Introduction to Python

Denis Oluka
Software Developer
Interswitch East Africa (U) Ltd

About the Course

Interactive lectures with real-time coding demonstrations.
Hands-on exercises and coding challenges after each module.
Physical and Virtual classes via Google Meet
Self-paced assignments and projects.

Frequency: 3 days a week - Days and time to be discussed

Module 1:

What is Python?

24th Dec 2024

Denis Oluka olukadeno@gmail.com

What is Python

- Python is an interpreted, object oriented, high level programming language with dynamic semantics
- Python is simple easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance
- Python supports modules and packages which emphasizes program modularity and code reuse.

History

Started by Guido Van Rossum as a hobby
Now widely spread
Open Source! Free!
Versatile

Who Uses Python

On-line games
Web services
Applications
Science
Instrument control
Embedded systems

Who Uses Python

Developed a large and active scientific computing and data analysis community

Now one of the most important languages for

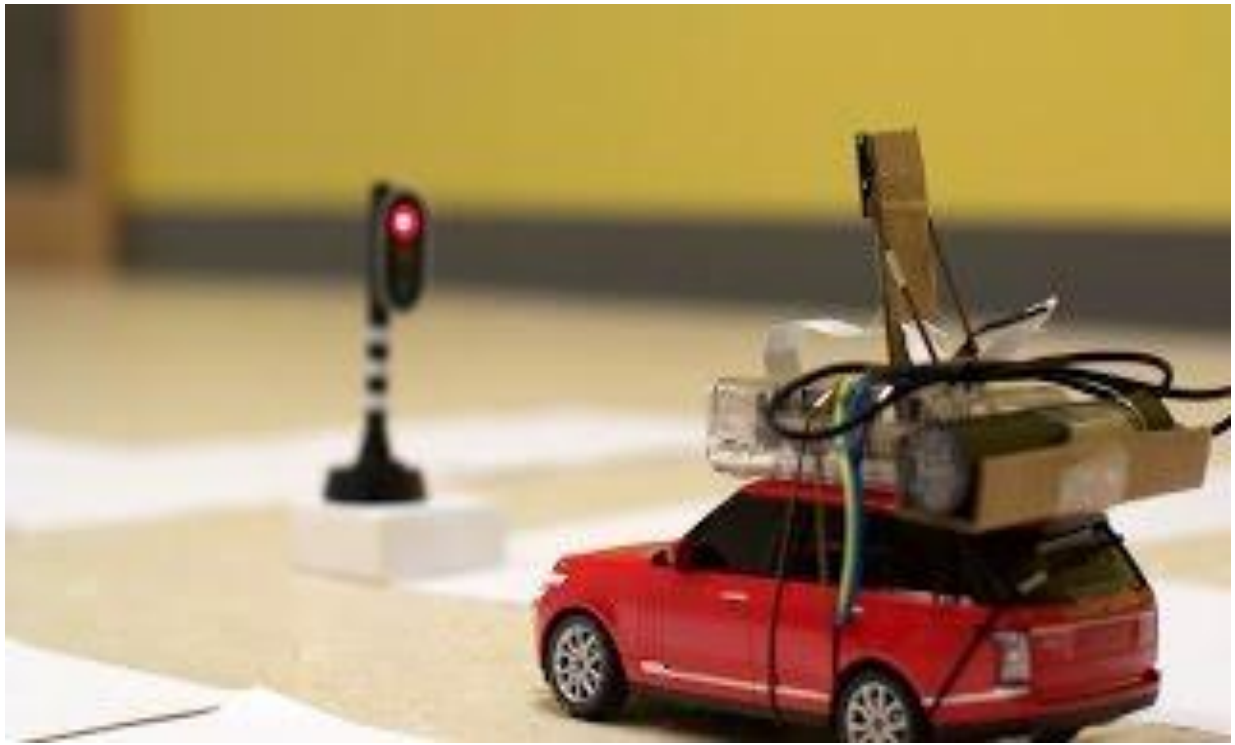
- Data science
- Machine learning
- General software development

Packages: NumPy, pandas, matplotlib, SciPy, scikit-learn, statsmodels

Failures

Coding is all about trial and error.
Don't be afraid of it.
Error messages aren't scary, they are useful.

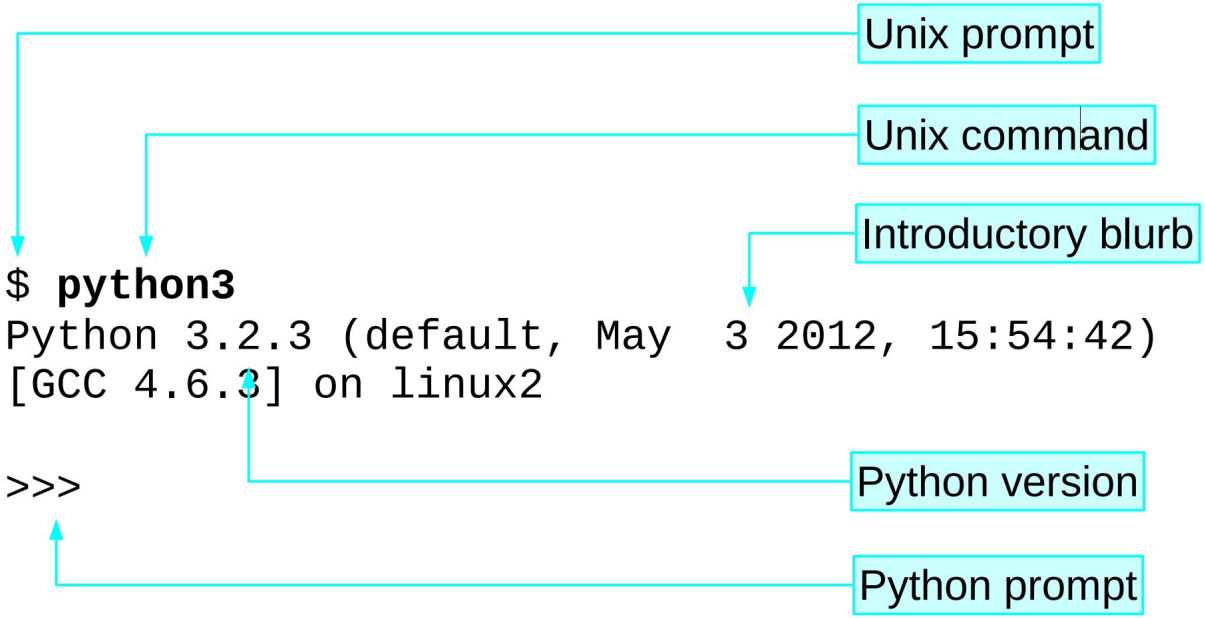
Demo



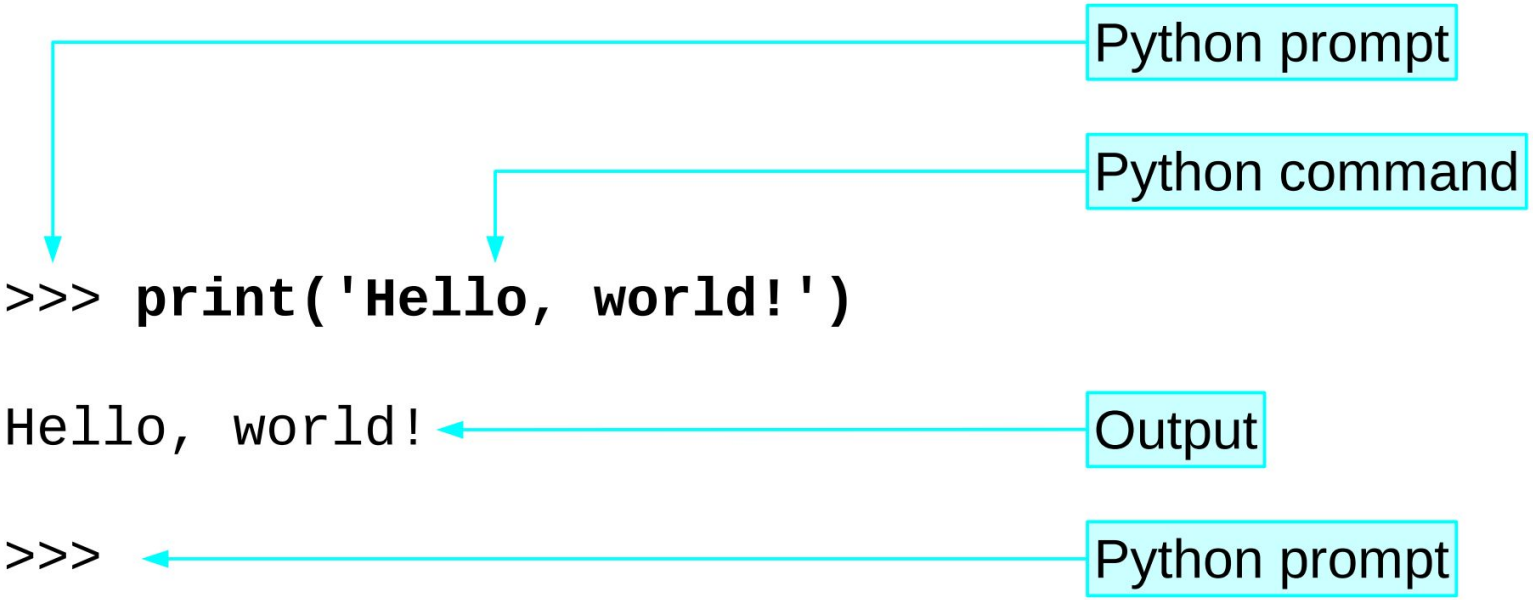
Tools

Python3
Command prompt
Vs Code
Git - VCS

Running Python



Syntax


Python prompt
Python command
>>> print('Hello, world!')
Hello, world!
Output
>>>
Python prompt

Installation

Visit
https://python.org/

Select and Download the stable version of python

Install Python

Run python command prompt

Thank you

Module 2:

Python Variables and Data Types

27th Dec 2024

Denis Oluka olukadeno@gmail.com

Variables in Python

What is a variable in programming?

A variable is essentially a reserved memory location that stores data values. In Python, variables do not require explicit declaration; they are created the moment a value is assigned to them. This flexibility allows for dynamic typing, meaning the type of a variable can change as needed.

They go on to serve as symbolic names for data stored in memory, allowing programmers to manipulate and reference values throughout their code. Here's a detailed explanation of variables, including their creation, usage, naming conventions, and scope.

Variables

Creating Variables

To create a variable in Python, you use the assignment operator (=) to assign a value to a name. The syntax is straightforward:

variable_name = value

```
x = 5          # x is an integer
y = "Hello"    # y is a string
z = 3.14       # z is a float
```

Variables

Dynamic Typing

Python automatically determines the data type of a variable based on the value assigned to it.

For instance, if you assign an integer to a variable and later assign a string to the same variable, Python will adjust its type accordingly:

```
x = 4          # x is of type int
x = "Sally"    # x is now of type str
```

Variables Rules

- Must start with a letter or underscore (_), not a number.
- Can’t contain spaces or special characters.
- Are case-sensitive (e.g., Name and name are different).

Valid naming
my_name = “Denis”
height = 30

Invalid naming
1user = "Invalid"
user-name = "Invalid"

Data Types

Data types in Python are attributes that tell a computer how to interpret a piece of data. They help identify the type of data, its size, and the functions associated with it.

Python has a rich set of built-in data types that are used to store, manipulate, and process data efficiently,these data types are stored in containers called variables

The data types are categorised as numeric data types,sequence data types,mapping ,sets,booleans, binaries and none types as explained below.

Numeric Data Types

int (Integer)

Description: Represents whole numbers without decimal points.
Example: 20, 10, -2, -100, 1000
Use: Counting, indexing, and mathematical operations that don't require fractional values.

float (Floating Point)

Description: Represents numbers with decimal points.
Example: 3.14, -0.5, 2.0, 0.021
Use: Precise calculations, scientific computations, and measurements.

Text Data Types

str (String)

Description: Represents a sequence of characters (text data).
Example: “hello”, ‘python’, “How are you!”
Use: Storing and manipulating textual information such as names, messages, or paragraphs

Sequence Data Types

list
Description: Represents an ordered, mutable collection of elements (can be of different types).
Example: [1, 19, 2, 90], [‘Denis’, 34, 12.9]
Use: Storing and organizing a collection of items that may need to change.

Sequence Data Types cont'd

tuple

Description: Represents an ordered, immutable collection of elements.
Example: (1, 19, 2, 90), ('Denis', 34, 12.9)
Use: Storing a fixed collection items, and the elements can be of mixed types.

Mapping data types
These do allow for the storage of data in key value pairs
dict (Dictionary)

Description: Represents a collection of key-value pairs.
Example: {'name': 'Alice', 'age': 25}
Use: Storing and retrieving data via keys, such as configurations or JSON-like structures

Set Data Types

set

Description: Represents an unordered collection of unique items.
Example: `{1, 2, 3}`, `{'apple', 'banana'}`
Use: Eliminating duplicates, performing set operations like union and intersection.

bool (Boolean Data type)

Description: Represents `True` or `False` values
Example: True, False
Use: Logic and control flow in decision-making statements

Binary Data Types

Binary data types are used for handling binary data.

bytes
These are immutable sequences majorly used for binary data such as file streams or network data.
Example: b“Hello”

None
The special type or none value represent the absence of a null value or ‘no value’ in python programming.

Casting Data Types

Casting is the operation of converting a variable from one data type to another.

Example

```
x = 10
y = “20”
x + int(y)
```

Castings include; int(), float(), str(), bool(), dict(), list(), tuple(), set()

Python Operators

Python operators are special symbols that perform operations on variables and values.

They can be categorized into several types, including arithmetic, assignment, comparison, logical, bitwise, membership, and identity operators.

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations.

Order of precedence is the same as in Mathematics.

Operator	Operation	Example
`+`	Addition	`5 + 2 = 7`
`-`	Subtraction	`4 - 2 = 2`
`*`	Multiplication	`2 * 3 = 6`
`/`	Division	`4 / 2 = 2.0`
`//`	Floor Division	`10 // 3 = 3`
`%`	Modulo	`5 % 2 = 1`
`**`	Exponentiation	`4 ** 2 = 16`

Assignment Operators

Assignment operators assign values to variables and can also perform operations during assignment.

Operator	Description	Example
`=`	Assignment	`a = 10`
`+=`	Addition Assignment	`a += 5 # a = a + 5`
`-=`	Subtraction Assignment	`a -= 3 # a = a - 3`
`*=`	Multiplication Assignment	`a *= 4 # a = a * 4`
`/=`	Division Assignment	`a /= 3 # a = a / 3`
`%=`	Modulus Assignment	`a %= 10 # a = a % 10`
`**=`	Exponent Assignment	`a **= 2 # a = a ** 2`
`//=`	Floor Division Assignment	`a //= 3 # a = a // 3`

Comparison Operators

Comparison operators compare two values and return Boolean results (True or False)

Operator	Description	Example
`==`	Equal to	`5 == 5` (True)
`!=`	Not equal to	`5 != 4` (True)
`>`	Greater than	`5 > 4` (True)
`<`	Less than	`5 < 6` (True)
`>=`	Greater than or equal to	`5 >= 5` (True)
`<=`	Less than or equal to	`5 <= 6` (True)

Logical Operators

Logical operators are used to combine conditional statements.

Operator	Description
`and`	Returns True if both statements are true
`or`	Returns True if at least one statement is true
`not`	Reverses the logical state of its operand

Logical Operators

Logical operators are used to combine conditional statements.

Operator	Description
`and`	Returns True if both statements are true
`or`	Returns True if at least one statement is true
`not`	Reverses the logical state of its operand

Membership Operators

Membership operators test for membership in sequences such as lists, tuples, or strings.

Operator	Description
`in`	Returns True if the value is found in the sequence
`not in`	Returns True if the value is not found in the sequence

Operator Precedence

When a calculation has multiple operators, each operator is evaluated in order of **precedence**

Operator	Meaning
()	Parentheses
**	Exponentiation (right associative)
*, /, //, %	Multiplication, division, floor division, modulo
+, -	Addition, subtraction
<, <=, >, >=, ==, !=	Comparison operators

Math Module

Importing a Module

A **module** is previously written code that can be imported in a program. The **import statement** defines a
variable for accessing code in a module. Import statements often appear at the beginning of a program.

Function	Description	Examples
Number-theoretic		
`math.ceil(x)`	The ceiling of x: the smallest integer greater than or equal to x.	`math.ceil(7.4)` → `8` `math.ceil(-7.4)` → `-7`
`math.floor(x)`	The floor of x: the largest integer less than or equal to x.	`math.floor(7.4)` → `7` `math.floor(-7.4)` → `-8`
Power and logarithmic		
`math.log(x)`	The natural logarithm of x (to base e).	`math.log(math.e)` → `1.0` `math.log(0)` → `ValueError: math domain error`

Math Module

Importing a Module

Example

```
import math

x1 = float(input("Enter x1: "))
y1 = float(input("Enter y1: "))
x2 = float(input("Enter x2: "))
y2 = float(input("Enter y2: "))
distance = math.sqrt((x2-x1)**2 + (y2-y1)**2)
print("The distance is", distance)
```

Module 3:

Control Statements

31st Dec 2024

Denis Oluka olukadeno@gmail.com

Control Statements

Control statements are essential in Python programming as they dictate the flow of execution based on certain conditions.

They allow for decision-making, looping, and managing the sequence of operations.

Conditional Statements

If Statement

A **condition** is an expression that evaluates to true or false. An **if statement** is a decision-making structure that contains a condition and a body of statements. If the condition is true, the body is executed. If the condition is false, the body is not executed.

```
if condition:
    # code to execute if condition is true
else:
    # code to execute if the above condition is false
```

Conditional Statements

Elif example

```
hour = 9
if hour < 8:
    print("Too early")
elif hour < 12:
    print("Good morning")
elif hour < 13:
    print("Lunchtime")
elif hour < 17:
    print("Good afternoon")
else:
    print("Too late")
```

Conditional Statements

Chained Decisions (elif)

Sometimes, a complicated decision is based on more than a single condition.

```
if condition:
    # code to execute if condition is true
elif another_condition:
    # code to execute if another_condition is true
else:
    # code to execute if none of the conditions are true
```

Conditional Statements

Loops

A **loop** is a code block that runs a set of statements while a given condition is true. A loop is often used for performing a repeating task.

Two types of loops, for loop and while loop.

Uses:
Alarms
Sending messages

Conditional Statements

While Loop

A **while loop** is a code construct that runs a set of statements, known as the loop body, when given condition, known as the loop expression, is true. At each iteration, once the loop statement is executed, the loop expression is evaluated again.

- If true, the loop body will execute at least one more time (also called looping or iterating one more time).
- If false, the loop's execution will terminate and the next statement after the loop body will execute.

Conditional Statements

While Loop

```
# Initialization
counter = 1

# While loop condition
while counter <= 10:
    if counter % 2 == 1:
        print(counter)
    # Counting up and increasing counter's value by 1 in each iteration
    counter += 1
```

Conditional Statements

For Loop

. A **for loop** iterates over all elements in a container. Ex: Iterating over a class roster and printing students' names.

```
str_var = "My Name"
count = 0
for c in str_var:
        print(c)
        count += 1
print(count)
```

Conditional Statements

Nested Loops

Example 1:

```
for i in range(1, 4):
        for j in range(4):
                print(i * j)
```

Conditional Statements

Nested Loops

Example2:

```
numbers = [12, 5, 3]
i = 0
for n in numbers:
        while i < n:
                print (i, end = " ")
                i += 2
        i = 0
        print()
```

Conditional Statements

Break and Continue

Break

A **break** statement is used within a for or a while loop to allow the program execution to exit the loop once a given condition is triggered. A break statement can be used to improve runtime efficiency when further loop execution is not required.

```
user_string = "This is a string."
for i in range(len(user_string)):
        if user_string[i] == 'a':
                print("Found a at index:", i)
                break
```

Conditional Statements

Break and Continue

Continue

A **continue** statement allows for skipping the execution of the remainder of the loop without exiting the loop entirely. A continue statement can be used in a for or a while loop. After the continue statement's execution, the loop expression will be evaluated again and the loop will continue from the loop's expression

```
i = 10
while i >= 0:
        i -= 1
        if i%3 != 0:
                continue
        print(i)
```

Conditional Statements

Loop else

A **loop else** statement runs after the loop's execution is completed without being interrupted by a break statement. A loop else is used to identify if the loop is terminated normally or the execution is interrupted by a break statement.

```
numbers = [2, 5, 7, 11, 12]
for i in numbers:
    if i == 10:
        print("Found 10!")
        break
else:
    print("10 is not in the list.")
```

Module 4: Functions and Modules
7th Jan 2025

Denis Oluka olukadeno@gmail.com

Functions and Modules

Python functions and modules are essential components that facilitate code organization, reusability, and maintainability.

This section will cover the basics of defining and using functions, as well as creating and importing modules in Python.

Functions and Modules

Functions

A function in Python is a block of reusable code that performs a specific task. Functions can take inputs (arguments) and may return outputs (values).

They help in breaking down complex problems into smaller, manageable parts

In Python there are two types of functions; Built-in and User-defined functions

Functions and Modules

Functions

Types of Functions

Built-in Functions: These are pre-defined functions provided by Python, such as print(), len(), and input().

User-defined Functions: These are functions created by the user to perform specific tasks. They are defined using the **def** keyword.

Functions and Modules

Functions

Syntax

```
def function_name(parameters):
    # function body
    return value  # optional
```

Example

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

result = add(5, 3)  # result is 8
```

Functions and Modules

Modules

A module is a file containing Python code (functions, classes, variables) that can be reused in other programs. It allows for better organization of code.

To create a module, save your functions in a `.py` file.

Functions and Modules

Modules

Using Modules

To use a module, you import it using the `import` statement:

```
import mymodule

mymodule.greeting("Jonathan")  # Output: Hello, Jonathan
```

Functions and Modules

Modules

Importing Specific Elements

You can import specific functions or variables from a module:

```
from mymodule import greeting

greeting("Alice")  # Output: Hello, Alice
```

Functions and Modules

Modules

Importing with Aliases

You can create an alias for a module to simplify its usage:

```
import mymodule as mx

mx.greeting("Bob")  # Output: Hello, Bob
```

Functions and Modules

Positional and Keyword Arguments

Positional Arguments

They must be passed to the function in the same order as the parameters are defined in the function signature.

- **Order Matters**: The first positional argument corresponds to the first parameter, the second to the second parameter, and so on.
- **Mandatory**: All positional arguments must be provided when calling a function, unless default values are defined for some parameters.
- **Type Matching:** The types of the arguments should ideally match the expected types of the parameters.

Functions and Modules

Positional and Keyword Arguments

Keyword Arguments

Keyword arguments allow you to pass arguments to a function by explicitly specifying the parameter name along with its value.

- **Order Independence:** When using keyword arguments, you can specify them in any order.
- **Optional Parameters:** Keyword arguments can be used for optional parameters that have default values.
- **Clarity**: Using keyword arguments makes it clear what each argument represents, which is particularly useful in functions with many parameters.

Functions and Modules

Arguments

Variable-Length Arguments in Python (*args)

This allows the function to accept any number of positional arguments.

- Tuple Storage: All the extra positional arguments passed to the function are stored in a tuple.
- Zero or More Arguments: Functions can accept zero or more arguments when defined with `*args`.
- Order of Parameters: A regular positional parameter can precede `*args`, but not follow it. Keyword arguments can be included after `*args`.

Functions and Modules

Built in Modules

The **Python Standard Library** is a collection of built-in functions and modules that support common programming tasks.

Module	Description
`calendar`	General calendar-related functions.
`datetime`	Basic date and time types and functions.
`email`	Generate and process email messages.
`math`	Mathematical functions and constants.
`os`	Interact with the operating system.
`random`	Generate pseudo-random numbers.
`statistics`	Mathematical statistics functions.
`sys`	System-specific parameters and functions.
`turtle`	Educational framework for simple graphics.
`zipfile`	Read and write ZIP-format archive files.

Functions and Modules

Third-party modules

The **Python Package Index** (PyPI), available at pypi.org, is the official third party software library for Python.

PyPI allows anyone to develop and share modules with the Python community. Module authors include individuals, large companies, and non-profit organizations. PyPI helps programmers install modules and receive updates

Module	Description
`arrow`	Convert and format dates, times, and timestamps.
`BeautifulSoup`	Extract data from HTML and XML documents.
`bokeh`	Interactive plots and applications in the browser.
`matplotlib`	Static, animated, and interactive visualizations.
`moviepy`	Video editing, compositing, and processing.
`nltk`	Natural language toolkit for human languages.
`numpy`	Fundamental package for numerical computing.
`pandas`	Data analysis, time series, and statistics library.
`pillow`	Image processing for jpg, png, and other formats.

Module 5:

Exception & File Handling

17th Jan 2025

Denis Oluka olukadeno@gmail.com

Exception Handling

In Python, exceptions are a fundamental part of error handling. They allow developers to manage errors gracefully without crashing the program.

The Exception Hierarchy

The exception hierarchy in Python is structured as follows:

- BaseException: This is the root class for all exceptions in Python. It should not be used directly in code as it captures all exceptions, including system-exiting exceptions like `SystemExit`

Exception Hierarchy

- **Exception**: This is a subclass of BaseException and is the most commonly used class for catching exceptions. It encompasses all standard exceptions that are not intended to terminate the program3.
- **Standard Exceptions**: These include various built-in exceptions derived from Exception, such as:
  - **ArithmeticError**: Base class for arithmetic-related errors.ZeroDivisionError
  - **ValueError**: Raised when a function receives an argument of the right type but inappropriate value.
  - **TypeError**: Raised when an operation or function is applied to an object of inappropriate type

Catching Exceptions

```
try:
    # some code that may raise an exception
except ValueError:
    # handle ValueError
except Exception:
    # handle any other exception
```

Order of Handling: More specific exceptions should be caught before more general ones.

Raising Exceptions

You can raise exceptions using the raise statement. This can be used to trigger an exception intentionally when a certain condition is met:

```
def check_positive(number):
    if number < 0:
        raise ValueError("Number must be positive")
```

File Handling

This allows for the creation, reading, writing, and manipulation of files. Python provides built-in functions that simplify these operations without the need for additional libraries.

Key Functions

The primary functions involved in file handling include:

- open(): Opens a file and returns a file object.
- read(): Reads data from a file.
- write(): Writes data to a file.
- close(): Closes the file and releases resources.

File Modes

Mode	Description
`r`	Read mode; opens a file for reading (file must exist).
`w`	Write mode; creates a new file or truncates an existing file.
`a`	Append mode; opens a file for appending (creates if it doesn't exist).
`x`	Exclusive creation; fails if the file already exists.
`b`	Binary mode; used for binary files (e.g., images).
`t`	Text mode; default mode for text files.

Basic Operations

Opening File

To open a file, use the `open()` function

Syntax

```
file_object = open('filename.txt', 'mode')
```

Reading from a file

To open a file, use the `open()` function

- read(size): Reads up to `size` bytes from the file.
- readline(): Reads a single line from the file.
- readlines(): Reads all lines and returns them as a list.

Basic Operations

Read a file example:

With context manager

```
with open('example.txt', 'r') as f:
    content = f.read()
    print(content)
```

Without context manager

```
file = open("example.txt", "r")
try:
   content = file.read()
   print(content)
finally:
   file.close()
```

Basic Operations

Writing to a File

The **write()** function is used to write to an already opened file. The write() function will only accept a string parameter. Other variable types must be cast to string before writing using write().

Example

```
with open('output.txt', 'w') as f:
    f.write("Hello, World!")
```

The write() function does not automatically add a newline character, a newline must be added explicitly by adding a newline ('\n') character.

NOTE: Always use close() after every file operation

Handling File Exceptions

Runtime Errors

Various errors may occur when reading a file:

- FileNotFoundError: The filename or path is invalid.
- IndexError/ValueError: The file's format is invalid.
- Other errors caused by invalid contents of a file

File Handling

Example

```
name = input("Enter a filename: ")
try:
  file = open(name)
  lines = file.readlines()
  count = len(lines)
  print(name, "has", count, "lines")
except FileNotFoundError:
  print("File not found:", name)
print("Have a nice day!")
```

Thank you