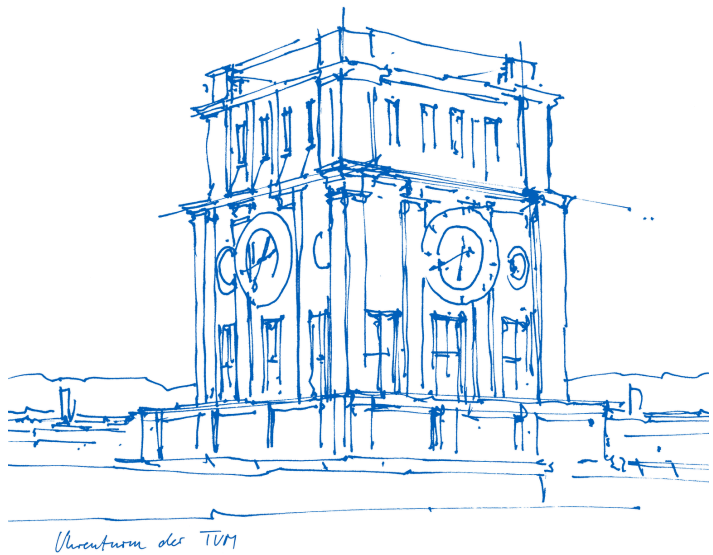
Crash course: Data Science with Python and R

Python (Part 1)

Adapted from Dr. Raoul Rothfeld's slides

Cheng Lyu & Dr. Mohamed Abouelela
Technical University of Munich
TUM School of Engineering and Design
Chair of Transportation Systems Engineering
Munich, 11.03.2024



Course Overview

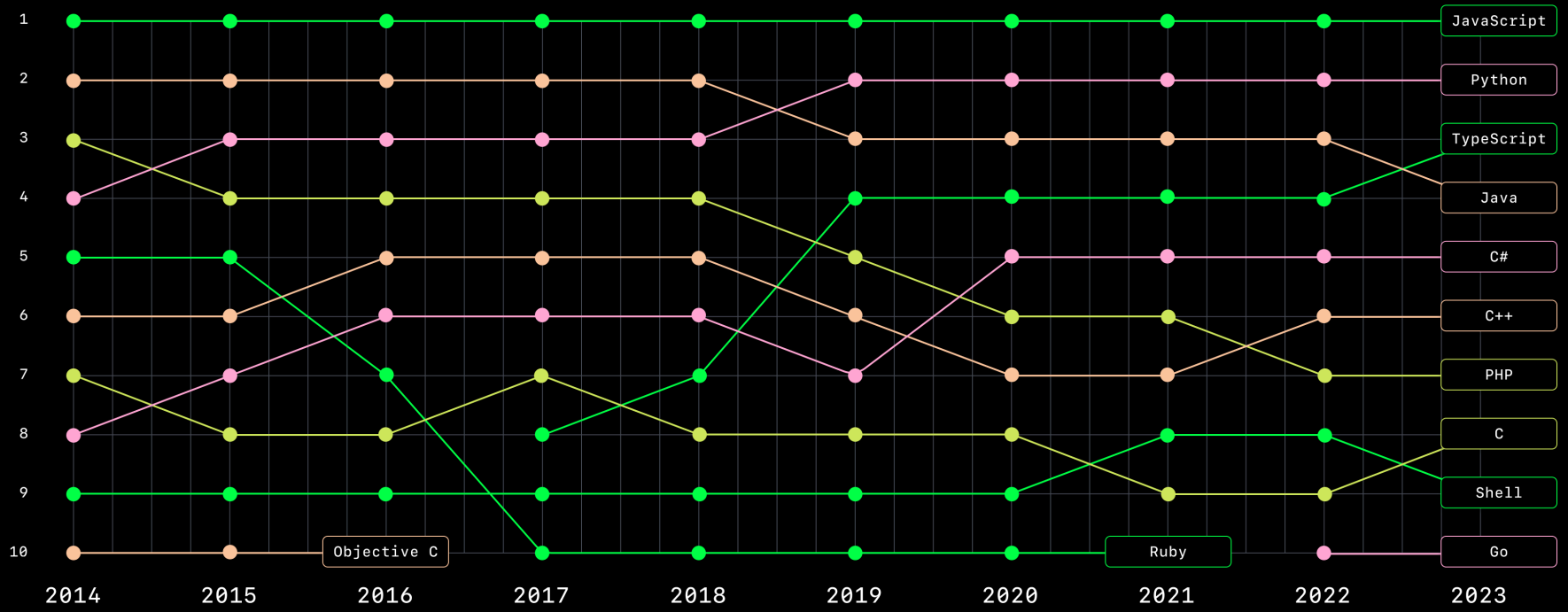
Lecture time and format:

- Monday to Thursday from 10:00 to 12:00 (CET) via Zoom (video web conference)
- After each lecture, a recording of the lecture will be available on-demand via Moodle
- For communication, we use Zoom, Moodle, and GitLab (all are TUM services)
- Interactive lecture with integrated exercises

Objectives:

- Assuming that many students have never programmed, provide basic programming understanding and functional coding ability (in contrast to object-oriented programming and software engineering)
- Enable simple data analytics and visualizations, and provide a basis for further self-learning

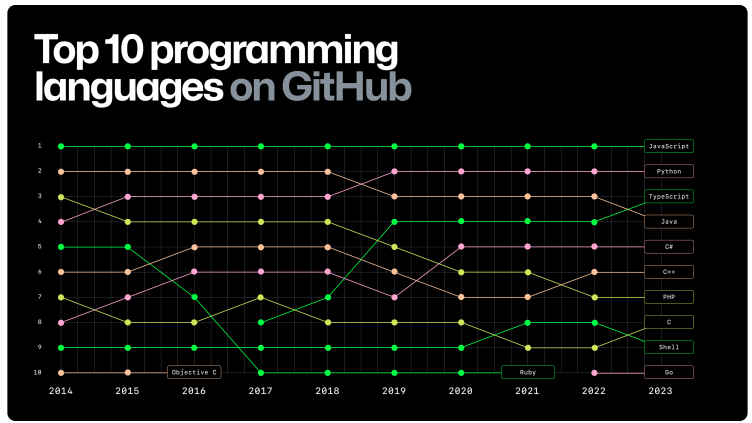
Top 10 programming languages on GitHub



Popularity of Programming Languages

Some language categories:

- General-purpose (e.g. **Python**, Java, C#)
- Statistical/mathematical (e.g. **R**, Matlab, SAS)
- Web-functionality (e.g. JavaScript, PHP, Ruby)
- Lower-level (e.g. C++, C, Assembly)
- ...



Source: GitHub Octoverse

Where is Python mostly used?

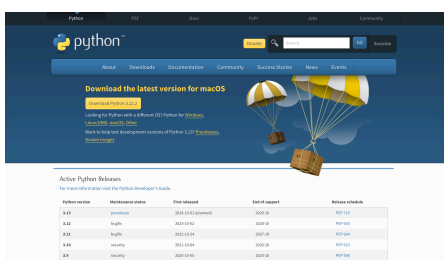
- **Data Analysis**: Data wrangling, data visualization
- **Scientific Computing**: Numerical simulations, optimization
- **Artificial Intelligence & Machine Learning**: Predictive models, neural networks
- **Automation & Scripting**: Task automation, system administration
- Web Development: Web frameworks, web apps
- Desktop Applications: Cross-platform GUIs
- Game Development: Game engines, indie games

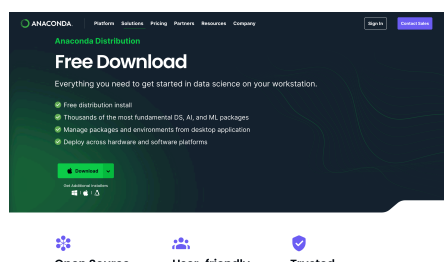
Necessary Tools for the Course

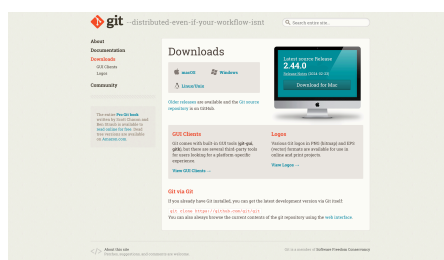
An installation guide can be found on the Moodle page of the course.

Tools that you will need to have installed:

- Python (programming language): https://www.python.org/downloads/
- Anaconda (environment manager): https://www.anaconda.com/products/distribution
- Visual Studio Code (code editor): https://code.visualstudio.com
- Git (code versioning, optional): https://git-scm.com/downloads







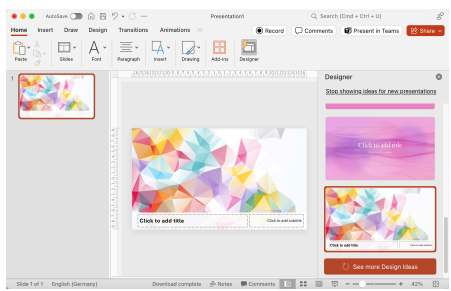
Terminal, GUI, IDE and Code Editor

Terminal, GUI, IDE and Code Editor

- Terminal/console: text-based interfaces for human-computer interaction (HCI).

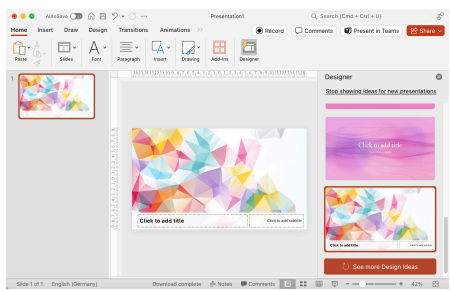
Terminal, GUI, IDE and Code Editor

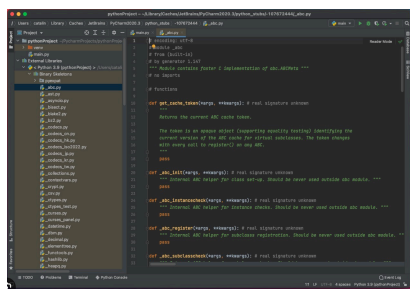
- Terminal/console: text-based interfaces for human-computer interaction (HCI).
- Graphical user interface (GUI): visual interfaces for HCI that allow point-and-click interaction.



Terminal, GUI, IDE and Code Editor

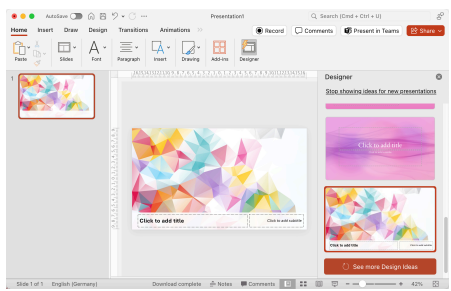
- Terminal/console: text-based interfaces for human-computer interaction (HCI).
- Graphical user interface (GUI): visual interfaces for HCI that allow point-and-click interaction.
- Integrated development environment (IDE): GUIs that combine all tools to write, test, and execute programmes.

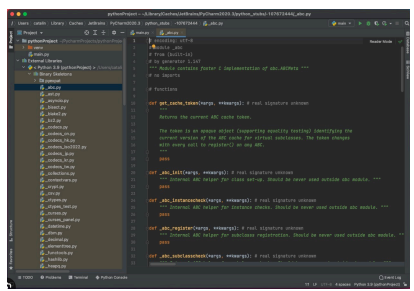
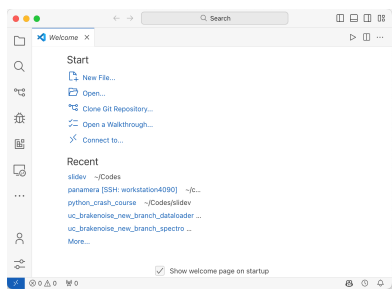




Terminal, GUI, IDE and Code Editor

- Terminal/console: text-based interfaces for human-computer interaction (HCI).
- Graphical user interface (GUI): visual interfaces for HCI that allow point-and-click interaction.
- Integrated development environment (IDE): GUIs that combine all tools to write, test, and execute programmes.
- Code editor: a simpler version of an IDE, with fewer features, but faster and more lightweight.

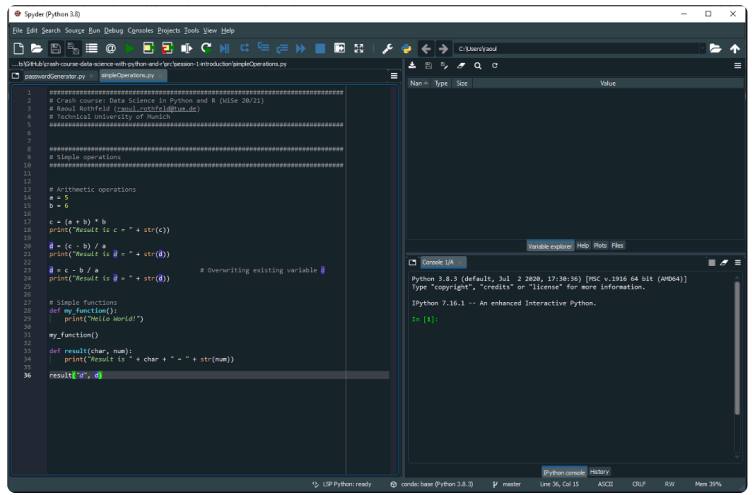


Terminal, GUI, IDE and Code Editor

Integrated development environment (IDE): GUIs that combine all tools to write, test, and execute programmes. Most IDEs are language- and purpose-specific, e.g.:

- *Spyder*: (open-source) data science for Python
- *PyCharm*: (commercial) general-purpose/software development for Python
- *DataSpell*: (commercial) data science with Jupyter notebooks of Python

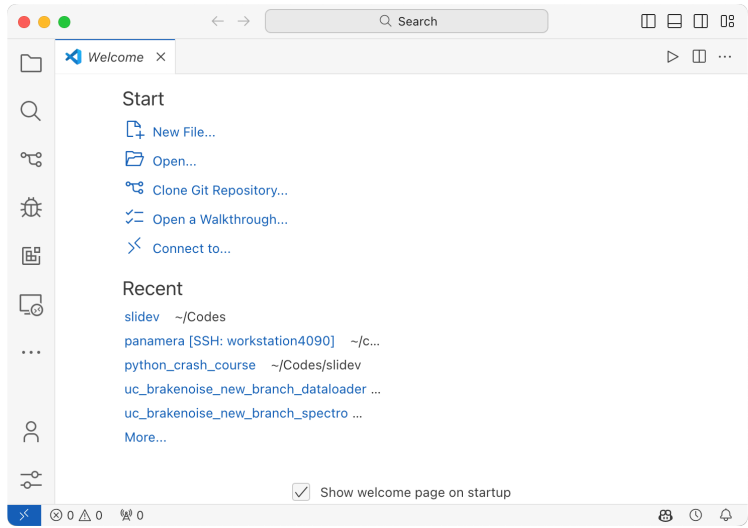


Spyder

Terminal, GUI, IDE and Code Editor

Code Editor: GUIs similar to IDEs that combine all tools to write, test, and execute programmes. Compared with IDEs, editors are not language- and purpose-specific, allowing users to work on multiple tasks simultaneously.

- *Visual Studio Code*: (open-source)
- *Atom*: (open-source)
- *Sublime Text*: (commercial)



Visual Studio Code

First Line in Python

- Open the **terminal** (e.g. Command Prompt, PowerShell, Terminal).
- Launch the **Python interpreter** by typing `` `python` `` and pressing `Enter`.

```
python
```

- Now we are in the interactive Python interpreter.
- **Greet the world** with a simple line of code!
- Press `Enter` to execute the code.

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

- Strings in Python are enclosed in either single or double quotes.
- Want to **get out of the Python interpreter**?

```
exit()
```

First Line in Python

- Open the **terminal** (e.g. Command Prompt, PowerShell, Terminal).
- Launch the **Python interpreter** by typing `` `python` `` and pressing `Enter`.

```
python
```

- Now we are in the interactive Python interpreter
- **Greet the world** with a simple line of code!
- Press `Enter` to execute the code.

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

- Strings in Python are enclosed in either single or double quotes.
- Want to **get out of the Python interpreter**?

```
exit()
```

First Line in Python

The interactive Python interpreter can also accept input from the user.

- Launch the **Python interpreter**.
- Use the `input()` function to prompt the user for input.

```
name = input("What is your name? ")
print("Hello,", name)
```

- The `input()` function returns a *string*.
- The `print()` function can take multiple arguments, separated by commas.

Basic Arithmetic Operations

- Launch the **Python interpreter**.
- Simple arithmetic operations, including addition, subtraction, multiplication, and division, can be directly performed in the Python interpreter.
- The answer will be displayed immediately after pressing `Enter` in the interactive interpreter.

```
2 + 3
```

- Compound operations can also be performed in a natural way.

```
5 - 2 * 3 + 8 / (2 - 1)
```

- **Exponentiation**, **modulo**, and **integer division** are also supported.

```
2**3, 10 % 3, 10 // 3
```

Variables

Variables are used to store data.

- In Python, variables are created when a value is **assigned** to them.

```
name = "Alice"
length = 123456789
radius = 12.5
```

- The variable name can contain letters, numbers, and underscores, but cannot start with a number.
- Variable names are **case-sensitive**, and should NOT be the same as **Python keywords**.
  - `name`, `Name123`, `__name__`, `__Name__`, `__123Name`,
  - `123Name`, `name@123`, `Name-123`, `and`, `def`
- The value of a variable can be changed at any time.

```
name = "Bob"
length = length + 100
radius *= 2
```

Variables

Variables are used to store data.

- In Python, variables are created when a value is **assigned** to them.

```
name = "Alice"
length = 123456789
radius = 12.5
```

- The variable name can contain letters, numbers, and underscores, but cannot start with a number.
- Variable names are **case-sensitive**, and should NOT be the same as **Python keywords**.
  - `name`, `Name123`, `__name__`, `__Name__`, `__123Name`,
  - ~~`123Name`, `name@123`, `Name 123`, `and`, `def`~~
- The value of a variable can be changed at any time.

```
name = "Bob"
length = length + 100
radius *= 2
```

Variables

- There is no constant in Python, but by convention, variable names in **UPPERCASE** are considered as constants.

```
PI = 3.14159265359
```

- What happens when creating a variable in Python?
  - A **memory space** is allocated to store the value.
  - The **value** is stored in the memory space.
  - The **variable name** is associated with the memory space.

```
old_name = "Alice"
new_name = old_name
old_name = "Bob"
print(new_name)
```

Data Types

Although not explicitly declared, variables in Python do have **data types**.

Basic built-in data types:

Data Types

Although not explicitly declared, variables in Python do have **data types**.

Basic built-in data types:

- **Integers**: whole numbers, e.g. `1, -2, 0, 1230, 456789`
  - Special: `0b1010` (binary), `0o123` (octal), `0x1A` (hexadecimal)
  - Separators: `1_000_000`

Data Types

Although not explicitly declared, variables in Python do have **data types**.

Basic built-in data types:

- **Integers**: whole numbers, e.g. `1, -2, 0, 1230, 456789`
  - Special: `0b1010` (binary), `0o123` (octal), `0x1A` (hexadecimal)
  - Separators: `1_000_000`
- **Float**: numbers with a decimal point, e.g. `1.0, 2.01, 999.999`
  - Scientific notation: `1e3` (1×10^3), `1.23e-4` (1.23×10^{-4})

Data Types

Although not explicitly declared, variables in Python do have **data types**.

Basic built-in data types:

- **Integers**: whole numbers, e.g. `1, -2, 0, 1230, 456789`
  - Special: `0b1010` (binary), `0o123` (octal), `0x1A` (hexadecimal)
  - Separators: `1_000_000`
- **Float**: numbers with a decimal point, e.g. `1.0, 2.01, 999.999`
  - Scientific notation: `1e3` (1×10^3), `1.23e-4` (1.23×10^{-4})
- **Complex**: numbers with a real and an imaginary part, e.g. `1 + 2j, 3.2 - 0.4j`
  - Both parts are floats, and the imaginary part is denoted by `j` or `J`

Data Types

Although not explicitly declared, variables in Python do have **data types**.

Basic built-in data types:

Data Types

Although not explicitly declared, variables in Python do have **data types**.

Basic built-in data types:

- Integers, Float, Boolean

Data Types

Although not explicitly declared, variables in Python do have **data types**.

Basic built-in data types:

- Integers, Float, Boolean
- **Boolean**: `True` or `False`
  - Comparison: `==` , `!=` , `<` , `>` , `<=` , `>=`
  - Logical: `and` , `or` , `not`
  - Bitwise: `&` , `|` , `^` , `~` , `<<` , `>>`
  - Identity & Membership: `is` , `is not` , `in` , `not in`

Data Types

Although not explicitly declared, variables in Python do have **data types**.

Basic built-in data types:

- Integers, Float, Boolean
- **Boolean**: `True` or `False`
  - Comparison: `==` , `!=` , `<` , `>` , `<=` , `>=`
  - Logical: `and` , `or` , `not`
  - Bitwise: `&` , `|` , `^` , `~` , `<<` , `>>`
  - Identity & Membership: `is` , `is not` , `in` , `not in`
- **None**: a special type representing the absence of a value

Data Types

Although not explicitly declared, variables in Python do have **data types**.

Basic built-in data types:

Data Types

Although not explicitly declared, variables in Python do have **data types**.

Basic built-in data types:

- Integers, Float, Boolean, None

Data Types

Although not explicitly declared, variables in Python do have **data types**.

Basic built-in data types:

- Integers, Float, Boolean, None
- **String**: text, e.g. `"Hello", "world", '123'`
  - Escape characters: `"\t", "\\", "\'", "\"", "\n", "\r"`, (seldom used: `"\b", "\f", "\110", "\x48"`)
  - Raw strings: `r"Hello\nworld"`
  - Multiline strings: ...
  - f-strings: `f"Hello {name}!"`

Data Types

- Conversion between data types is possible.
  - `int()`: convert a numeric string or float to an integer
  - `float()`: convert a numeric string or integer to a float
  - `str()`: convert any data type to a string
  - `bool()`: convert any data type to a boolean
- The data type of a variable can be checked using the `type()` function.

```
a = 123
b = 123.0
c = "123"
d = 2 + 3j
print(type(a), type(b), type(c), type(d.imag))
```

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Indexing**: access individual characters using square brackets.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Indexing**: access individual characters using square brackets.

```
s = "Hello hello hello "
print(s[0], s[1], s[-1], s[-2])
```

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Indexing**: access individual characters using square brackets.

```
s = "Hello hello hello "
print(s[0], s[1], s[-1], s[-2])
```

- **Slicing**: access a range of characters using square brackets.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Indexing**: access individual characters using square brackets.

```
s = "Hello hello hello "
print(s[0], s[1], s[-1], s[-2])
```

- **Slicing**: access a range of characters using square brackets.

```
print(s[1:3], s[:3], s[2:], s[:], s[::2], s[-5::-1])
```

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Indexing**: access individual characters using square brackets.

```
s = "Hello hello hello "
print(s[0], s[1], s[-1], s[-2])
```

- **Slicing**: access a range of characters using square brackets.

```
print(s[1:3], s[:3], s[2:], s[:], s[::2], s[-5::-1])
```

- **Concatenation**: combine strings using the `+` operator.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Indexing**: access individual characters using square brackets.

```
s = "Hello hello hello "
print(s[0], s[1], s[-1], s[-2])
```

- **Slicing**: access a range of characters using square brackets.

```
print(s[1:3], s[:3], s[2:], s[:], s[::2], s[-5::-1])
```

- **Concatenation**: combine strings using the `+` operator.

```
print(s + " world!")
```

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Indexing**: access individual characters using square brackets.

```
s = "Hello hello hello "
print(s[0], s[1], s[-1], s[-2])
```

- **Slicing**: access a range of characters using square brackets.

```
print(s[1:3], s[:3], s[2:], s[:], s[::2], s[-5::-1])
```

- **Concatenation**: combine strings using the `+` operator.

```
print(s + " world!")
```

- **Repetition**: repeat a string using the `*` operator.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Indexing**: access individual characters using square brackets.

```
s = "Hello hello hello "
print(s[0], s[1], s[-1], s[-2])
```

- **Slicing**: access a range of characters using square brackets.

```
print(s[1:3], s[:3], s[2:], s[:], s[::2], s[-5::-1])
```

- **Concatenation**: combine strings using the `+` operator.

```
print(s + " world!")
```

- **Repetition**: repeat a string using the `*` operator.

```
print(s * 3)
```

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Length**: get the length of a string using the `` `len()` `` function.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Length**: get the length of a string using the `len()` function.

```
print(len(s))
```

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Length**: get the length of a string using the `len()` function.

```
print(len(s))
```

- **Membership**: check if a character or a substring is in a string using the `in` and `not in` operators.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Length**: get the length of a string using the `len()` function.

```
print(len(s))
```

- **Membership**: check if a character or a substring is in a string using the `in` and `not in` operators.

```
print("H" in s, "H" not in s, "h" in s)
```

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Length**: get the length of a string using the `len()` function.

```
print(len(s))
```

- **Membership**: check if a character or a substring is in a string using the `in` and `not in` operators.

```
print("H" in s, "H" not in s, "h" in s)
```

- **Case**: change the case of a string using the `upper()`, `lower()`, `title()`, `capitalize()`, `swapcase()` methods.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Length**: get the length of a string using the `len()` function.

```
print(len(s))
```

- **Membership**: check if a character or a substring is in a string using the `in` and `not in` operators.

```
print("H" in s, "H" not in s, "h" in s)
```

- **Case**: change the case of a string using the `upper()`, `lower()`, `title()`, `capitalize()`, `swapcase()` methods.

```
print(s.upper(), s.lower(), s.title(), s.capitalize(), s.swapcase())
```

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Length**: get the length of a string using the `len()` function.

```
print(len(s))
```

- **Membership**: check if a character or a substring is in a string using the `in` and `not in` operators.

```
print("H" in s, "H" not in s, "h" in s)
```

- **Case**: change the case of a string using the `upper()`, `lower()`, `title()`, `capitalize()`, `swapcase()` methods.

```
print(s.upper(), s.lower(), s.title(), s.capitalize(), s.swapcase())
```

- **Strip**: remove leading and trailing whitespace using the `strip()`, `lstrip()`, `rstrip()` methods.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Length**: get the length of a string using the `len()` function.

```
print(len(s))
```

- **Membership**: check if a character or a substring is in a string using the `in` and `not in` operators.

```
print("H" in s, "H" not in s, "h" in s)
```

- **Case**: change the case of a string using the `upper()`, `lower()`, `title()`, `capitalize()`, `swapcase()` methods.

```
print(s.upper(), s.lower(), s.title(), s.capitalize(), s.swapcase())
```

- **Strip**: remove leading and trailing whitespace using the `strip()`, `lstrip()`, `rstrip()` methods.

```
s = "  Hello  "
print(s.strip(), s.lstrip(), s.rstrip())
```

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Replace**: replace a substring with another using the `` `replace()` `` method.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Replace**: replace a substring with another using the `` `replace()` `` method.

```
print(s.replace("Hello", "Python"))
```

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Replace**: replace a substring with another using the `replace()` method.

```
print(s.replace("Hello", "Python"))
```

- **Split**: split a string into a list of substrings using the `split()` or `partition()` method.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Replace**: replace a substring with another using the `replace()` method.

```
print(s.replace("Hello", "Python"))
```

- **Split**: split a string into a list of substrings using the `split()` or `partition()` method.

```
s = "Hello, world!"
print(s.split(","), s.partition(","))
```

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Replace**: replace a substring with another using the `replace()` method.

```
print(s.replace("Hello", "Python"))
```

- **Split**: split a string into a list of substrings using the `split()` or `partition()` method.

```
s = "Hello, world!"
print(s.split(","), s.partition(","))
```

- **Join**: join a list of strings into a single string using the `join()` method.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Replace**: replace a substring with another using the `replace()` method.

```
print(s.replace("Hello", "Python"))
```

- **Split**: split a string into a list of substrings using the `split()` or `partition()` method.

```
s = "Hello, world!"
print(s.split(","), s.partition(","))
```

- **Join**: join a list of strings into a single string using the `join()` method.

```
s = ["Hello", "world!"]
print(" ".join(s))
```

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Replace**: replace a substring with another using the `replace()` method.

```
print(s.replace("Hello", "Python"))
```

- **Split**: split a string into a list of substrings using the `split()` or `partition()` method.

```
s = "Hello, world!"
print(s.split(","), s.partition(","))
```

- **Join**: join a list of strings into a single string using the `join()` method.

```
s = ["Hello", "world!"]
print(" ".join(s))
```

- **Fill**: fill a string to a certain length using the `ljust()`, `rjust()`, and `center()` methods.

Data Types

Strings are sequences of characters, and can be manipulated in various ways.

- **Replace**: replace a substring with another using the `replace()` method.

```
print(s.replace("Hello", "Python"))
```

- **Split**: split a string into a list of substrings using the `split()` or `partition()` method.

```
s = "Hello, world!"
print(s.split(","), s.partition(","))
```

- **Join**: join a list of strings into a single string using the `join()` method.

```
s = ["Hello", "world!"]
print(" ".join(s))
```

- **Fill**: fill a string to a certain length using the `ljust()`, `rjust()`, and `center()` methods.

```
s = "Hello"
print(s.center(20, "*"), s.ljust(20, "*"), s.rjust(20, "*")
```

Data Types

Other string manipulation methods:

- **Search**: find the index of a substring using the `find()` and `index()` methods.

```
s = "Hello, world!"
print(s.find("l"), s.index("l"))
```

- **Count**: count the occurrences of a substring using the `count()` method.

```
print(s.count("l"))
```

- **Starts/Ends**: check if a string starts or ends with a substring using the `startswith()` and `endswith()` methods.

```
print(s.startswith("Hello"), s.endswith("Python"))
```

- **Check**: check if a string is alphanumeric, lowercase, etc., using various `is...` methods.

```
print(s.isalpha(), s.islower(), s.isspace(), s.istitle(), s.isupper())
```

Data Types

Strings have a rich set of methods for manipulation because they contain a collection of sub-elements. A similar class of data types is **container**.

Container data types:

Data Types

Strings have a rich set of methods for manipulation because they contain a collection of sub-elements. A similar class of data types is **container**.

Container data types:

- **List**: a collection of elements, e.g. `[1, 2, 3, "a", "b", "c"]`
  - Indexing, slicing, concatenation, repetition, length, membership, etc.
  - Methods: `append()`, `extend()`, `insert()`, `remove()`, `pop()`, `clear()`, `index()`, `count()`, `sort()`, `reverse()`, etc.

Data Types

Strings have a rich set of methods for manipulation because they contain a collection of sub-elements. A similar class of data types is **container**.

Container data types:

- **List**: a collection of elements, e.g. `[1, 2, 3, "a", "b", "c"]`
  - Indexing, slicing, concatenation, repetition, length, membership, etc.
  - Methods: `append()`, `extend()`, `insert()`, `remove()`, `pop()`, `clear()`, `index()`, `count()`, `sort()`, `reverse()`, etc.
- **Tuple**: an immutable collection of elements, e.g. `(1, 2, 3, "a", "b", "c")`
  - Indexing, slicing, concatenation, repetition, length, membership, etc.
  - Methods: `count()`, `index()`

Data Types

Strings have a rich set of methods for manipulation because they contain a collection of sub-elements. A similar class of data types is **container**.

Container data types:

Data Types

Strings have a rich set of methods for manipulation because they contain a collection of sub-elements. A similar class of data types is **container**.

Container data types:

- List, Tuple

Data Types

Strings have a rich set of methods for manipulation because they contain a collection of sub-elements. A similar class of data types is **container**.

Container data types:

- List, Tuple
- **Set**: an unordered collection of unique elements, e.g. `{1, 2, 3, "a", "b", "c"}`
  - Membership, length, etc.
  - Methods: `add()`, `remove()`, `pop()`, `clear()`, `union()`, `intersection()`, `difference()`, `symmetric_difference()`, etc.

Data Types

Strings have a rich set of methods for manipulation because they contain a collection of sub-elements. A similar class of data types is **container**.

Container data types:

- List, Tuple
- **Set**: an unordered collection of unique elements, e.g. `{1, 2, 3, "a", "b", "c"}`
  - Membership, length, etc.
  - Methods: `add()`, `remove()`, `pop()`, `clear()`, `union()`, `intersection()`, `difference()`, `symmetric_difference()`, etc.
- **Dictionary**: a collection of key-value pairs, e.g. `{"name": "Alice", "age": 25}`
  - Indexing, length, membership, etc.
  - Methods: `keys()`, `values()`, `items()`, `get()`, `pop()`, `popitem()`, `clear()`, `update()`, etc.

Coding with an Editor / IDE

Interactive Python interpreter is good for quick tests, but not for writing and saving code.

- Open a **code editor / IDE** (e.g. Visual Studio Code).
- **Greet the world** again with a simple line of code.

```
name = "Alice"
greeting = f"Hello, {name}!"
print(greeting)
```

- Try to **execute the code** in the terminal or directly in the code editor / IDE.

```
python hello.py
```

It's Conditional...

There are occasions when we want to execute different code based on different conditions.

- **if** statement: execute a block of code if a condition is true.
- **else** statement: execute a block of code if the same condition is false.
- **elif** statement: execute a block of code if the previous condition is false and the new condition is true.

```
grade = 85
if grade >= 50:
    print("Pass")
else:
    print("Fail")
```

- The code above can be suppressed using the **ternary operator**.

```
result = "Pass" if grade >= 50 else "Fail"
print(result)
```

It's Conditional...

There are occasions when we want to execute different code based on different conditions.

- **if** statement: execute a block of code if a condition is true.
- **else** statement: execute a block of code if the same condition is false.
- **elif** statement: execute a block of code if the previous condition is false and the new condition is true.

```
grade = 85
if grade >= 50:
    print("Pass")
else:
    print("Fail")
```

- The code above can be suppressed using the **ternary operator**.

```
result = "Pass" if grade >= 50 else "Fail"
print(result)
```

It's Conditional...

- Chaining `if` and `else` statements follows a sequential order.
- Where are the mistakes in the following code?

```
# An earthquake with a force of 6.3 on the richter scale occurred
richter = 6.3

if richter >= 3.5:
    print("Felt but no destruction")
if richter >= 4.5:
    print("Damage to poorly built buildings")
elif richter >= 6.0:
    print("Some buildings collapse")
elif richter >= 7.0:
    print("Many buildings destroyed")
elif richter >= 8.0:
    print("Most structures fall")
else:
    print("Not noticed")
```

- `#` is used to denote a **comment**, which is not executed.

It's Conditional...

- Chaining `if` and `else` statements follows a sequential order.
- Where are the mistakes in the following code?

```
# An earthquake with a force of 6.3 on the richter scale occurred
richter = 6.3

if richter >= 3.5:
    print("Felt but no destruction")
if richter >= 4.5:
    print("Damage to poorly built buildings")
elif richter >= 6.0:
    print("Some buildings collapse")
elif richter >= 7.0:
    print("Many buildings destroyed")
elif richter >= 8.0:
    print("Most structures fall")
else:
    print("Not noticed")
```

- `#` is used to denote a **comment**, which is not executed.

It's Conditional...

- **Nested if** statements are also possible.

```
temperature = 25
raining = False
if temperature > 20:
    if not raining:
        print("Go for a walk")
    else:
        print("Stay at home")
else:
    print("Stay at home")
```

- **Indentation** is crucial – sometimes a nightmare – in Python to define the scope of the code block.
- Indentation is usually **4 spaces**, but can be 2 or 8 spaces, or a tab.

It's Conditional...

- Since Python 3.10, the **match** statement is introduced to simplify the conditional logic for simple cases.
- Not recommended in most cases due to potential inconsistency and incompatibility.

```
error_code = 404
match error_code:
    case 200:
        print("OK")
    case 404:
        print("Not Found")
    case 500:
        print("Internal Server Error")
    case _:
        print("Unknown Error")
```

Loops

Loops allow code to be executed repeatedly (i.e. in iterations).

- **for** loop: execute a block of code for each element in a sequence.

```
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    print(number)
```

- **while** loop: execute a block of code as long as a condition is true.

```
number = 1
while number <= 5:
    print(number)
    number += 1
```

Loops

You can skip or stop both `for` and `while` loops from within loops for more control.

- **continue** statement: skip the rest of the code in the current iteration and move to the next iteration.

```
for number in range(10):
    if number % 2 == 0:
        continue
    print(number)
```

- **break** statement: stop the loop from executing and move to the next block of code.

```
for number in range(10):
    if number == 5:
        break
    print(number)
```

Loops

Tricks for loops:

- **else** statement: execute a block of code when the loop is finished without being stopped by a `break` statement.

```
for number in range(10):
    if number == 5:
        break
    print(number)
else:
    print("Loop finished")
```

Loops

Tricks for loops:

- **else** statement: execute a block of code when the loop is finished without being stopped by a `break` statement.

```
for number in range(10):
    if number == 5:
        break
    print(number)
else:
    print("Loop finished")
```

- Try to comment out the highlighted line and see what happens.

Loops

Tricks for loops:

- **else** statement: execute a block of code when the loop is finished without being stopped by a `break` statement.

```
for number in range(10):
    if number == 5:
        break
    print(number)
else:
    print("Loop finished")
```

- Try to comment out the highlighted line and see what happens.
- **List comprehension**: list comprehension provides a more efficient way of looping.

```
squares = [number**2 for number in range(10)]
print(squares)
```

Loops

Useful generators for loops:

- **range()** function: generate a sequence of integers.

```
for number in range(5, 10, 2):
    print(number)
```

- **enumerate()** function: generate a sequence of index-value pairs.

```
for index, value in enumerate(["a", "b", "c"]):
    print(index, value)
```

- **zip()** function: generate a sequence of element-wise pairs.

```
for a, b in zip([1, 2, 3], ["a", "b", "c"]):
    print(a, b)
```

Functions

Repeated code can be encapsulated into a function for reuse.

- **Functions** are blocks of reusable code that can be called to perform a specific task.
- Similar to functions in mathematics, a function has the input (i.e., **argument**) and the output (i.e., **return value**).

```
def greet(name):
    return f"Hello, {name}!"
```

- **Default values** can be set for arguments.
- Functions can be called with **positional arguments** or **keyword arguments**.

```
def show_info(name, age=30, height=1.75):
    return f"{name} is {age} years old and {height:.2f} meters tall."

print(show_info("Alice"))
print(show_info("Bob", 25))
print(show_info(height=1.80, name="Charlie"))
```

Functions

- Arguments can be **positional-only**, **keyword-only**, or **positional-or-keyword**.

```
def show_info(old_name, new_name, /, age, city, *, height, weight):
    return f"{old_name} is now {new_name}, {age} years old,"
           f" from {city}, {height:.2f} meters tall,"
           f" and {weight:.2f} kg heavy."

print(show_info("Alice", "Bob", 25, city="Munich", height=1.80, weight=70))
```

- Sometimes you don't know how many arguments will be passed to a function.

```
def get_info(name, *args, **kwargs):
    print(f"Hey, {name}!")
    print(args)
    print(kwargs)

get_info("Alice", 30, 1.75, city="Munich", weight=70)
```

Functions

- Functions can return multiple values.
- Variables inside a function are **local** by default, and cannot be accessed outside.

```python
def get_info(a, b, c):
    return a, b, c

name, age, height = get_info("Alice", 30, 1.75)
print(a, b, c)
print(name, age, height)
```

Functions

- A function can call itself inside, which is called **recursion**.

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

print(factorial(5))
```

- A special function called **lambda** can be used to create small anonymous functions.
- Similar to mathematical functions, Python functions can be nested.

```
f = lambda x: x**2
g = lambda x, y: x + y
print(f(3))
print(f(g(3)))
```

Functions

Python has a rich set of built-in functions.

Commonly used built-in functions:

Check the official documentation for more built-in functions.

Functions

Python has a rich set of built-in functions.

Commonly used built-in functions:

- `print()`, `input()`, `len()`, `type()`, `zip()`, `enumerate()`, `range()`, `int()`, `float()`, `str()`, `bool()`

Check the official documentation for more built-in functions.

Functions

Python has a rich set of built-in functions.

Commonly used built-in functions:

- `print()`, `input()`, `len()`, `type()`, `zip()`, `enumerate()`, `range()`, `int()`, `float()`, `str()`, `bool()`
- `list()`, `tuple()`, `set()`, `dict()`: create a list, tuple, set, or dictionary

Check the official documentation for more built-in functions.

Functions

Python has a rich set of built-in functions.

Commonly used built-in functions:

- `print()`, `input()`, `len()`, `type()`, `zip()`, `enumerate()`, `range()`, `int()`, `float()`, `str()`, `bool()`
- `list()`, `tuple()`, `set()`, `dict()`: create a list, tuple, set, or dictionary
- `sum()`, `max()`, `min()`, `abs()`, `round()`, `pow()`, `divmod()`, ...: mathematical operations

Check the official documentation for more built-in functions.

Functions

Python has a rich set of built-in functions.

Commonly used built-in functions:

- `print()`, `input()`, `len()`, `type()`, `zip()`, `enumerate()`, `range()`, `int()`, `float()`, `str()`, `bool()`
- `list()`, `tuple()`, `set()`, `dict()`: create a list, tuple, set, or dictionary
- `sum()`, `max()`, `min()`, `abs()`, `round()`, `pow()`, `divmod()`, ...: mathematical operations
- `sorted()`, `reversed()`, `map()`, `filter()`, `all()`, `any()`, ...: sequence operations

Check the official documentation for more built-in functions.

Expanding your Toolbox!

Python has a rich set of libraries/modules for various purposes.

- **Libraries/modules** are code packages that you can load within your script in order to reuse functions that other programmers have already developed and made available.
- Python comes with some **internal libraries** (see https://docs.python.org/3/library/).
- **External libraries** can be installed by yourself through package managers like `pip` or `conda`.

Expanding your Toolbox!

Modules can be loaded using the `import` statement.

Commonly used internal libraries:

Expanding your Toolbox!

Modules can be loaded using the `import` statement.

Commonly used internal libraries:

- `math`: mathematical functions and constants

Expanding your Toolbox!

Modules can be loaded using the `import` statement.

Commonly used internal libraries:

- `math`: mathematical functions and constants

```
import math
print(math.pi) # constant
print(math.sqrt(2), math.sin(3.2), math.log(10)) # functions
```

Expanding your Toolbox!

Modules can be loaded using the `import` statement.

Commonly used internal libraries:

- `math`: mathematical functions and constants

```
import math
print(math.pi) # constant
print(math.sqrt(2), math.sin(3.2), math.log(10)) # functions
```

- `random`: random number generation

Expanding your Toolbox!

Modules can be loaded using the `import` statement.

Commonly used internal libraries:

- `math`: mathematical functions and constants

```
import math
print(math.pi) # constant
print(math.sqrt(2), math.sin(3.2), math.log(10)) # functions
```

- `random`: random number generation

```
import random
print(random.random(), random.randint(1, 10), random.choice(["a", "b", "c"]))
```

Expanding your Toolbox!

Modules can be loaded using the `import` statement.

Commonly used internal libraries:

- `math` : mathematical functions and constants

```
import math
print(math.pi) # constant
print(math.sqrt(2), math.sin(3.2), math.log(10)) # functions
```

- `random` : random number generation

```
import random
print(random.random(), random.randint(1, 10), random.choice(["a", "b", "c"]))
```

- `datetime` : date and time manipulation

Expanding your Toolbox!

Modules can be loaded using the `import` statement.

Commonly used internal libraries:

- `math` : mathematical functions and constants

```
import math
print(math.pi) # constant
print(math.sqrt(2), math.sin(3.2), math.log(10)) # functions
```

- `random` : random number generation

```
import random
print(random.random(), random.randint(1, 10), random.choice(["a", "b", "c"]))
```

- `datetime` : date and time manipulation

```
import datetime
print(datetime.datetime.now(), datetime.date.today())
```

Expanding your Toolbox!

Commonly used internal libraries:

Expanding your Toolbox!

Commonly used internal libraries:

- `iterools`: functions creating iterators for efficient looping

Expanding your Toolbox!

Commonly used internal libraries:

- `itertools`: functions creating iterators for efficient looping

```
import itertools
print(list(itertools.permutations([1, 2, 3], 2)))
```

Expanding your Toolbox!

Commonly used internal libraries:

- `iterools`: functions creating iterators for efficient looping

```
import itertools
print(list(itertools.permutations([1, 2, 3], 2)))
```

- `functools`: higher-order functions and operations on callable objects

Expanding your Toolbox!

Commonly used internal libraries:

- `iterools`: functions creating iterators for efficient looping

```
import itertools
print(list(itertools.permutations([1, 2, 3], 2)))
```

- `functools`: higher-order functions and operations on callable objects

```
import functools
print(functools.reduce(lambda x, y: x * y, [1, 2, 3, 4]))
```

Expanding your Toolbox!

Commonly used internal libraries:

- `iterools`: functions creating iterators for efficient looping

```
import itertools
print(list(itertools.permutations([1, 2, 3], 2)))
```

- `functools`: higher-order functions and operations on callable objects

```
import functools
print(functools.reduce(lambda x, y: x * y, [1, 2, 3, 4]))
```

- `collections`: container datatypes

Expanding your Toolbox!

Commonly used internal libraries:

- `iterools`: functions creating iterators for efficient looping

```
import itertools
print(list(itertools.permutations([1, 2, 3], 2)))
```

- `functools`: higher-order functions and operations on callable objects

```
import functools
print(functools.reduce(lambda x, y: x * y, [1, 2, 3, 4]))
```

- `collections`: container datatypes

```
import collections
print(collections.Counter("hello"))
```

Expanding your Toolbox!

Commonly used internal libraries:

Expanding your Toolbox!

Commonly used internal libraries:

- `os`: operating system interfaces

Expanding your Toolbox!

Commonly used internal libraries:

- `os` : operating system interfaces

```
import os
print(os.getcwd(), os.listdir(), os.path.exists("hello.py"))
```

Expanding your Toolbox!

Commonly used internal libraries:

- `os` : operating system interfaces

```
import os
print(os.getcwd(), os.listdir(), os.path.exists("hello.py"))
```

- `sys` : system-specific parameters and functions

Expanding your Toolbox!

Commonly used internal libraries:

- `os` : operating system interfaces

```
import os
print(os.getcwd(), os.listdir(), os.path.exists("hello.py"))
```

- `sys` : system-specific parameters and functions

```
import sys
print(sys.version, sys.platform, sys.argv)
```

Expanding your Toolbox!

Commonly used internal libraries:

- `os` : operating system interfaces

```
import os
print(os.getcwd(), os.listdir(), os.path.exists("hello.py"))
```

- `sys` : system-specific parameters and functions

```
import sys
print(sys.version, sys.platform, sys.argv)
```

- `argparse` : command-line parsing

Expanding your Toolbox!

Commonly used internal libraries:

- `os` : operating system interfaces

```
import os
print(os.getcwd(), os.listdir(), os.path.exists("hello.py"))
```

- `sys` : system-specific parameters and functions

```
import sys
print(sys.version, sys.platform, sys.argv)
```

- `argparse` : command-line parsing

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("name")
args = parser.parse_args()
print(args.name)
```

Working with Files

Python has built-in functions for file operations.

- `open()` function: open a file and return a file object.
- `read()` method: read the entire file content.
- `readline()` method: read a single line from the file.
- `readlines()` method: read all lines from the file and return a list.
- `close()` method: close the file.

```
file = open("hello.txt", "r") # `r` for reading, `w` for writing, `a` for appending
first_line = file.readline()
content = file.read()
file.close()
print(first_line)
print(content)
```

Working with Files

Python has built-in functions for file operations.

- `write()` method: write a string to the file.

```
file = open("hello.txt", "w")
file.write("Hello, world!")
file.close()
```

- Remember to **close the file** after reading or writing!
- `with` statement (**recommended**): automatically close the file after reading or writing.

```
with open("hello.txt", "r") as file:
    for line in file:
        print(line)
```

Oops, Errors!

Python has built-in exceptions for error handling.

- **Exceptions** are events that can modify the flow of control through a program.
- **Error handling** is the process of responding to exceptions.
  - `try` block: the code that may raise an exception.
  - `except` block: the code that handles the exception.

```
try:
    print(1 / 0)
except ZeroDivisionError as e:
    print(e)
```

Oops, Errors!

Python has built-in exceptions for error handling.

- Different types of exceptions can be handled separately.
- `else` block: the code that is executed if no exception is raised.
- `finally` block: the code that is always executed.

```
f = lambda x: 1 / x
try:
    # print(f(name='Alice'))
    print(f(0))
except ZeroDivisionError as e:
    print('Denominator cannot be zero')
except Exception as e: # catch all other exceptions
    print(e)
else:
    print("No exception")
finally:
    print("Always executed")
```

Oops, Errors!

Python has built-in exceptions for error handling.

- `raise` statement: raise an exception manually.

```
def divide(x, y):
    if y == 0:
        raise ZeroDivisionError("Denominator cannot be zero")
    return x / y

divide(1, 0)
```

- `assert` statement: raise an exception if a condition is false.

```
old_password = "123"
new_password = "1234"
assert new_password != old_password, "New password cannot be the same as the old one"
```