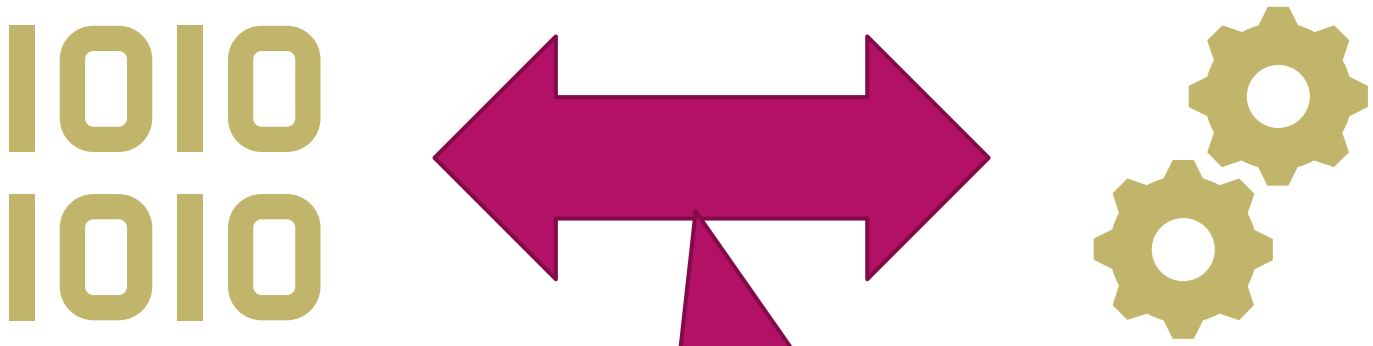
Any computer program consists of two components



Data

Processing

Any computer program consists of two components



Data Processing

We write programs to **tell** the computer:
- What are the data
- How to process them to get the result

Machine language

- Machines operate by electricity
  - 1 = on
  - 0 = off

Machine Language

- Binary code
  - uses two different states, on and off, to translate instructions to its processors.

Programming

is to translate a solution of a problem from human language to a computer/machine language

- Human Language
- "What's the result of adding 5 to 7?"

- Machine language
- 0110100101010

Low Level Language

Assembly Language

- Example problem:
  - Save the number 97 for later use
    - **Data:** the constant value 97
    - **Process:** save that value
- Binary Language

  10110000 01100001

Assembly Language

- Example problem:
  - Save the number 97 for later use
    - **Data:** the constant value 97
    - **Process:** save that value
- Binary Language

10110000 01100001

Binary representation of the number 97

Binary code for saving instruction

Code for the **AL** register where the value to be stored

Assembly Langage

was introduced to make programming easier

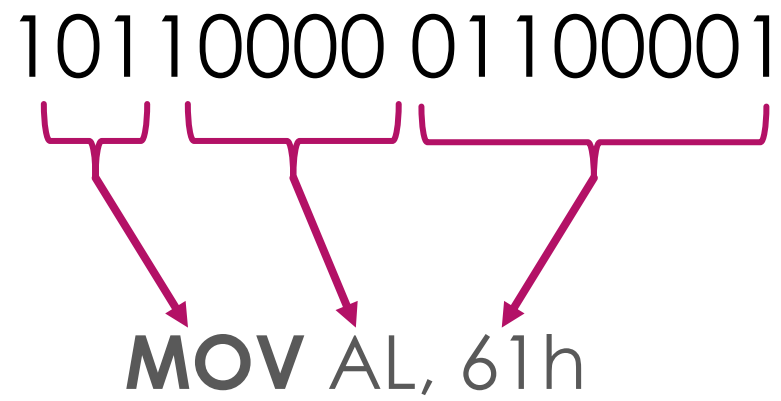
- Human Language
- "What's the result of adding 5 to 7?"

- Assembly Language

- Machine language
- 0110100101010

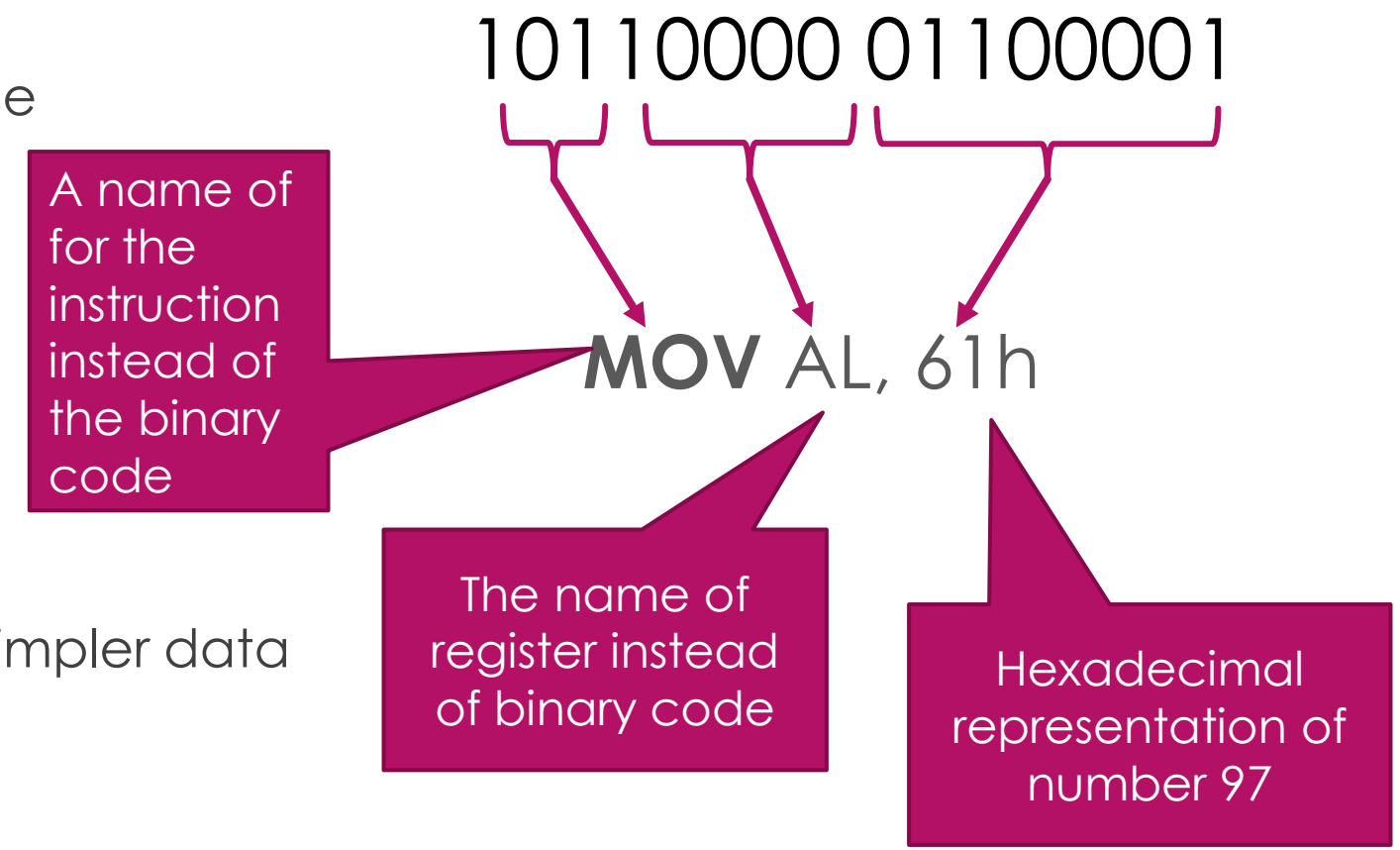
Assembly Language

- Example problem:
  - Save the number 97 for later use
    - **Data:** the constant value 97
    - **Process:** save that value
- Binary Language
  - 0's and 1's
- Assembly Language
  - Abstracting using names and simpler data representation



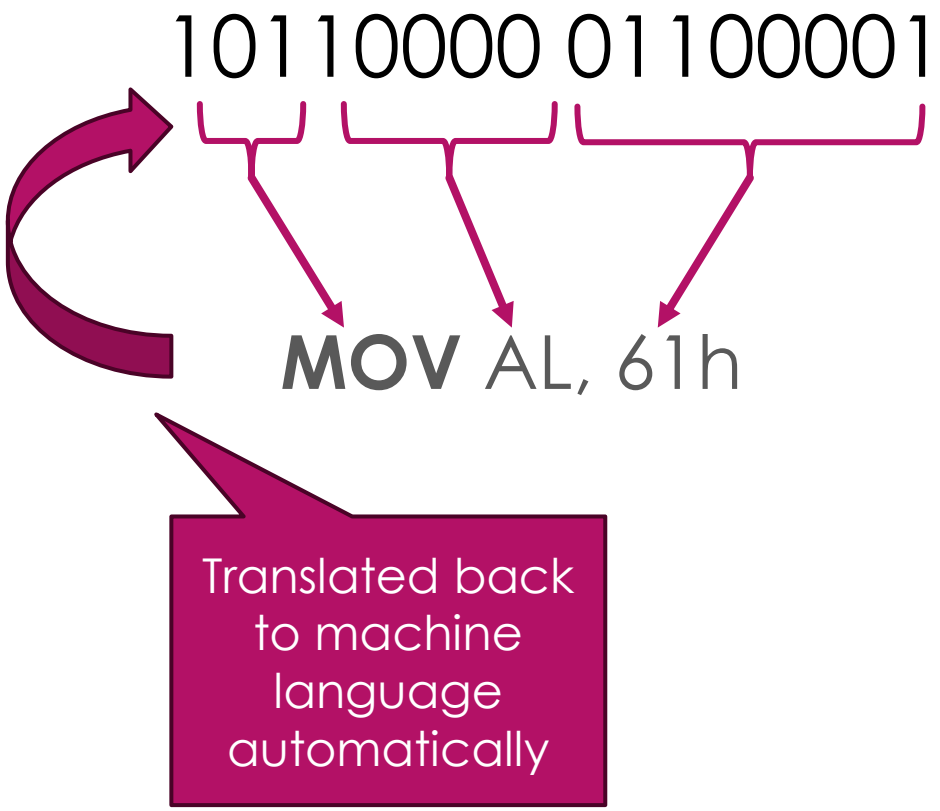
Assembly Language

- Example problem:
  - Save the number 97 for later use
    - **Data:** the constant value 97
    - **Process:** save that value
- Binary Language
  - 0's and 1's
- Assembly Language
  - **Abstracting** using names and simpler data representation



Assembly Language

- Example problem:
  - Save the number 97 for later use
    - **Data:** the constant value 97
    - **Process:** save that value
- Binary Language
  - 0's and 1's
- Assembly Language
  - Abstracting using names and simpler data representation



Even higher-level Languages

Higher level languages gradually started to emerge one after the other introducing more features that make programming easier and allow for more abstractions

- Human Language
- "What's the result of adding 5 to 7?"

- Higher-Level languages
  - E.g. C , Pascal, C++, Java, Python

- Assembly Language

- Machine language
- 0110100101010

Main Programming Constructs

Control Flow statements

- for-loop, while-loop, if-statements, case-statement

Code Blocks

- Begin-End , {}

Functions and Procedures

- Passing and returning variables

Data types

- Range of values
- Specific operations

Meaningful names

- For data holders
- For functions and procedures

Object-Oriented Features

- Classes, objects

Reusable Code

- Libraries
- Modules

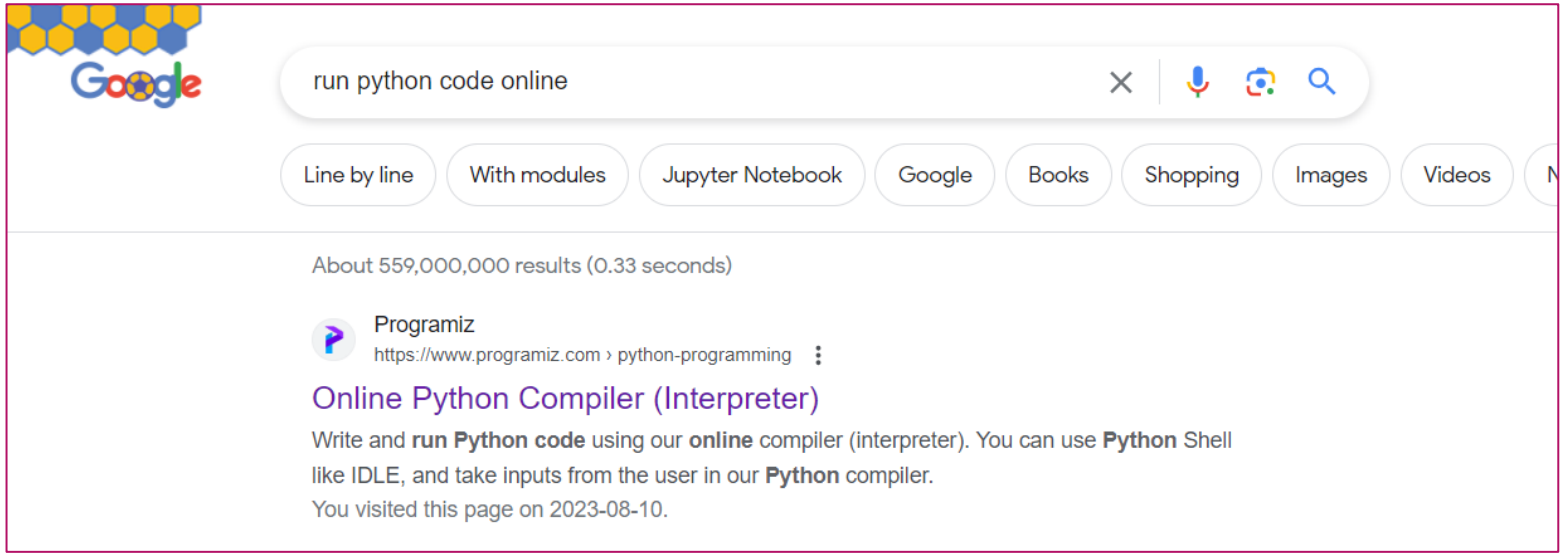
Hello Python, Worlds!

Hello Python, Worlds!

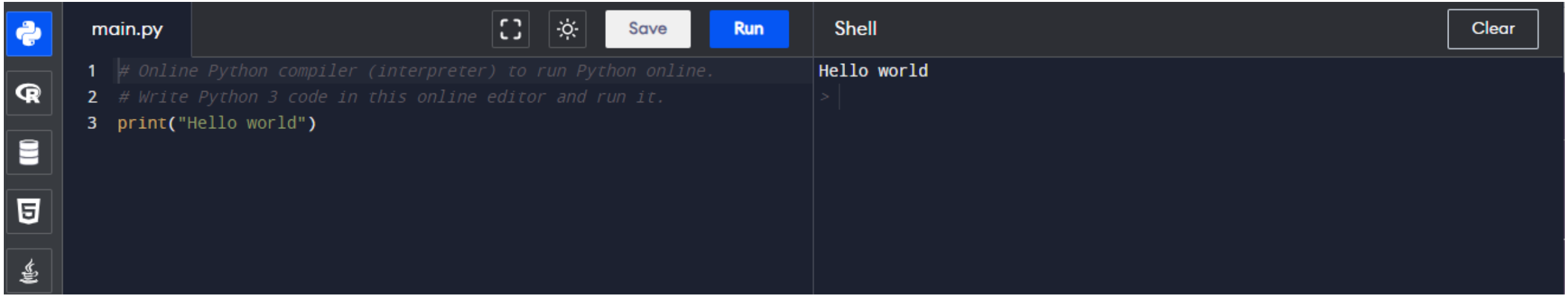
- It's a common practice when learning a new language to start by printing "Hello, World!" using that language.
- In order to execute the code you need an environment that includes the translator specific to this language + other helpful pre-written code.
- The translator of python is called "Interpreter".
- There are different ways that you can write and execute your python code
  - You can write the code in a file with the extension ".py"
  - You can write and execute part by part in a Jupyter Notebook (usually used for experimentation)

Python file

- When we work with the Pi-Cars we'll work with the interpreter directly, but for this section we just need to focus on the programing constructs so we'll use an online environment that has the interpreter already available behind the scenes.
- https://www.programiz.com/python-programming/online-compiler/

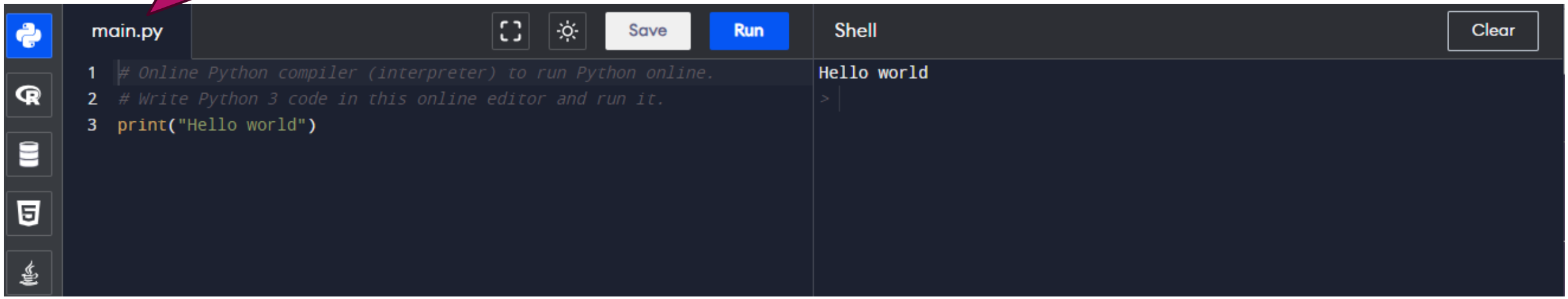


Hello, World!


main.py
Save
Run
Shell
Clear
1 # Online Python compiler (interpreter) to run Python online.
2 # Write Python 3 code in this online editor and run it.
3 print("Hello world")
Hello world
>

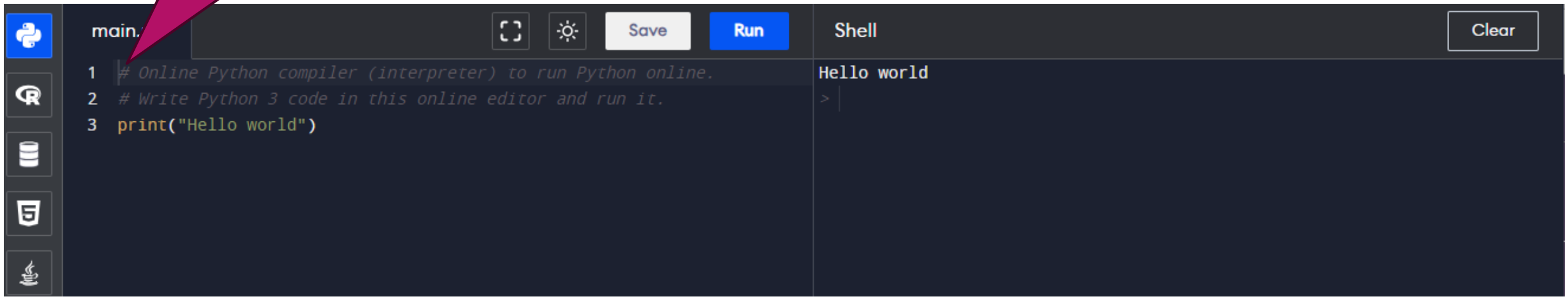
Hello, World!

You'd write your code in a file that has .py extension



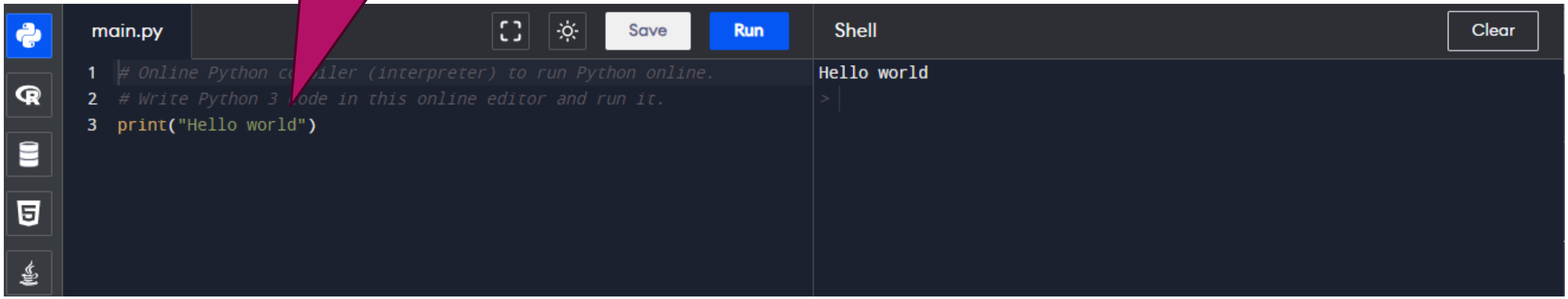
Hello, World!

A line that starts with a '#' is a comment so it doesn't need interpretation or execution



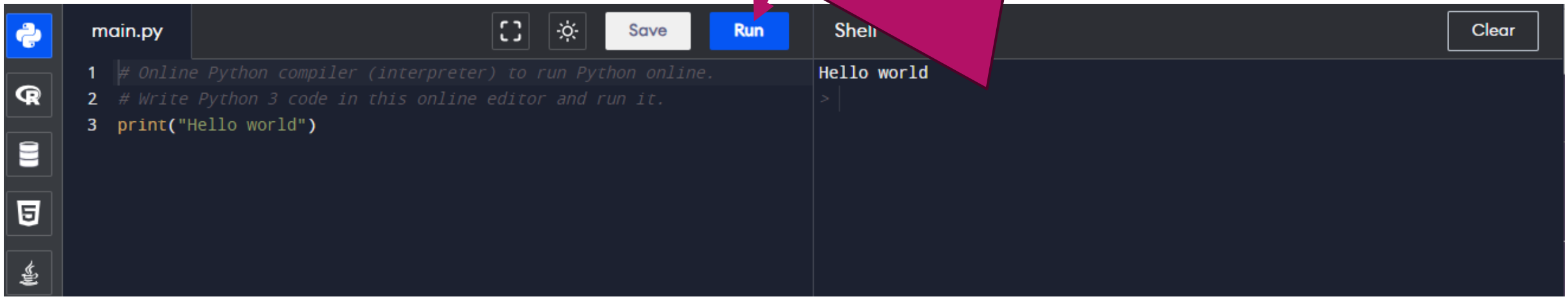
Hello, World!

This is the only line of code that we have in this script and it's a function that print the text on the screen

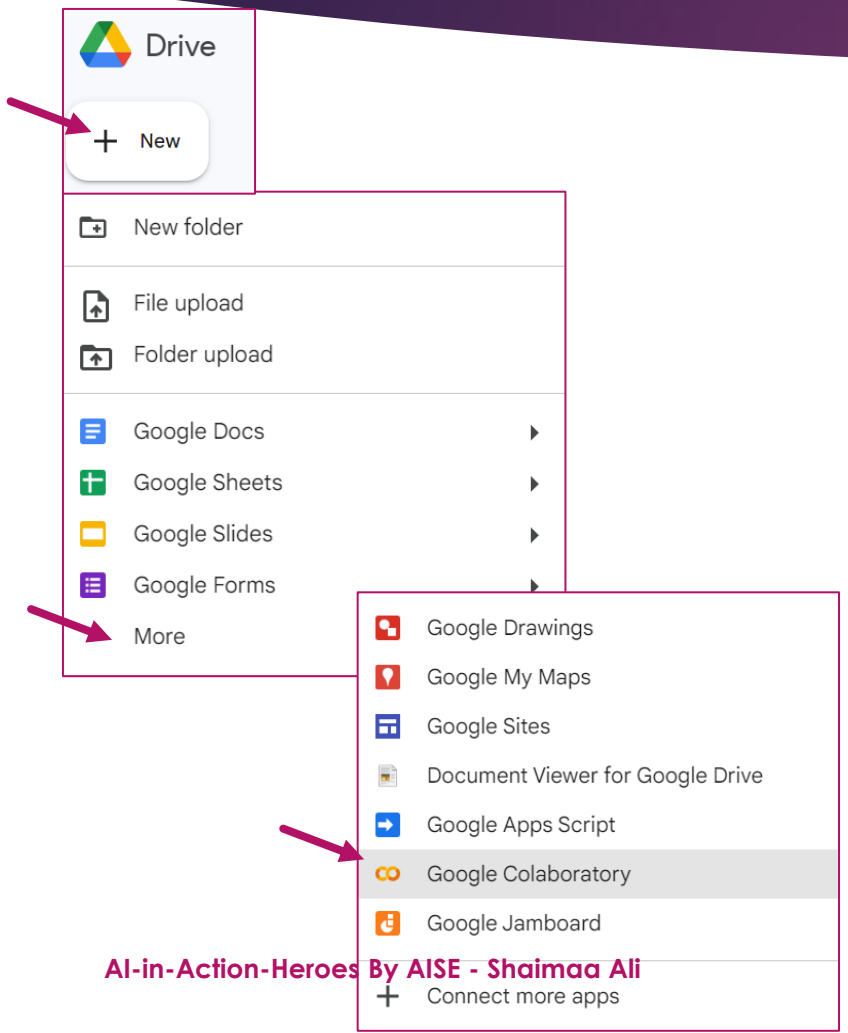


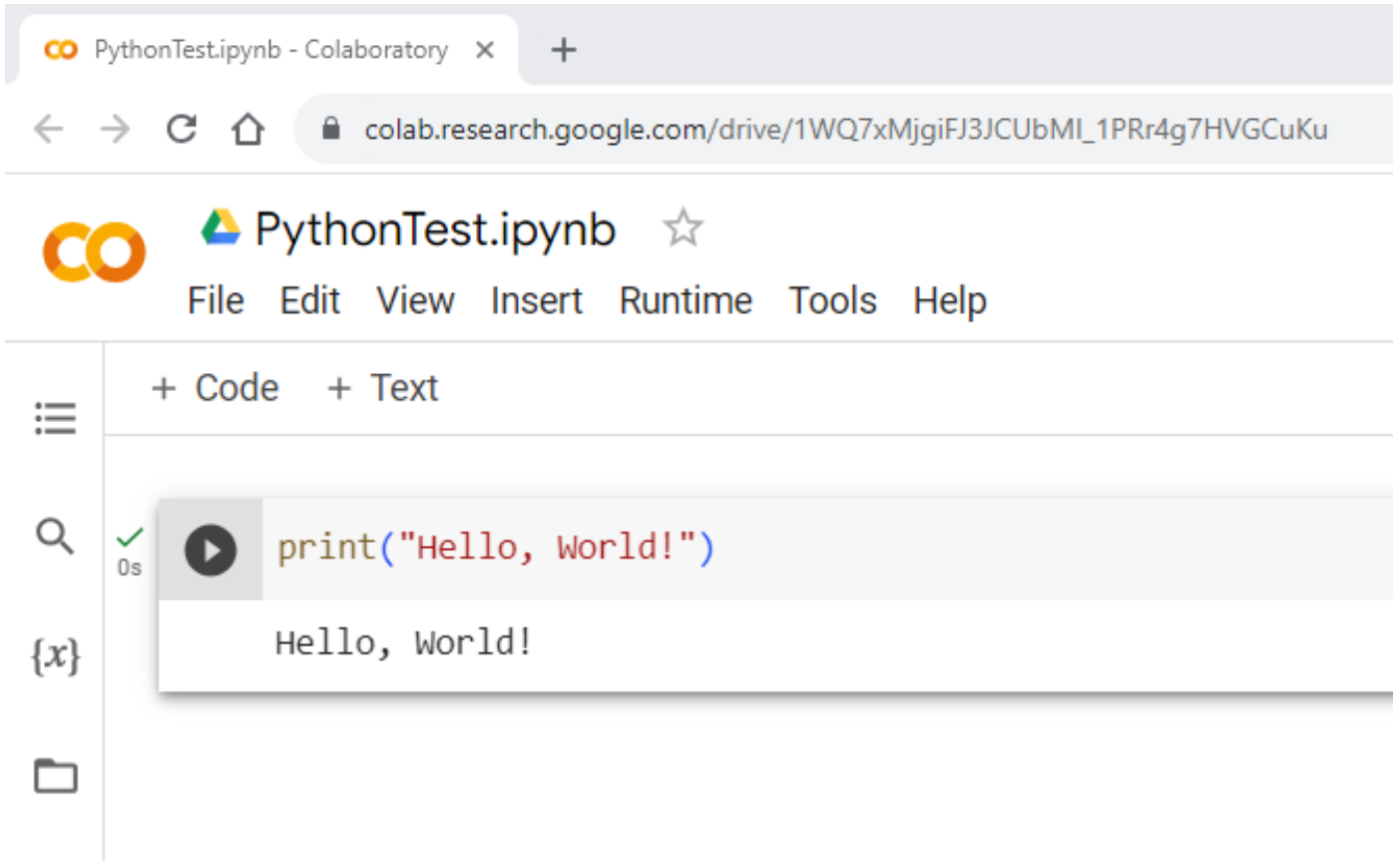
Hello, World!

When you hit 'run' the lines of code are intereprted and executed one by one and if there's any thing that should appear on this side of the screen



Jupyter Notebook – Example free environment is Google Colab





Basic Data Types & Operations

Main Built-In Data Types

NoneType
- Special value **None**

Numeric Types
- int, float

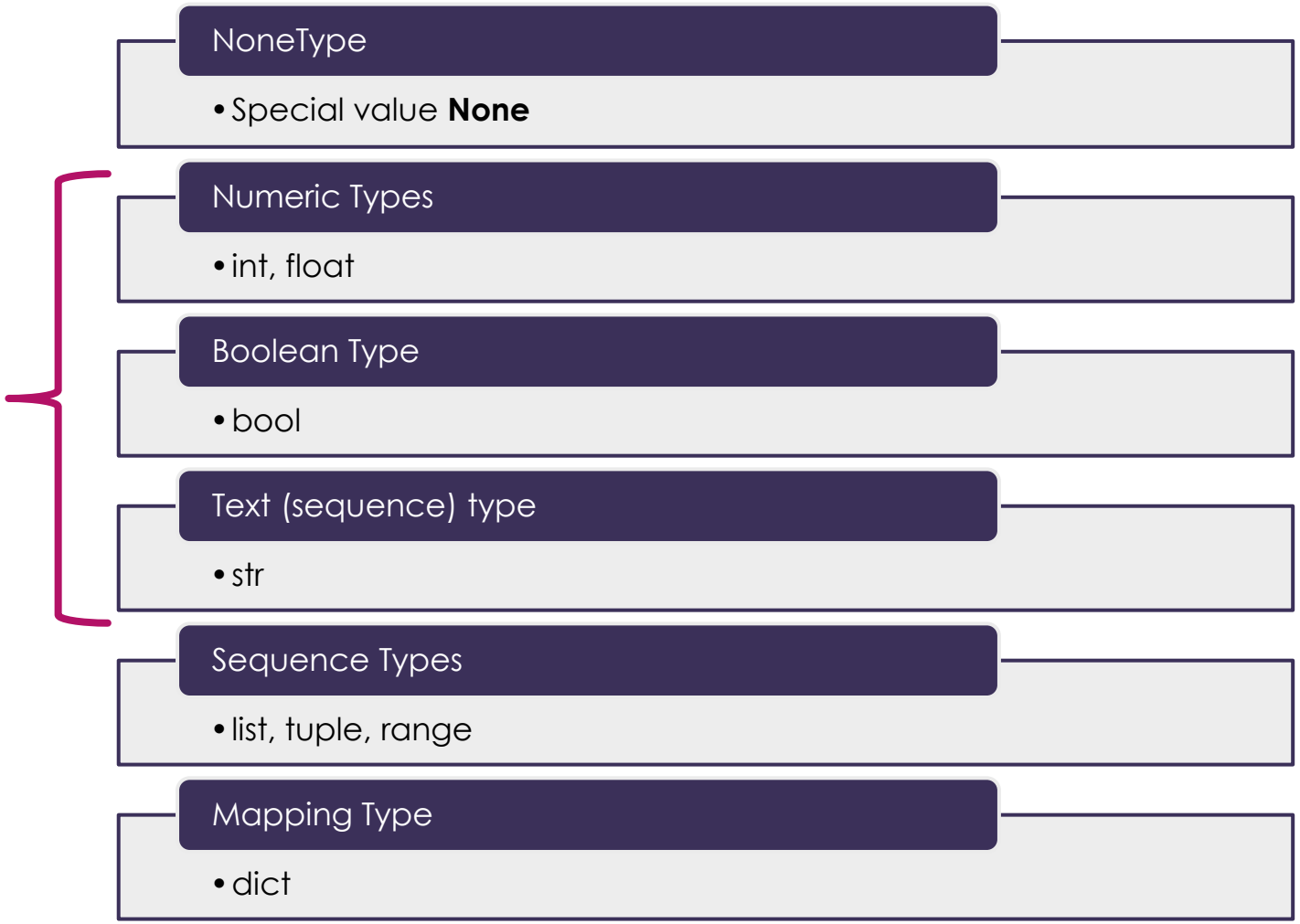
Boolean Type
- bool

Text (sequence) type
- str

Sequence Types
- list, tuple, range

Mapping Type
- dict

Main Built-In Data Types



Main Built-In Data Types

Numeric Types

- int – discrete numbers (e.g. 5)
- float – numbers with fractions (e.g. 5.5)

Boolean Type

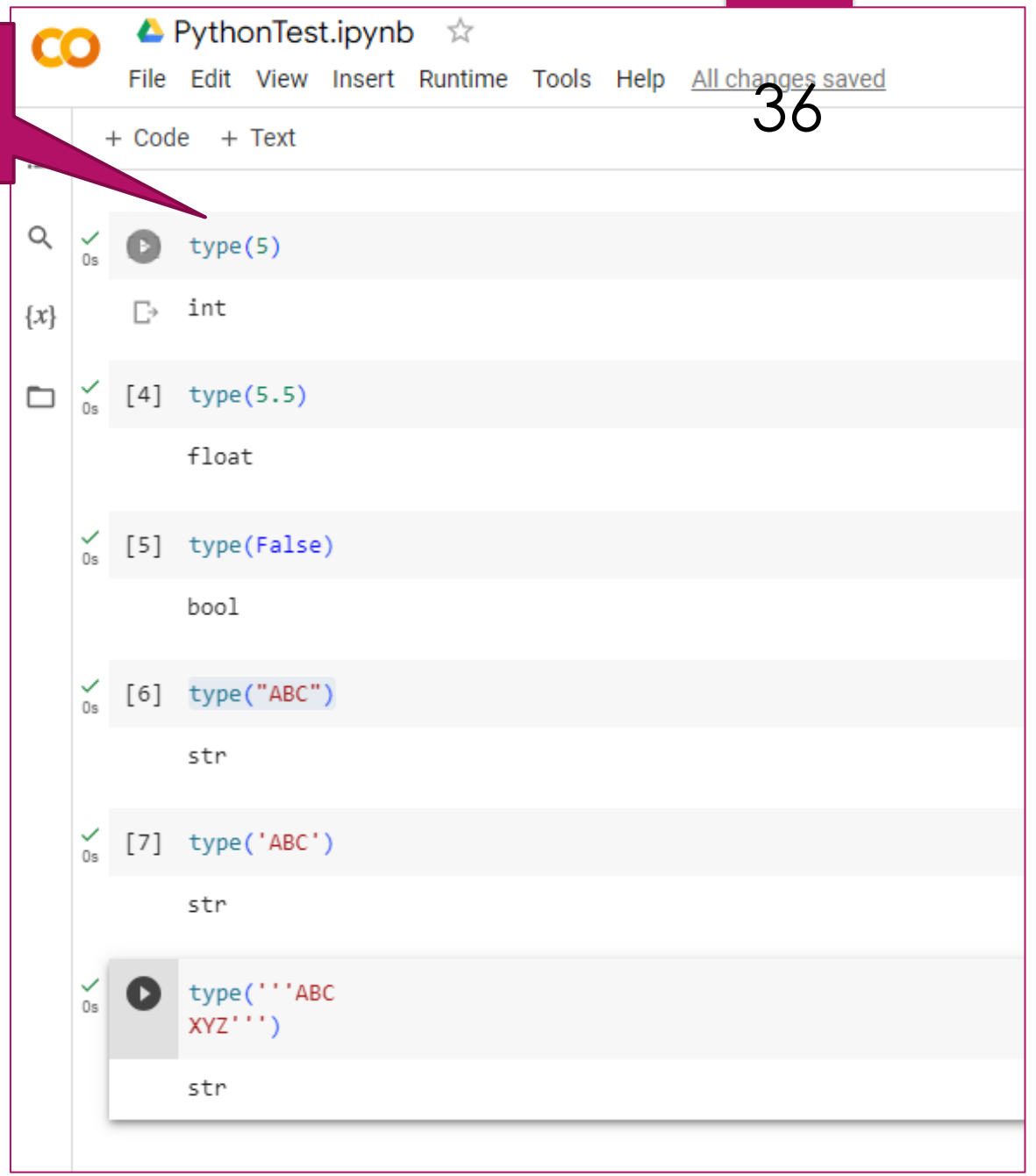
- bool - True or False (case sensitive)

Text (sequence) type

- str – Any sequence of characters surrounded by
  - Single quotes: 'allows embedded "double" quotes'
  - Double quotes: "allows embedded 'single' quotes"
  - Triple quoted: '''Three single quotes''', """Three double quotes"""

Main Built-In Data Types & Operations

A built-in function that returns the type of a given value



Numeric Types

- int – discrete numbers (e.g. 5)
- float – numbers with fractions (e.g. 5.5)

Boolean Type

- bool - True or False (case sensitive)

Text (sequence) type

- str – Any sequence of characters surrounded by
  - Single quotes: 'allows embedded "double" quotes'
  - Double quotes: "allows embedded 'single' quotes"
  - Triple quoted: '''Three single quotes''', """Three double quotes"""

Variables

- Named data holder
- We can use the assignment operatory (=) to store data value in a named variable
- Valid names
  - No special characters except the underscore (_)
  - Can contain any character or digits but cannot start with a digit
  - Cannot be a reserved word

Main Built-In Data Types & Operations

Numeric Types

- int – discrete numbers (e.g. 5)
- float – numbers with fractions (e.g. 5.5)

Boolean Type

- bool - True or False (case sensitive)

Text (sequence) type

- str – Any sequence of characters surrounded by
  - Single quotes: 'allows embedded "double" quotes'
  - Double quotes: "allows embedded 'single' quotes"
  - Triple quoted: '''Three single quotes''', """Three double quotes"""

Python operation	Arithmetic operator	Python expression
Addition	`+`	`f + 7`
Subtraction	`-`	`p - c`
Multiplication	`*`	`b * m`
Exponentiation	`**`	`x ** y`
True division	`/`	`x / y`
Floor division	`//`	`x // y`
Remainder (modulo)	`%`	`r % s`

Main Built-In Data Types & Operations

Numeric Types

- int – discrete numbers (e.g. 5)
- float – numbers with fractions (e.g. 5.5)

Boolean Type

- bool - True or False (case sensitive)

Text (sequence) type

- str – Any sequence of characters surrounded by
  - Single quotes: 'allows embedded "double" quotes'
  - Double quotes: "allows embedded 'single' quotes"
  - Triple quoted: '''Three single quotes''', """Three double quotes"""

Part of the code that results in a value

Python operation	Arithmetic operator	Python expression
Addition	`+`	`f + 7`
Subtraction	`-`	`p - c`
Multiplication	`*`	`b * m`
Exponentiation	`**`	`x ** y`
True division	`/`	`x / y`
Floor division	`//`	`x // y`
Remainder (modulo)	`%`	`r % s`

Main Built-In Data Types & Operations

Numeric Types

- int – discrete numbers (e.g. 5)
- float – numbers with fractions (e.g. 5.5)

Boolean Type

- bool - True or False (case sensitive)

Text (sequence) type

- str – Any sequence of characters surrounded by
  - Single quotes: 'allows embedded "double" quotes'
  - Double quotes: "allows embedded 'single' quotes"
  - Triple quoted: '''Three single quotes''', """Three double quotes"""

- ‘f‘ is replaced by the value stored in the variable
- The value is added to 7
- The whole expression results in (evaluates to the result of the addition)

Python operation	Arithmetic operat	ython expression
Addition	+	f + 7
Subtraction	-	p - c
Multiplication	*	b * m
Exponentiation	**	x ** y
True division	/	x / y
Floor division	//	x // y
Remainder (modulo)	%	r % s

Main Built-In Data Types & Operations

Numeric Types

- int – discrete numbers (e.g. 5)
- float – numbers with fractions (e.g. 5.5)

Boolean Type

- bool - True or False (case sensitive)

Text (sequence) type

- str – Any sequence of characters surrounded by
  - Single quotes: 'allows embedded "double" quotes'
  - Double quotes: "allows embedded 'single' quotes"
  - Triple quoted: '''Three single quotes''', """Three double quotes"""

Python operation	Arithmetic operator	Python expression
Addition	+	f + 7
Subtraction	-	p - c
Multiplication	*	b * m
Exponentiation	**	** y
True division	/	y
Floor division	//	y
Remainder (modulo)	%	

- 'p' is evaluated to the value stored in the variable
- 'c' is evaluated to the value stored in the variable
- The whole expression results in (evaluates to the result of the subtraction)

Main Built-In Data Types & Operations

Comparison operators result in a Boolean value

Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal
`is`	object identity
`is not`	negated object identity

Numeric Types

- int – discrete numbers (e.g. 5)
- float – numbers with fractions (e.g. 5.5)

Boolean Type

- bool - True or False (case sensitive)

Text (sequence) type

- str – Any sequence of characters surrounded by
  - Single quotes: 'allows embedded "double" quotes'
  - Double quotes: "allows embedded 'single' quotes"
  - Triple quoted: '''Three single quotes''', """Three double quotes"""

P	Q	P **or** Q	P **and** Q	**not** P	**not** Q
True	True	True	True	False	False
True	False	True	False	False	True
False	True	True	False	True	False
False	False	False	False	True	True

Logical operators
There are some specifics of how these logical operators work on python, we won't get into these specifics for now

Main Built-In Data Types & Operations

Numeric Types

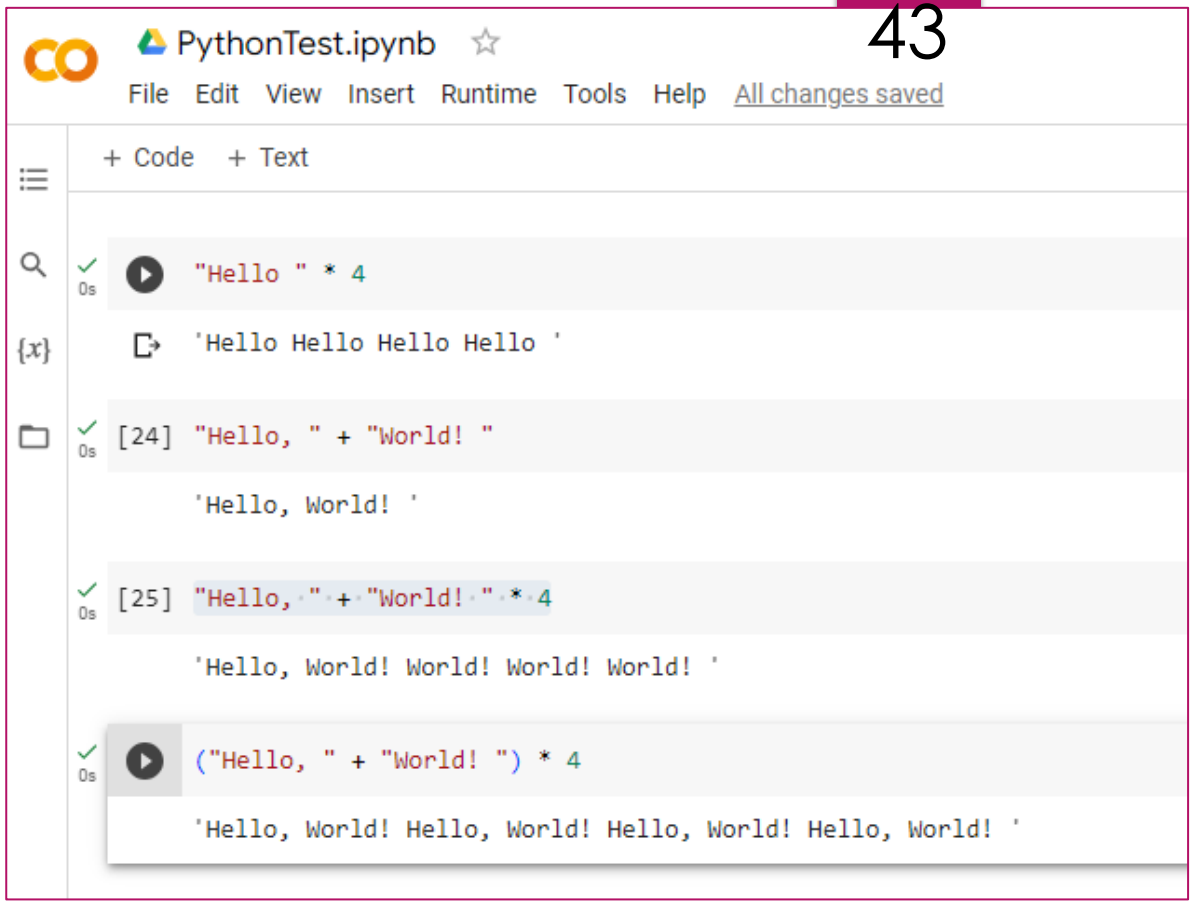
- int – discrete numbers (e.g. 5)
- float – numbers with fractions (e.g. 5.5)

Boolean Type

- bool - True or False (case sensitive)

Text (sequence) type

- str – Any sequence of characters surrounded by
  - Single quotes: 'allows embedded "double" quotes'
  - Double quotes: "allows embedded 'single' quotes"
  - Triple quoted: '''Three single quotes''', """Three double quotes"""



\+ : concatenation
* : repetition

Flow Control
–
Conditional statements

if-statement

Syntax

```
if <expr>:
    <statement>
```

```
if <expr>: <statement>
```

```
if <expr>: <statement_1>; <statement_2>; ...; <statement_n>
```

if-statement

Syntax

```
if <expr>:
    <statement>
```

```
if <expr>: <statement>
```

```
if <expr>: <statement_1>; <statement_2>; ...; <statement_n>
```

Generally,
the expression
of the if statement is called the condition and it should result in a Boolean value

if-statement

Syntax

```
if <expr>:
    <statement>
```

```
if <expr>: <statement>
```

```
grade = 80
```

```
if (grade >= 80):
    print("A")
```

```
if (grade >= 80):
    print("A");
```

A new line is enough indication of the end of the statement

```
if (grade >= 80): print("A")
```

```
if (grade >= 80): print("A");
```

```
if <expr>: <statement_1>; <statement_2>; ...; <statement_n>
```

if-statement

Syntax

```
if <expr>:
    <statement>
```

```
if <expr>: <statement>
```

```
grade = 80
```

```
if (grade >= 80):
    print("A")
```

```
if (grade >= 80):
    print("A");
```

The ; is not necessary unless we are writing multiple statements in the same line

```
if (grade >= 80): print("A")
```

```
if (grade >= 80): print("A");
```

```
if <expr>: <statement_1>; <statement_2>; ...; <statement_n>
```

if-statement

Syntax

```
if <expr>:
    <statement>
```

```
if <expr>: <statement>
```

```
grade = 80
```

```
if (grade >= 80):
    print("A")
```

```
if (grade >= 80):
    print("A");
```

Indentation indicate a block of code

```
if (grade >= 80):
print("A")
```

```
if <expr>: <statement_1>; <statement_2>; ...; <statement_n>
```

if-statement

Syntax

```
if <expr>:
    <statement>
```

```
if <expr>: <statement>
```

```
grade = 80
```

```
if (grade >= 80):
    print("You grade is:")
    print("A")
```

```
if (grade >= 80):
    print("You grade is:")
print("A")
```

No error message here as it'll assume that the second statement is not part of the block

```
if <expr>: <statement_1>; <statement_2>; ...; <statement_n>
```

if-statement

Syntax

```
if <expr>:
    <statement>
```

```
if <expr>: <statement>
```

```
grade = 80
```

```
if (grade >= 80):
    print("You grade is:")
    print("A")
```

```
if (grade >= 80):
    print("You grade is:")
print("A")
```

```
if (grade >= 80):
    print("You grade is:"); print("A")
```

```
if (grade >= 80): print("You grade is:"); print("A")
```

```
if <expr>: <statement_1>; <statement_2>; ...; <statement_n>
```

if-statement

```
if <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
    ...
else:
    <statement(s)>
```

```
grade = 55

if (grade >= 80):
    print("Excellent")
    print("A")
elif grade < 80 and grade <= 70:
    print("Very Good")
    print("B")
elif grade < 70 and grade <= 60:
    print("Good")
    print("C")
elif grade < 60 and grade <= 50:
    print("Pass")
    print("C")
else:
    print("F")
    print("Please try again!")
```

```
grade = 55

if (grade >= 80): print("Excellent"); print("A");
elif grade < 80 and grade <= 70: print("Very Good"); print("B");
elif grade < 70 and grade <= 60: print("Good"); print("C");
elif grade < 60 and grade <= 50: print("Pass"); print("C");
else: print("F"); print("Please try again!");
```

Hands-ON

Sequence Types & operations

Main Built-In Data Types

NoneType
- Special value **None**

Numeric Types
- int, float

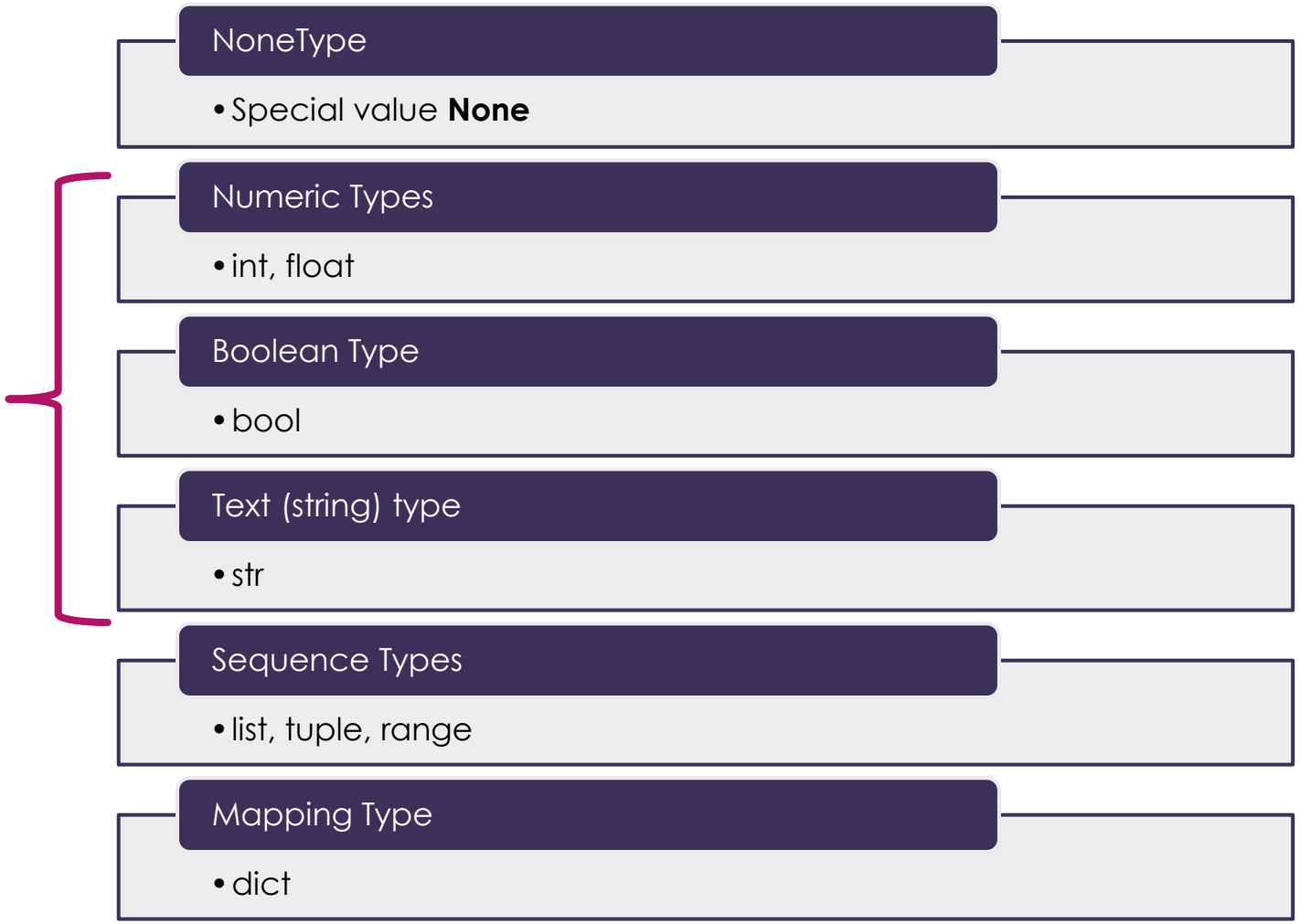
Boolean Type
- bool

Text (string) type
- str

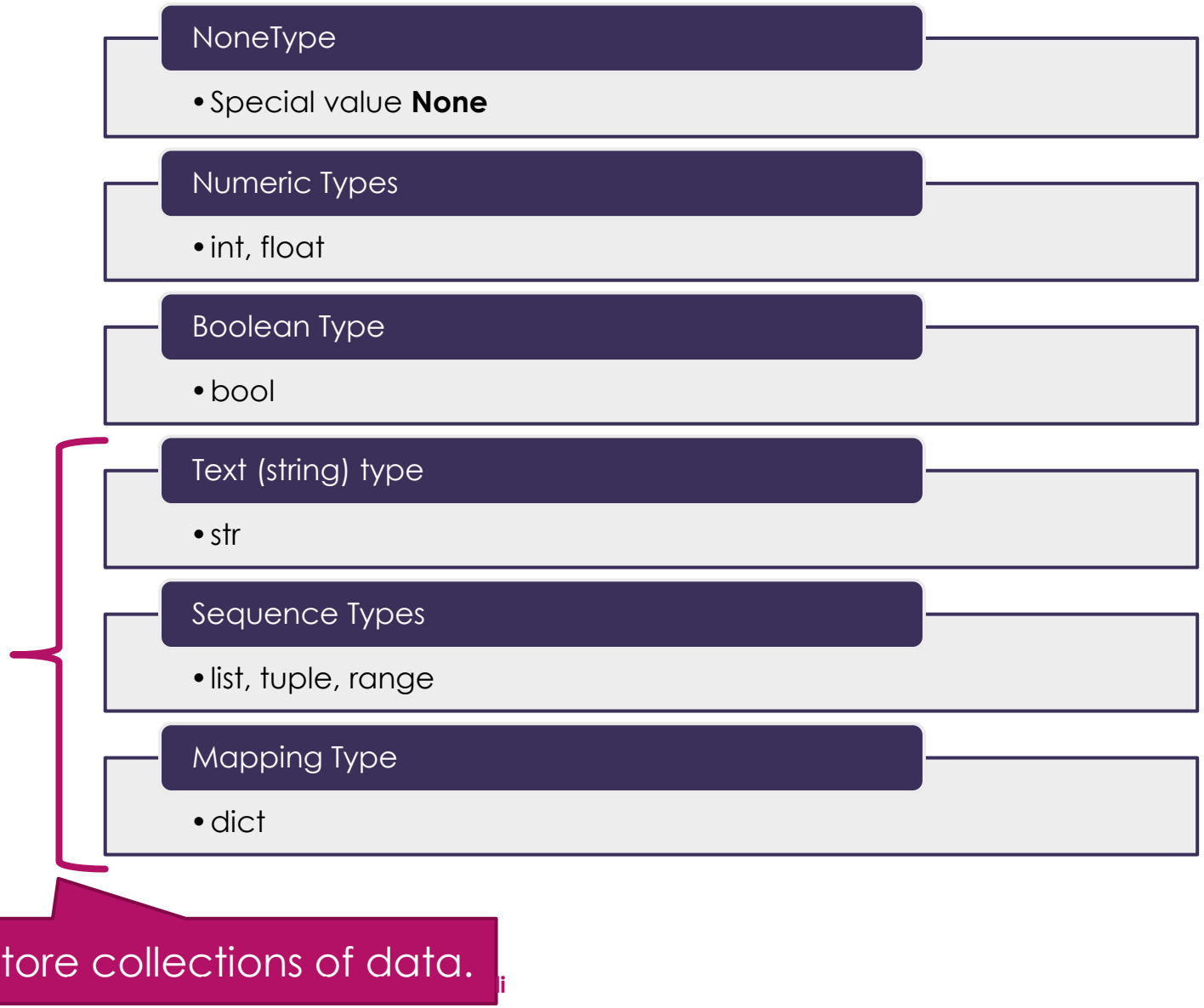
Sequence Types
- list, tuple, range

Mapping Type
- dict

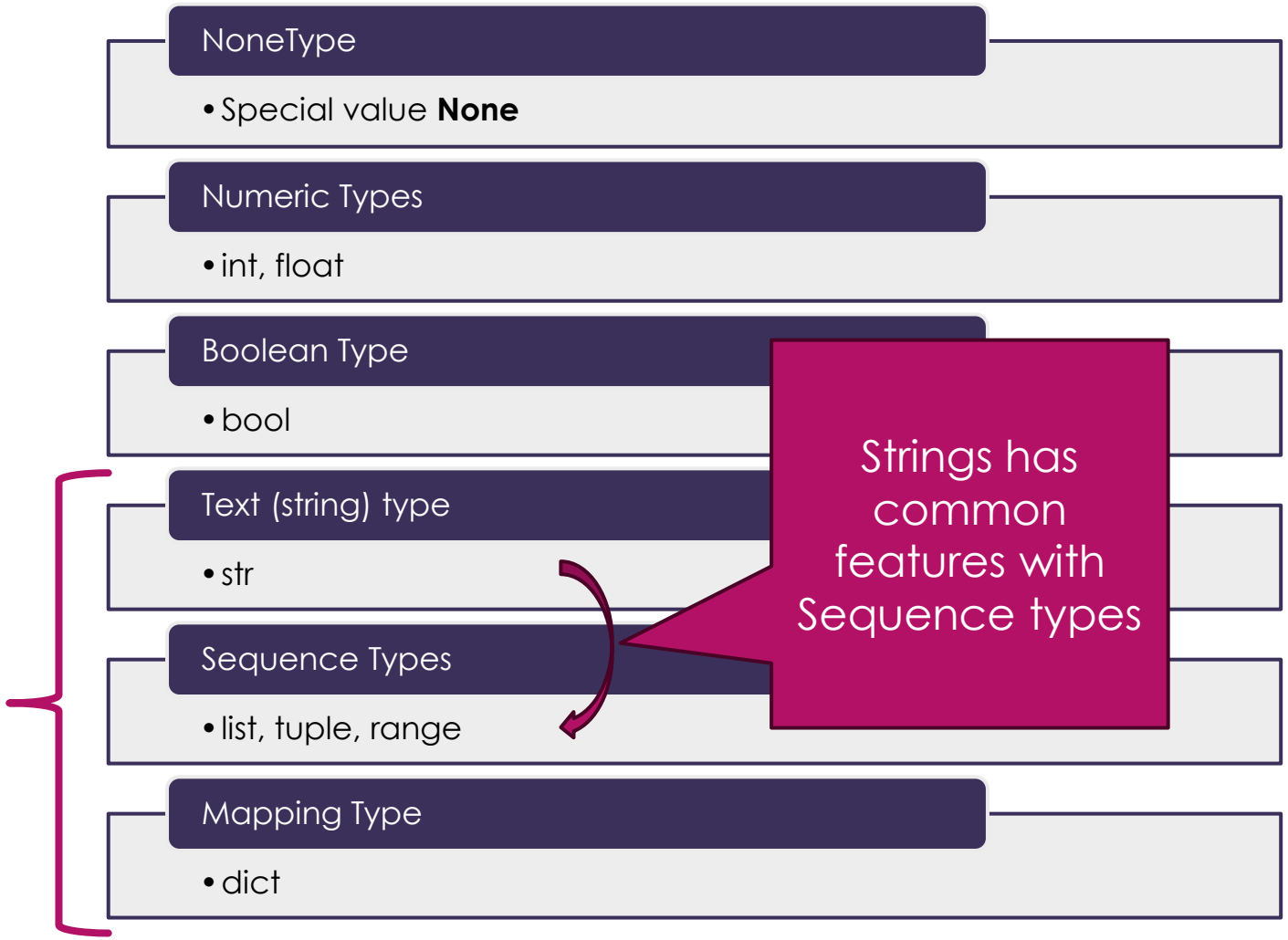
Main Built-In Data Types



Main Built-In Data Types



Main Built-In Data Types



Sequence Types

- Lists
- Tuples
- Strings
- Ranges

Common Sequence Operations

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Sequence Types

- Lists
- Tuples
- Strings
- Ranges

Common Sequence Operations

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	
`max(s)`	
`s.index(x[, i[, j]])`	rence of *x* in *s* (at or after x *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Elements of the sequence types are numbered starting from 0, this number is called the **index**

Sequence Types

- Lists
- Tuples
- Strings
- Ranges

Common Sequence Operations

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Sequence Types

Lists

- A list is a mutable sequence of values, surrounded by square brackets and separated by commas.
- Elements are ordered and accessible by a zero-based index

Tuples

Strings

Ranges

```
In [1]: mylist = [5, "abc", True, [3,5,1]]

In [2]: mylist[5]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-2-30998d85754d> in <module>
----> 1 mylist[5]

IndexError: list index out of range

In [3]: mylist[2]
Out[3]: True

In [4]: mylist[3]
Out[4]: [3, 5, 1]

In [5]: mylist[3][1]
Out[5]: 5

In [6]: mylist[0] = 10

In [7]: mylist
Out[7]: [10, 'abc', True, [3, 5, 1]]

In [8]:
```

Sequence Types

- Lists
  - A list is a mutable sequence of values, surrounded by square brackets and separated by commas.
  - Elements are ordered and accessible by a zero-based index
- Tuples
- Strings
- Ranges

Common Sequence Operations

Operation	
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Hands On

Sequence Types

- Lists
- Tuples
  - A tuple is a sequence of comma separated elements
  - Maybe surrounded by round parentheses
    - The parentheses are only required for an empty tuple
  - Sigle element tuples must have a trailing comma
- Strings
- Ranges

```
In [1]: t1 = ()

In [2]: t2 = 5,

In [3]: t3 = (5,)

In [4]: nt3 = (5)

In [5]: t4 = 7,"abc", True, ()

In [6]: t5 = (7,"abc", True, ())
```

```
In [7]: type(t1)
Out[7]: tuple

In [8]: type(t2)
Out[8]: tuple

In [9]: type(t3)
Out[9]: tuple

In [10]: type(nt3)
Out[10]: int

In [11]: type(t4)
Out[11]: tuple
```

Sequence Types

- Lists
- Tuples
  - A tuple is an immutable sequence of comma separated elements
  - Maybe surrounded by round parentheses
    - The parentheses are only required for an empty tuple
  - Sigle element tuples must have a trailing comma
- Strings
- Ranges

Common Sequence Operations

Operation	
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Hands On

Sequence Types

- Lists
- Tuples
- Strings
  - Special type of sequences
  - Elements surrounded by quotes and NOT separated by commas
  - Elements are Unicode characters
- Ranges

Common Sequence Operations

Operation	
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

Hands On

Sequence Types

- Lists
- Tuples
- Strings
- Ranges

  - The range type represents an immutable sequence of **numbers** and is commonly used for looping a specific number of times in for loops.
  - implement all of the common sequence operations except concatenation and repetition to avoid violating the generation pattern.

```
class range(stop)
class range(start, stop[, step])
```

Starting value (default is 0)

increment value (default is 1)

```
In [1]: r = range(10)

In [2]: list(r)
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: 5 in r
Out[3]: True

In [4]: even = range(0,10,2)

In [5]: 5 in even
Out[5]: False

In [6]: odd = range(1, 10, 2)

In [7]: 5 in odd
Out[7]: True

In [8]: len(r)
Out[8]: 10
```

```
In [10]: tuple(r)
Out[10]: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

In [11]: str(r)
Out[11]: 'range(0, 10)'
```

Flow Control – Iterations / Loops

Loops

- Are used to repeat executing a block of code.
- A loop has three main components in addition to the block to execute
  - Initialization (starting point)
  - Stopping criteria
  - An update that move towards the stopping criteria
- There are two main types of loops
  - A loop based on specified number of iterations (**for** loop)
  - A loop based on a condition (**while** loop)

while loop
Iterating based on a condition

```
i = 0
while i < 5:
  i += 1
  print(i)
```

```
1
2
3
4
5
```

while loop
Iterating based on a condition

keyword

```
i = 0
while i < 5:
  i += 1
  print(i)
```

```
1
2
3
4
5
```

while loop
Iterating based on a condition

```
i = 0
while i < 5:
  i += 1
  print(i)
```

```
1
2
3
4
5
```

To indicate that there's a sub-block coming

while loop
Iterating based on a condition

sub-block with consistent indentation

```
i = 0
while i < 5:
    i += 1
    print(i)
```

```
1
2
3
4
5
```

while loop
Iterating based on a condition

```
i = 0
while i < 5:
  i += 1
  print(i)
```

```
1
2
3
4
5
```

while loop
Iterating based on a condition

Initialization / starting point

```
i = 0
while i < 5:
  i += 1
  print(i)
```

```
1
2
3
4
5
```

while loop
Iterating based on a condition

```
i = 0
while i < 5:
  i += 1
  print(i)
```

```
1
2
3
4
5
```

Condition to determine the stopping criteria (will stop when the condition is False)

while loop
Iterating based on a condition

```
i = 0
while i < 5:
  i += 1
  print(i)
```

```
1
2
3
4
5
```

```
i = 0
while i < 5:
  i += 1
  if i == 3:
    continue
  print(i)
```

```
1
2
4
5
```

Skips the rest of the block and go to the next iteration

```
i = 0
while i < 5:
  i += 1
  if i == 3:
    break
  print(i)
```

```
1
2
```

Exists the entire loop

for loops

- In python a for loop iterates over the elements of a given sequence
- If we need to specify a number of iterations we can use a range.

```
[9]  for i in range(1,6):
       print(i)

     1
     2
     3
     4
     5
```

```
fruits = ["apple", "banana","cherry"]
for x in fruits:
    if x == "banana":
      continue
    print(x)

apple
cherry
```

```
for i in (2,4,6):
  print(i)

2
4
6
```

Functions

Functions

- A function is a block of code that has
  - Name
    - Used to call the function for execution
  - parameter list
    - To receive arguments to apply the code on
  - Optionally return value
    - Using the 'return' keywords

Functions

Function definition

Function call

```
def print_name(first_name, last_name):
    full_name = first_name + " " + last_name
    print(full_name)

print_name("Shaimaa", "Ali")
```

```
Shaimaa Ali
```

Functions

keyword

Function definition

```
def print_name(first_name, last_name):
    full_name = first_name + " " + last_name
    print(full_name)

print_name("Shaimaa", "Ali")
```

```
Shaimaa Ali
```

Functions

A name that we choose

Function definition

```
def print_name(first_name, last_name):
    full_name = first_name + " " + last_name
    print(full_name)

print_name("Shaimaa", "Ali")
```

```
Shaimaa Ali
```

Functions

Whether we want the functions to receive parameters or not the parentheses are mandatory

Function definition

```
def print_name(first_name, last_name):
    full_name = first_name + " " + last_name
    print(full_name)

print_name("Shaimaa", "Ali")
```

```
Shaimaa Ali
```

Functions

Function call

```
def print_name(first_name, last_name):
  full_name = first_name + " " + last_name
  print(full_name)

print_name("Shaimaa", "Ali")
```

```
Shaimaa Ali
```

Functions

```
def add (a, b):
    c = a + b
    return c

result = add(5, 7)
print(result)
```