



**Hochschule  
Bonn-Rhein-Sieg**  
University of Applied Sciences

# Introduction to Python Programming

Foundation Course @H-BRS, March 2024  
By: Vedika Chauhan(MAS)

# What is Python?

Python is a high-level, interpreted programming language known for its simplicity and readability.

1. High-level: language abstracts away most of the complex details of the computer hardware
2. Interpreted: code is executed “directly”, line-by-line, by an interpreter

# What makes Python popular?

1. Python is simple to understand and write
2. Rich ecosystem of libraries and frameworks

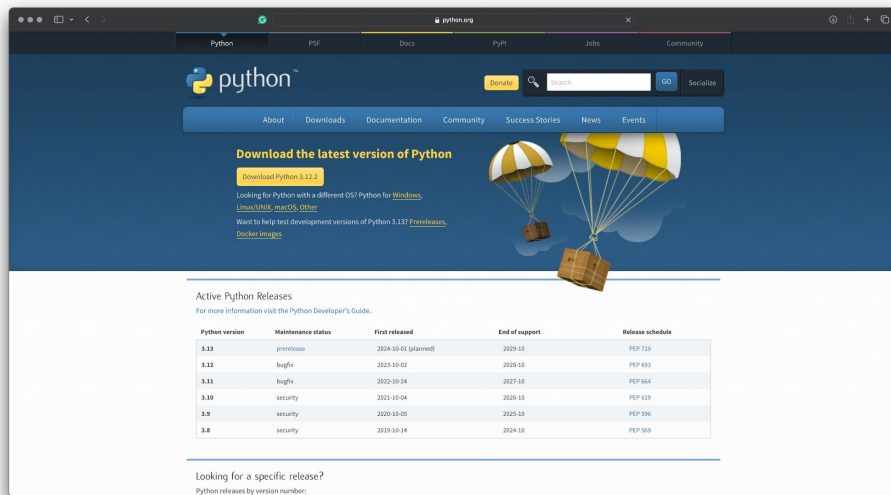
For us, this is important because:

1. Almost all machine learning libraries are developed in Python
2. For robotics, it interfaces well with ROS2
3. Many mathematics problems are best solved with a language like Python

How to get started?

# Installation

You should have Python already installed on your computer.



# Documentation

Python has amazing [official language documentation](#), and plenty of online resources (such as Stack Overflow).

# Interactive mode

1. Interactive mode allows you to execute Python commands one at a time in a command line interface/shell
2. Type Python code straight into the terminal, see results immediately
3. Great for experimentation, debugging, learning the language

You can invoke it like this:

```
$ python3
```

Let's open our terminals



# Using an IDE

For your future studies/work, make sure you have a preferred text editor/IDE installed, and that you are familiar with it.

Some examples include:

1. VSCode
2. PyCharm

# Jupyter notebooks

Your course assignments will come in form of Jupyter notebooks

1. Jupyter notebooks are documents that contain live code/equations/visualizations/narrative text
2. Python is one of the supported languages
3. They are interactive (you can execute code in individual “cells”)

# Basics of Python

# Our first Python code

As the tradition goes, here's our first Python code:

```
>>> print("Hello world!")
```

# Variables and data types

```
>>> age = 20
>>> temperature = 18.2
>>> name = 'Alice'
>>> studies_at_hbrs = True
```

These variables are examples of the basic data types:

1. Integer: whole number without a decimal point)
2. Float: numbers with decimal point
3. String: sequence of characters in single or double quotes
4. Boolean: represents a binary true/false value

# Operations and expressions



```
# Addition
sum = 5 + 3
print("5 + 3 =", sum)

# Subtraction
difference = 10 - 2
print("10 - 2 =", difference)

# Multiplication
product = 4 * 7
print("4 * 7 =", product)

# Division
quotient = 20 / 4
print("20 / 4 =", quotient)

# Floor Division (discards the fractional part)
floor_division = 17 // 3
print("17 // 3 =", floor_division)

# Modulus (remainder of the division)
remainder = 18 % 7
print("18 % 7 =", remainder)

# Exponentiation (power)
power = 2 ** 3
print("2 ** 3 =", power)
```

# Control flow

Thus far, we've executed a bunch of statements in top-down order. What if you need to make some decisions? Perhaps repeat an operation multiple times?

Let's go through the three control flow statements in Python next.





```
# IF statement
```

```
age = 18
```

```
if age >= 18:
```

```
    print("You are an adult.")
```



```
# WHILE statement  
count = 0  
while count < 5:  
    print("Count:", count)  
    count += 1
```



```
# FOR loop  
fruits = ["apple", "banana", "cherry"]  
for fruit in fruits:  
    print("Fruit:", fruit)
```

# Control flow cont.

As part of control flow statements, you should be familiar with `break` and `continue` keywords.



```
# BREAK statement
# Use a for loop to find and print the first positive number in a list
numbers = [-3, -2, -1, 0, 1, 2, 3]
for number in numbers:
    if number > 0:
        print("The first positive number is:", number)
        break # Exits the loop

# CONTINUE statement
# Print only odd numbers from 0 to 10
for number in range(11):
    if number % 2 == 0:
        continue # Skips the rest of the loop and continues with the next iteration
    print("Odd number:", number)
```

# Functions

Function is a block of reusable code that performs a specific task.

Why would you use them?

1. Organize code into manageable sections
2. Improve code readability
3. Facilitate code reuse
4. Reduce redundancy



```
# Simple function
def greet():
    print("Hello, welcome to Python programming!")

greet()

# Function with parameters and a return statement
def add_numbers(number1, number2):
    sum = number1 + number2
    return sum

result = add_numbers(5, 3)
print("The sum is:", result)

# Function with default function arguments
def greet(name="User"):
    print("Hello,", name + "! Welcome to Python programming.")

greet("Alice")
greet()
```

# Data structures

Besides the basic data types we have seen before, Python offers the following data structures (which are used all the time):

1. Lists
2. Sets
3. Tuple
4. Dictionary





```
# Lists: ordered, mutable, allows duplicates
```

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

```
fruits.append("orange") # Adding an item
```

```
print("List:", fruits)
```

```
# Sets: unordered, mutable, no duplicates
```

```
unique_numbers = {1, 2, 3, 4, 4}
```

```
unique_numbers.add(5) # Adding an item, duplicates not added
```

```
print("Set:", unique_numbers)
```

```
# Tuples: ordered, immutable, allows duplicates
```

```
coordinates = (10.0, 20.0)
```

```
print("Tuple:", coordinates)
```

```
# Dictionaries: key-value pairs, unordered, mutable
```

```
person = {"name": "Alice", "age": 25}
```

```
person["location"] = "Wonderland" # Adding a new key-value pair
```

```
print("Dictionary:", person)
```



```
# List comprehension
squares = [x**2 for x in range(10)]
print("List Comprehension:", squares)

# Some cool set operations
set_a = {1, 2, 3}
set_b = {3, 4, 5}
union_set = set_a | set_b # Union of sets
print("Set Union:", union_set)

# Tuple unpacking
x, y = coordinates
print("Tuple Unpacking:", x, y)

# Dictionary access and iteration
for key, value in person.items():
    print(f"{key}: {value}")
```

# Object-oriented programming in Python

# What is object-oriented programming?

Object-oriented programming (OOP) is a paradigm based on the concept of “objects”. These objects can contain data (**attributes**) and instructions (**methods**).

Important to know the difference between a class and an object:

1. Class: blueprint for creating objects, they define attributes and methods
2. Object: **instance** of a class

# Importance of OOP

Many of your assignments will assume basic understanding of OOP.

Moreover, you will be working with ROS2 a lot. Creating subscriber/publisher nodes will demand understanding of OOP.



```
# Everyone uses animals and vehicles, but we'll be cool instead  
# This is how you define a class
```

```
class Robot:
```

```
    # __init__ is a constructor
```

```
    def __init__(self, name, type):
```

```
        # Initialize attributes name and type
```

```
        self.name = name
```

```
        self.type = type
```

```
    # Example of a method, note the 'self' parameter!
```

```
    def introduce(self):
```

```
        return f"I am {self.name} of type {self.type}."
```

```
my_robot = Robot("Robbie", "Explorer")
```

```
print(my_robot.introduce())
```

# What's next in OOP?

If you're coming from a computer science background, you will most likely be familiar with the following four pillars of OOP:

1. Abstractions
2. Inheritance
3. Encapsulation
4. Polymorphism

If you're not, don't worry: learn by building!

# In Python, everything is an object

All elements in Python, from basic data types to more complex structures are treated as objects.

1. Uniformity: every item represented as an object simplifies the language syntax
2. Ability to have (helper) attributes/methods associated with it
3. Improves extensibility and consistency





```
>>> age = 25
```

```
>>> age.
```

age.as_integer_ratio( )	age.denominator	age.numerator
age.bit_count( )	age.from_bytes(	age.real
age.bit_length( )	age.imag	age.to_bytes(
age.conjugate(	age.is_integer( )	



```
>>> name = 'Alice'
```

```
>>> name.
```

name.capitalize()	name.isalpha()	name.ljust()	name.rsplit()
name.casefold()	name.isascii()	name.lower()	name.rstrip()
name.center()	name.isdecimal()	name.lstrip()	name.split()
name.count()	name.isdigit()	name.maketrans()	name.splitlines()
name.encode()	name.isidentifier()	name.partition()	name.startswith()
name.endswith()	name.islower()	name.removeprefix()	name.strip()
name.expandtabs()	name.isnumeric()	name.removesuffix()	name.swapcase()
name.find()	name.isprintable()	name.replace()	name.title()
name.format()	name.isspace()	name.rfind()	name.translate()
name.format_map()	name.istitle()	name.rindex()	name.upper()
name.index()	name.isupper()	name.rjust()	name.zfill()
name.isalnum()	name.join()	name.rpartition()	



```
>>> name = 'alice'  
>>> name_array = [char for char in name]  
>>> name_array.reverse()  
>>> '👏'.join(name_array)  
'e👏c👏i👏l👏a'
```

Good to know

# Modules and the standard library

Do not reinvent the wheel, if you don't need to.

Modules:

1. Python file containing definitions/functions/statements
2. Designed to organize Python code for better readability and reusability
3. Can be imported, and divided into separate namespaces

Python standard library is a collection of modules included with Python.

# More on Python standard library

In the Python standard library, you can find modules for:

1. File I/O
2. System calls
3. Sockets
4. Mathematical functions (math)
5. Operating system interface (os)
6. ...



```
>>> import os
>>> os.getcwd( )
'/Users/username/path/to/current/working/directory'
>>> os.cpu_count( )
10
```



```
>>> import math
>>> math.pi
3.141592653589793
>>> math.pi * math.pow(4, 2)
50.26548245743669
>>> math.cos(2 * math.pi)
1.0
```



# I/O

I/O (Input/Output) operations allow programs to interact with the filesystem, networks, and other devices.

Next demo is going to be exciting!



```
import subprocess
```

```
# Execute a Bash command: echo "Hey there from a file!" > example.txt
```

```
subprocess.run(  
    ['echo', 'Hey there from a file!'],  
    stdout=open('example.txt', 'w'),  
)
```

```
# Open a file and append a string to the end of it
```

```
with open('example.txt', 'a') as file:  
    file.write("Hello from Python!\n")
```

```
# Execute a Bash command: cat example.txt
```

```
subprocess.run(['cat', 'example.txt'])
```

# Exceptions and exception handling

Exception get “raised” when there’s a irreconcilable problem in the code. Exception handling is a mechanism for gracefully responding/recovering from errors during program’s execution.

```
try:  
    # Code that might raise an exception  
except ExceptionType:  
    # Code that runs if an exception of ExceptionType occurs  
finally:  
    # Code that runs no matter what
```

# Writing documentation


Why would you write documentation?

1. For others to understand what you did
2. For yourself to remember what you did

Documentation comes in form of simple code comments or docstrings.

How to write good documentation?

1. Keep documentation updates as code evolves
2. Make is accessible for your audience
3. Use clear, concise language
4. Include examples if applicable



```
class Robot:
    """
    Represents a basic robot with a name and type. This class provides
    foundational attributes and methods for different kinds of robots.

    Attributes:
        name (str): The name of the robot.
        type (str): The type/category of the robot (e.g., 'Explorer', 'Assistant').

    Methods:
        introduce(self): Prints a message introducing the robot.
    """

    def __init__(self, name, type):
        """
        Initializes a new Robot instance with a name and type.

        Parameters:
            name (str): The name of the robot.
            type (str): The type/category of the robot.
        """
        self.name = name
        self.type = type

    def introduce(self):
        """Prints an introduction message mentioning the robot's name and type."""
        print(f"I am {self.name} of type {self.type}.")
```

# Type hinting


Type hints in Python are a way to explicitly indicate the data type of variables, parameters, and return values within your code.

*But Python is a dynamically-typed language?*

Yes, type hints are only used by static type checkers (in your IDE, for example), linters, and similar software to identify issues before runtime.

# Benefits of type hinting your code

1. Improve code readability and clarity
2. Facilitate better development practices by making type expectations clear
3. Helps static type checkers verify type correctness



```
class Robot:
    """
    Represents a basic robot with a name and type. This class provides
    foundational attributes and methods for different kinds of robots.

    Attributes:
        name (str): The name of the robot.
        type (str): The type/category of the robot (e.g., 'Explorer', 'Assistant').

    Methods:
        introduce(self): Prints a message introducing the robot.
    """

    def __init__(self, name: str, type: str) -> None:
        """
        Initializes a new Robot instance with a name and type.

        Parameters:
            name (str): The name of the robot.
            type (str): The type/category of the robot.
        """
        self.name = name
        self.type = type

    def introduce(self) -> None:
        """Prints an introduction message mentioning the robot's name and type."""
        print(f"I am {self.name} of type {self.type}.")
```



Any questions?