

Python Training

Table of Contents

PART A: Basic Python	1
1. Requirements	2
1.1. Python Installation	2
1.2. Python IDE	2
1.3. Version Control	2
1.4. Style Guide	2
2. Introduction	3
2.1. History	3
2.2. Characteristics	3
2.3. Use Cases	4
2.4. Packages	5
2.5. Hello, world!	5
2.6. print() under the hood	5
2.7. Statements	6
2.8. Expressions	6
2.9. Zen of Python	7
2.10. Importing Modules	7
3. Variables	9
3.1. Numerics	10
3.2. Booleans	11
3.3. Lists	11
3.4. Tuples	11
3.5. Sets	12
3.6. Dictionaries	12
3.7. Strings	12
3.8. None	16
3.9. Mutable	16
3.10. Immutable	17
4. Comments	18
4.1. Single-line Comments	18
4.2. Multi-line Comments	18
5. Operators	19
5.1. Arithmetic Operators	19
5.2. Comparison Operators	20
5.3. Logical Operators	21
5.4. Identity Operators	21
5.5. Membership Operators	22
5.6. Bitwise Operators	22

5.7. Assignment Operators	23
5.8. Conditional Operators	25
5.9. Operator Precedence	25
6. User Input	28
6.1. User Input in Python 2	28
6.2. User Input in Python 3	28
6.3. Writing Compatible Code	28
7. Control Flow	30
7.1. if Statement	30
7.2. for Statement	30
7.3. while Statement	31
7.4. break Statement	31
7.5. continue Statement	31
7.6. pass Statement	32
8. Functions	33
8.1. Positional Arguments	33
8.2. Named Arguments	34
8.3. Default Arguments	34
8.4. Variable Arguments	34
8.5. Unpacking Arguments	35
8.6. Variable Scope	35
8.7. Nested Functions	36
8.8. Closures	36
8.9. Decorator Functions	37
8.10. Factory Functions	39
8.11. Memoization	40
8.12. Function Annotations	40
8.13. Function Attributes	41
8.14. Callback Functions	42
8.15. Recursive Functions	42
8.16. Lambda Functions	43
8.17. Positional-Only Arguments	44
8.18. Keyword-Only Arguments	45
8.19. Function Introspection	45
9. Classes	48
9.1. Class structure	48
9.2. Initializer	49
9.3. Constructor	51
9.4. Class instance	54
9.5. Class as an object	56
9.6. Class as a namespace	61

9.7. Properties	62
9.8. Superclass	62
9.9. Class inheritance	63
9.10. Method resolution order	65
9.11. Abstract class	68
9.12. Interface class	71
9.13. Mixin class	72
9.14. Class as a decorator	73
9.15. Nested classes	75
9.16. Named Constructors	76
9.17. Introspection	79
10. Docstrings	80
10.1. Class docstrings	80
10.2. Function docstrings	81
10.3. Module docstrings	81
11. Debugging	84
11.1. Breakpoints	84
11.2. Conditional breakpoints	84
11.3. Watchlists	85
12. Linting	87
PART B: Advanced Python	88
1. Requirements	89
1.1. Python Installation	89
1.2. Python IDE	89
1.3. Version Control	89
1.4. Style Guide	89
2. OOP Principles	90
2.1. Abstraction (Modeling)	90
2.2. Inheritance (Code Reuse)	91
2.3. Encapsulation (Data Hiding)	93
2.4. Polymorphism (Flexibility)	95
2.5. Association (Relationships)	97
3. SOLID Design	99
3.1. Single Responsibility Principle	99
3.2. Open/Closed Principle	102
3.3. Liskov Substitution Principle	104
3.4. Interface Segregation Principle	106
3.5. Dependency Inversion Principle	111
4. Design Patterns	114
4.1. Creational Patterns	114
4.2. Structural Patterns	123

4.3. Behavioral Patterns	135
5. System Architecture	155
5.1. Development Environment	155
5.2. Deployment Strategy	155
5.3. Functional Requirements	156
5.4. Non-Functional Requirements	156
5.5. Architecture and Design	157
5.6. Development Process	159
5.7. Quality Assurance	160
6. Logging	162
6.1. Logging Hierarchy	162
6.2. Logging Levels	163
6.3. Basic Configuration	163
6.4. Advanced Configuration	164
6.5. Best Practices	169
7. Exceptions	171
7.1. Handling Exceptions	171
7.2. Program Flow	173
7.3. Chaining Exceptions	175
7.4. Organizing Exceptions	176
7.5. Built-in Exceptions	178
7.6. Best Practices	184
8. Dunder Methods	186
8.1. Object representation	186
8.2. Custom behavior of classes and instances	188
8.3. Custom behavior of built-in types	189
8.4. Operator overloading	191
8.5. Iterator protocol	193
8.6. Context manager protocol	194
8.7. Best Practices	195
9. Advanced Data	196
9.1. Data Manipulation	196
9.2. Binary Serialization	200
9.3. Text Serialization	203
9.4. Data Conversion	203
9.5. Data Classes	213
10. Import System	226
10.1. Namespace	226
10.2. Module Search Path	229
10.3. Packages	230
10.4. Relative Imports	231

10.5. Absolute Imports	232
10.6. Main Modules	233
10.7. Main Packages	234
10.8. Import Process	234
10.9. Importlib	236
10.10. Naming Conventions	236
11. Compatibility	237
11.1. Syntax	237
11.2. Using <code>__futures__</code>	237
11.3. Using <code>six</code>	238
12. Testing	240
12.1. unittest	240
12.2. pytest	241
12.3. Coverage	242
12.4. Hypothesis	243
12.5. tox	243
References	245
1. Bibliography	246
2. Links	247
2.1. Beginner	247
2.2. Intermediate	247
2.3. Expert	248
2.4. Libraries	249

PART A: Basic Python

Chapter 1. Requirements

The following pre-requisites are required to complete this module:

1.1. Python Installation

- [Python 3.7 or higher](#) (*recommended*)
- [Python 2.7.18](#)

1.2. Python IDE

- [PyCharm](#) (*recommended*)
- [Visual Studio Code](#)

1.3. Version Control

- [Git Client](#) (*recommended*)
- [GitHub Account](#) (*recommended*)
- [SVN Client](#)

1.4. Style Guide

- [PEP-08](#) (*mandatory*)
- [Google Style Guide](#) (*mandatory*)

Chapter 2. Introduction

Python is a high-level, interpreted, interactive, and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where other languages use punctuation. It has fewer syntactical constructions than other languages.

Some key features of Python are:

- Python is a dynamically typed language. This means that you do not need to declare variables before using them, or declare their type. Every variable in Python is an object.
- Python is well suited to object-oriented programming in that it allows the definition of classes along with composition and inheritance. Python does not have access specifiers (like C++'s public, private).
- In Python, functions are first-class objects. This means that they can be assigned to variables, returned from other functions, and passed into functions as arguments.
- Python has a large and comprehensive standard library.
- Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming. It features a fully dynamic type system and automatic memory management, similar to that of Scheme, Ruby, Perl, and Tcl.

2.1. History

Python was conceived in the late 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC language (itself inspired by SETL), capable of exception handling and interfacing with the Amoeba operating system. Python itself is written in C, Cython, and the Python programming language. Python was named after the BBC TV show Monty Python's Flying Circus.

Python 2 and Python 3 are two different versions of the Python programming language. Python 3 was released in 2008 and Python 2 was released in 2000. Python 3 is not backward compatible with Python 2. This means that code written in Python 2 may not work in Python 3 and vice versa. Python 2 reached its end of life on January 1, 2020. Python 2 should only be used if you need to maintain legacy code written in Python 2.

Python ranks consistently as one of the most popular programming languages. In January 2021 it ranked third in the TIOBE index, a measure of popularity of programming languages, behind C and Java. It was selected Programming Language of the Year in 2007, 2010, and 2018. Python is used in many application domains. It is used extensively in the scientific and financial industry, and for web development and software development. Python is also used in artificial intelligence and data science.

2.2. Characteristics

- Python is a **general-purpose language**. This means that you can use Python to write any type of program, from simple command-line scripts to complex web applications.

- Python is an **interpreted language**. This means that you do not need to compile your program before executing it. The Python interpreter reads your program and executes it one line at a time until the end of the file. This makes it easier to test your code and debug it.
- Python is a **high-level language**. When you write programs in Python, you do not need to worry about the low-level details such as managing the memory used by your program. This makes it easier to focus on the solution to the problem you are trying to solve.
- Python is an **object-oriented language**. This means that you can define classes and create objects that encapsulate data and functionality. This makes it easier to write reusable code.
- Python is a **dynamically typed language**. This means that you do not need to declare variables before using them, or declare their type. Every variable in Python is an object.
- Python is a **multi-paradigm language**. This means that it supports multiple programming paradigms, including procedural, object-oriented, and functional programming. It features a fully dynamic type system and automatic memory management, similar to that of Scheme, Ruby, Perl, and Tcl.
- Python is a **cross-platform language**. This means that you can write Python code on one platform and run it on any other platform without making any changes to the code. This makes it easier to develop and deploy your code on multiple platforms.

2.3. Use Cases

- Google (one of the three main languages used for development)
- Facebook (backend development)
- Instagram (written entirely in Python)
- Netflix (recommendation engine)
- Spotify (data analysis)
- Dropbox (desktop client)
- Reddit (rewritten in Python in 2005)
- Quora (web backend)
- Pinterest (analytics platform)
- YouTube (wvideo transcoding system)
- IBM (cloud computing platform)
- Nokia (cloud computing platform)
- NASA (Integrated Planning System)
- CERN (Large Hadron Collider)
- Mozilla (Firefox browser)
- and many more...

2.4. Packages

- Django, a web framework
- Flask, a micro web framework
- Matplotlib, a popular plotting library
- NumPy, a package for scientific computing
- Pandas, a package for data analysis
- Pygame, a set of Python modules designed for writing games
- PyTorch, a machine learning library
- SciPy, a library of scientific and numerical tools
- TensorFlow, a machine learning library
- requests, a package for making HTTP requests
- and many more...

2.5. Hello, world!

Our first program will greet us and tell us the result of the operation $1+1$.

```
print("Hello world!")  
print(1 + 1)
```

Now if after evaluation of the code, we will see that...

- we tell the computer what to do
- `print()` is a pre-defined code ready to be used by the developer
- `print()` uses a value to show it on the screen
- `print()` can be used with pre-defined values such as 'Hello, world!'
- `print()` can be used with the result of operations such as $1 + 1$

If we elaborate further we will see that a program generally is a...

- A sequence of instructions called statements
- Each statement might have expressions that return results
- Each statement might use pre-defined code blocks called functions

2.6. `print()` under the hood

- The `print()` function is used to print a value to the screen.
- The `print()` function is a built-in function, which means that it is available by default.

```
# Example: print() function

# By default, print() ends with a newline character.
print("Hello")
print("World")

# You can change this behavior by specifying the end parameter.
print("Hello", end="")
print("World", end="")

# A print statement with no arguments prints a newline character.
print()

# You can also specify the separator parameter.
print("Hello", "World", sep="")
print("Hello", "World", sep=" ")
```

2.7. Statements

Statements are instructions that perform an action. It is executed for its side effects, such as modifying variables, controlling the flow of execution, or producing output. Statements do not have a value or result that can be used further in the code.

```
x = 5
print(x)
```

In the above example, we have two statements. The first statement assigns the value 5 to the variable `x`. The second statement prints the value of `x` to the screen. The first statement does not produce any output, but it does have a side effect of changing the value of `x`. The second statement produces output, but it does not have a side effect.

2.8. Expressions

Expressions are a combination of values, variables, operators, and function calls that produce a value when evaluated. Expressions can be used wherever a value is expected in Python, such as assigning a value, passing arguments to a function, or printing a value.

```
1 + 2 + 3          # Simple expression
result = 1 + 2 + 3  # Assign statement after expression evaluation
```

In the above example, the first line is an expression that evaluates to the value 6. The second line is a statement that assigns the value of the expression `1 + 2 + 3` to the variable `result`.

2.9. Zen of Python

The Zen of Python is a collection of 19 "guiding principles" for writing computer programs that influence the design of the Python programming language. It is written as a poem, with each line of the poem corresponding to one of the principles. The Zen of Python is included in the Python interpreter as an Easter egg. You can view it by typing `import this` in the Python interpreter.

```
# Example: Use import to access the zen_of_python module

import this
```

The statement `import this` will do the following:

- It will provide all elements of the `this` module
- If there is executable code in the module, it will be executed

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

2.10. Importing Modules

Python modules are files that contain Python code. They can be used to organize code into logical units.

Each module has its own namespace - a region of code where the defined objects can access the given values using their names. Python can run multiple namespaces simultaneously, so objects with the same name from different namespaces (modules) do not influence each other.

Modules can be imported into other modules using the `import` statement. The `import` statement takes the name of the module to import as an argument.

Using `import` statement still does not import the names of the objects in the importing module namespace, but makes them accessible calling them via their module name.

Another way to import modules is to use the `from <module> import <object>` statement. This allows you to import specific objects from a module into the current namespace. You can also import all objects from a module into the current namespace using the `from <module> import *` statement.

For example, if you have a module named `math.py`, you can import it using the statement `import math`. You can also import specific objects from a module using the `from` statement.

```
# Example : Use import to access python modules

import math
from math import pi

print(math.pi)
print(pi)
```

Chapter 3. Variables

Let's print the same text 5 times on the screen using the print function.

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

Now if we want to modify the text, we have to change all five lines. Let's modify our program a little bit to be more reusable. Now we make the function print to work with any text possible. For this purpose, we will use a variable. A variable is a data container that can be reused many times in the program.

```
text = 'Hello world!'
print(text)
print(text)
print(text)
print(text)
print(text)
```

In the code above the variable text is declared and initialized. After this, it may be used many times in the program. Now we have to change only one line of code in case we need to change the message printed on the screen.

Python has several pre-defined variable types. Let's use a more complex example with the print function and see how it works:

```
var = 100
print(var, type(var))
```

We see that the variable is from type integer, but they are preceded by the word class. The class tells Python what type of data the variable stores and how Python can operate with the data. This can be demonstrated by using numbers of different types.

```
# The operator + adds two float numbers
a = 1.0
b = 1.0
print(a + b)
```

In the example above Python analyses the code and deducts that a and b are floating point numbers and uses the float class to check how to add them. Now let's change the variable types to complex:

```
# The operator + adds two complex numbers
a = 1 + 1j
b = 1 + 1j
print(a + b)
```

The code is analyzed and Python deduces that now `a` and `b` are complex numbers and thus another type of data. In this case, Python will use the addition operation for this data class.

3.1. Numerics

Numeric variables are variables that hold numeric values. Python supports three types of numeric values:

- Integer numbers
- Floating-point numbers
- Complex numbers

3.1.1. Integers

Integers are whole numbers, such as 1, 2, 3, 4, 5, etc. They can be positive or negative. They can also be written in binary, octal, or hexadecimal notation. Binary numbers are prefixed with `0b`, octal numbers are prefixed with `0o`, and hexadecimal numbers are prefixed with `0x`.

```
decimal_var = 10
binary_var = 0b10
octal_var = 0o10
hexadecimal_var = 0x10
print(decimal_var, binary_var, octal_var, hexadecimal_var)
```

3.1.2. Floating-point numbers

Floating-point numbers are numbers with a fractional part, such as 1.0, 2.0, 3.0, etc. They can also be written in scientific notation, with `e` indicating the power of 10.

```
float_1 = 1.0
float_2 = 1.1e1
print(float_1)
print(float_2)
```

3.1.3. Complex numbers

Complex numbers are numbers with a real and imaginary part, such as `1+2j`, `2+4j`, `3+6j`, etc.

```
z1 = 1 + 2j
```



```
z2 = 2 + 4j
print(z1 + z2)
```

3.2. Booleans

The boolean type is a special type that can only have two values: `True` and `False`. It is used to represent the truth values of logic and comparison operations. In Python, boolean values are written as `True` and `False` (the first letter is capitalized).

```
var = False
print(var)

var = True
print(var)
```

3.3. Lists

A list is a collection of items that are ordered and changeable. Lists are written with square brackets `[]`. Each item in a list is separated by a comma `,`. Lists can contain items of any data type, including other lists. Lists are mutable, meaning that you can change the values of items in a list. You can also add and remove items from a list. Lists are zero-indexed, meaning that the first item in a list has index 0.

```
# Create a list of 3 elements
a = [1, 2, 3]
print(a)

# Create a list of lists
b = [a, a]
print(b)
```

3.4. Tuples

A tuple is a collection of items that are ordered and unchangeable. Tuples are written with parentheses `()`. Each item in a tuple is separated by a comma `,`. Tuples can contain items of any data type, including other tuples. Tuples are immutable, meaning that you cannot change the values of items in a tuple. Tuples are zero-indexed, meaning that the first item in a tuple has index 0.

```
# Create a tuple of 3 elements
a = (1, 2, 3)
print(a)

# Create a tuple of tuples
b = (a, a)
```

```
print(b)
```

3.5. Sets

A set is a collection of items that are unordered and unindexed. Sets are written with curly braces `{}`. Each item in a set is separated by a comma `,`. Sets can contain items of any data type, but they cannot contain duplicate items. Sets are mutable, meaning that you can change the values of items in a set.

```
# Create a set of 3 elements
a = {1, 1, 2, 2, 3, 3}
print(a)

# Create a set of sets
b = (a, a)
print(b)
```

3.6. Dictionaries

A dictionary is a collection of items that are unordered, changeable, and indexed. Dictionaries are written with curly braces `{}`. Each item in a dictionary is written as a key-value pair, separated by a colon `:`. Each key-value pair is separated by a comma `,`. Dictionaries can contain items of any data type, but they cannot contain duplicate keys. Dictionaries are mutable, meaning that you can change the values of items in a dictionary.

```
a = {'a': 1, 'b': 2, 'c': 3}
print(a)

a = {'a':a, 'b': a}
print(a)
```

3.7. Strings

String variables are variables that hold string values. A string is a sequence of characters enclosed in single quotes `'` or double quotes `"`.

```
string1 = 'Hello'
string2 = "World"
string3 = "!"
```

3.7.1. Concatenation

String concatenation refers to the process of combining two or more strings into a single string.

```
string1 = 'Hello'
string2 = "World"
string3 = "!"
result = string1 + " " + string2 + string3
print(result)
```

3.7.2. Interpolation

String interpolation refers to the process of embedding expressions or variables within string literals to create formatted strings. It allows you to dynamically insert values into a string.

```
# Python 2.7+
first_name = "Branimir"
last_name = "Georgiev"
age = 25
print("My name is {0} {1} and I am {2} years old".format(first_name, last_name, age))

# Python 3.6+
first_name = "Branimir"
last_name = "Georgiev"
age = 25
print(f"My name is {first_name} {last_name} and I am {age} years old")
```

3.7.3. Escaping

Escaping refers to the process of inserting a special character into a string to change the meaning of the string literal (usually reserved by the language itself). It is done by prefixing the special character with a backslash `\`.

```
print("Hello\nWorld")          # Print new line inside string
print("He said, \"Goodbye\"")  # Print double quotes inside string
```

Some of the most common escape sequences are:

- `\n` - newline
- `\\` - backslash
- `\'` - single quote
- `\"` - double quote

3.7.4. Indexing

Indexing refers to the process of accessing individual characters in a string by their position. The position of a character in a string is called its index. The first character in a string has index 0, the second character has index 1, and so on.

```
string = "Hello, world!"
print(string[0])    # Print first character
print(string[1])    # Print second character
print(string[-1])   # Print last character
```

3.7.5. Slicing

String slicing in Python allows you to extract a portion of a string by specifying a range of indices. The format of the slice operator is [**<start>**: **<end>**: **<step>**].

In slicing, the start index is inclusive, and the end index is exclusive. If the start index is not specified, it defaults to the beginning of the string. If the end index is not specified, it defaults to the end of the string. The step value indicates the stride or the number of characters to skip between each character in the slice.

```
text = "0123456789ABCDEF"

print(text[0:5])    # Print first 5 characters
print(text[7:])     # Print characters from index 7 to the end
print(text[:5])     # Print characters from the beginning to index 4
print(text[::2])    # Print every second character
print(text[::-1])   # Print characters in reverse order
print(text[1:10:2]) # Print every second character from index 1 to 10
```

3.7.6. Case

Python provides several methods for changing the case of a string. The most common ones are:

- **upper()** - converts all characters to uppercase
- **lower()** - converts all characters to lowercase
- **title()** - converts the first character of each word to uppercase and the rest to lowercase
- **capitalize()** - converts the first character to uppercase and the rest to lowercase
- **swapcase()** - converts uppercase characters to lowercase and vice versa

```
text = "HELLO world!"
print(text.upper())
print(text.lower())
print(text.title())
print(text.capitalize())
print(text.swapcase())
```

3.7.7. Length

The length of a string is the number of characters in the string. You can use the **len()** function to get

the length of a string.

```
text = "0123456789"
print(len(text)) # 10 characters
```

3.7.8. Splitting

The `split()` method splits a string into a list of strings based on a delimiter. The delimiter is a string that separates the individual strings in the list. If no delimiter is specified, the string is split on whitespace. The `splitlines()` method splits a string into a list of strings based on the line break character `\n`.

```
# Split the string into a list of tokens using the split() method.
text = "1 2 3 4 5 6 7 8 9 10"
print(text.split())

# Split the string into a list of tokens using the split() method and the separator
argument.
text = "1,2, 3 4 5 6 7 8 9 10"
print(text.split(', '))

# Split the multi-line string into a list of tokens using the splitlines() method.
text = """ First line
Second line
Third line
"""
print(text.splitlines())
```

3.7.9. Joining

The `join()` method joins a list of strings into a single string using the specified delimiter. It is the inverse of the `split()` method. The `join()` method is called on the delimiter and takes the list of strings as an argument.

```
tokens = '1 2 3 4 5 6 7 8 9'.split()
print(tokens)

# Join the list of tokens into a string using the join() method and no separator.
text = ''.join(tokens)
print(text)

# Join the list of tokens into a string using the join() method and a space separator.
text = ' '.join(tokens)
print(text)

# Join the list of tokens into a string using the join() method and a comma separator.
text = ','.join(tokens)
```

```
print(text)
```

3.8. None

The None type is a special type that can only have one value: **None**. It is used to represent the absence of a value. None is often used as a placeholder value when you need to initialize a variable but don't have a value for it yet. None is also returned by functions that don't explicitly return a value. For example, the **print()** function returns None.

```
# Create the variable
a = None
print(a)

# The print function returns None
print(print())
```

3.9. Mutable

Mutable variables are objects whose values can be changed after they are created. In Python, the following types are mutable:

- Lists
- Sets
- Dictionaries

```
# Example: Mutable variables

# The list [1, 2, 3] has an id and it is mutable
test = [1, 2, 3]
print("ID: {}".format(id(test)))

# Assigning a new list to the variable will change the id
test = [1, 2, 3, 4]
print("ID after reassignment: {}".format(id(test)))

# Adding an element to the list will not change the id
test.append(5)
print("ID after append: {}".format(id(test)))

# This statement is not a copy, it is a reference.
# This is especially important when passing variables to functions.
reference = test
print("ID of reference: {}".format(id(reference)))

def test_function(x):
```

```
print("ID in a function (called by reference): {}".format(id(x)))
```

```
test_function(test)
```

3.10. Immutable

Immutable objects are objects whose values cannot be changed after they are created. In Python, the following types are immutable:

- Integers
- Floating-point numbers
- Complex numbers
- Booleans
- Tuples
- Strings

```
# Example: Mutable vs Immutable
```

```
# The integer 1 has an id and it is immutable
test = 1
print("Testing immutable constant 1 (int)")
print("ID of test   : {}".format(id(test)))
print("ID of 1      : {}".format(id(1)))

print("")
```

```
# The integer 2 has an id and it is immutable
test = 2
print("Testing immutable constant 2 (int)")
print("ID of test   : {}".format(id(test)))
print("ID of 2      : {}".format(id(2)))
```

Chapter 4. Comments

Comments are lines of text that are ignored by the Python interpreter. They are used to document code and make it easier to understand. Comments can be single-line or multi-line. Single-line comments start with a hash `#` and continue until the end of the line. Multi-line comments start and end with three quotes `"""`.

4.1. Single-line Comments

```
# This code will print a message and the sum of two numbers
print("Hello world!")
print(1 + 1)
```

4.2. Multi-line Comments

```
"""
This is a multi-line comment.

The following code will print the message `Hello, world!`
and then print the sum of two numbers.
"""

print("Hello world!")
print(1 + 1)
```


Chapter 5. Operators

Operators are special symbols that perform operations on one or more operands. Python supports the following types of operators:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

5.1. Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on numeric operands. Bear in mind that different major versions of Python may have different operators. For example, Python 2 uses `/` for integer division, while Python 3 uses `//` for integer division. For simplicity, we will only use Python 3 operators in this book.

- `+` - addition
- `-` - subtraction
- `*` - multiplication
- `/` - division (Note: Python 2 uses `/` for integer division)
- `//` - floor division (quotient)
- `%` - modulus (remainder)
- `**` - exponentiation (power)

```
# Example: Arithmetic Operators

# Initialize variables
a, b = 10, 20

# Sum
print(a+b)

# Subtraction
print(a-b)

# Multiplication
print(a*b)

# Division (Python 3: float, Python 2: int)
```

```
print(a/b)

# Floor Division (integer division)
print(a//b)

# Modulus (remainder)
print(a % b)

# Exponentiation (a to the power of b)
print(a**b)
```

5.2. Comparison Operators

Comparison operators are used to compare the values of two operands. They return a boolean value (**True** or **False**) depending on whether the comparison is true or false. The following comparison operators are supported in Python:

- **==** - equal to
- **!=** - not equal to
- **>** - greater than
- **<** - less than
- **>=** - greater than or equal to
- **<=** - less than or equal to

```
# Example: Comparison Operators

# Initialize variables
a, b = 10, 20

# Equal
print(a == b)

# Not equal
print(a != b)

# Greater than
print(a > b)

# Less than
print(a < b)

# Greater than or equal to
print(a >= b)

# Less than or equal to
print(a <= b)
```

5.3. Logical Operators

Logical operators are used to combine the results of two or more comparison operations. They return a boolean value (**True** or **False**) depending on whether the result of the logical operation is true or false. The following logical operators are supported in Python:

- **and** - logical AND
- **or** - logical OR
- **not** - logical NOT

```
# Example: Logical Operators

# Initialize variables
a, b = 10, 20

# AND
print(a < 100 and b > 15)

# OR
print(a < 100 or b > 100)

# NOT
print(not(a < 100 and b > 15))
```

5.4. Identity Operators

Identity operators are used to compare the memory addresses of two operands. They return a boolean value (**True** or **False**) depending on whether the memory addresses are equal or not. The following identity operators are supported in Python:

- **is** - is equal to
- **is not** - is not equal to

```
# Example: Identity Operators

# Initialize variables
a, b, c = 10, 20, 10

# is
print(a is b)
print(a is c)

# is not
print(a is not b)
```

5.5. Membership Operators

Membership operators are used to check whether a value is a member of a collection. They return a boolean value (**True** or **False**) depending on whether the value is a member of the collection or not. The following membership operators are supported in Python:

- **in** - is a member of
- **not in** - is not a member of

```
# Example: Membership Operators

# Initialize variables
a, b = 1, 5
test_list = [1, 2, 3]

# in
print(a in test_list)

# not in
print(b not in test_list)
```

5.6. Bitwise Operators

Bitwise operators are used to perform bitwise operations on the binary representations of integer operands. They return an integer value depending on the result of the bitwise operation. The following bitwise operators are supported in Python:

- **&** - bitwise AND
- **|** - bitwise OR
- **^** - bitwise XOR
- **~** - bitwise NOT
- **<<** - bitwise left shift
- **>>** - bitwise right shift

```
# Example: Bitwise Operators

# Initialize variables
a, b = 0xAA, 0x55

# Bitwise AND
print(hex(a & b))
# 0x00

# Bitwise OR
print(hex(a | b))
```

```
# 0xFF

# Bitwise XOR
print(hex(a ^ b))
# 0xFF

# Bitwise NOT
print(hex(~a))
# -0x55

# Bitwise Left Shift
print(hex(a << 1))
# 0x154

# Bitwise Right Shift
print(hex(a >> 1))
# 0x55
```

5.7. Assignment Operators

Assignment operators are used in programming languages, including Python, to assign values to variables. They combine the assignment operation with an arithmetic or logical operation, making it possible to perform an operation and assign the result to a variable in a single statement. The following assignment operators are supported in Python:

- `=` - assign
- `+=` - add and assign
- `-=` - subtract and assign
- `*=` - multiply and assign
- `/=` - divide and assign
- `//=` - floor divide and assign
- `%=` - modulus and assign
- `**=` - exponentiate and assign
- `&=` - bitwise AND and assign
- `|=` - bitwise OR and assign
- `^=` - bitwise XOR and assign
- `>>=` - bitwise right shift and assign
- `<<=` - bitwise left shift and assign

```
# Example: Assignment Operators

# Initialize variables
a = 10
```

```
b = 20

# Assign
test = a
print(test)

# Bitwise AND and Assign
test &= b
print(test)

# Bitwise OR and Assign
test |= b
print(test)

# Bitwise XOR and Assign
test ^= b
print(test)

# Left Shift and Assign
test <<= b
print(test)

# Right Shift and Assign
test >>= b
print(test)

# Add and Assign
test += b
print(test)

# Subtract and Assign
test -= b
print(test)

# Multiply and Assign
test *= b
print(test)

# Divide and Assign
test /= b
print(test)

# Floor Divide and Assign
test //= b
print(test)

# Modulus and Assign
test %= b
print(test)

# Exponentiation and Assign
```

```
test **= b
print(test)
```

5.8. Conditional Operators

Conditional operators or also ternary operators are used to evaluate a condition and return one of two values depending on whether the condition is true or false. The following conditional operators are supported in Python:


- **a if condition else b** - if condition is true, return a, otherwise return b

```
# Find the minimum of two numbers
a, b = 10, 20
result = a if a < b else b
print(result)

# Return None if a or b is None (or both) else return the highest value
a, b = (2, 1)
result= None if (a or b) is None else (a or b)
print(result)
```

5.9. Operator Precedence

Expressions in Python can make use of multiple operators. The order in which the separate operations are executed is defined by the level of precedence for the operator. Operators with higher level are executed first. If operators have the same level of precedence the associated operations are executed from left to right as they appear in the expression.

Operator	Description	Precedence
()	Parentheses	High
**	Exponentiation	
~x +x -x	Complement, Unary plus and minus	
* / // %	Multiplication, Division, Floor division, Modulus	
+ -	Addition, Subtraction	
>> <<	Right and left bitwise shift	
&	Bitwise AND	
^	Bitwise XOR	
	Bitwise OR	
in not in is is not < <= > >= != ==	Membership, Identity Comparison	
not	Boolean NOT	
and	Boolean AND	
or	Boolean OR	
		Low

```
# Example with mathematical operations
a = 2 + 3 - 4/5

print(a)
```


2 + 3 - 4/5

└───┘ ... Division has higher precedence than subtraction

2 + 3 - 0.8

└───┘ Left to right rule applies as addition and subtraction have the same precedence

5 - 0.8

└───┘

4.2

```
# Example with boolean operators
a = not False and True or False
# 1      True and True or False
# 2              True or False
# 3              True
print(a)
```

The order of precedence can be manipulated using `()`.

```
# Regular precedence
a = 2 + 2**3

# Precedence influenced by parentheses
b = (2 + 2)**3

print(a)
print(b)
```

Chapter 6. User Input

The `input()` function is used to get user input from the keyboard. It takes a single argument, which is the prompt to display to the user. The prompt is optional. If no prompt is specified, the user will not see anything when they are prompted to enter input.

6.1. User Input in Python 2

In Python 2, the `input()` function evaluates the user input as a Python expression and returns the result. This is a security risk because it allows the user to execute arbitrary Python code. To avoid this, use the `raw_input()` function instead. The `raw_input()` function returns the user input as a string.

```
# Use raw_input() to get input from the user
prompt = raw_input()
print(prompt)

# Use input() to get input from the user and evaluate it as a Python expression
prompt = input()
print(prompt)
```

6.2. User Input in Python 3

In Python 3, the `input()` function returns the user input as a string. To evaluate the user input as a Python expression, use the `eval()` function. The `eval()` function takes a string as an argument and evaluates it as a Python expression. It returns the result of the expression.

```
# Use input() to get input from the user
prompt = input()
print(prompt)

# Use eval() to evaluate the input as a Python expression
print(eval(prompt))
```

6.3. Writing Compatible Code

To write code that is compatible with both Python 2 and Python 3, use the `six` module. The `six` module provides a function called `input()` that works the same way in both Python 2 and Python 3. The `six` module is not part of the Python standard library. It must be installed separately. To install the `six` module, run the following command:

```
pip install six
```

An example of using the `six` module to get user input is shown below.

```
# Code that works in both Python 2 and 3
from six.moves import input
prompt = input()
prompt = prompt.encode('utf-8')
print(prompt)
```

Chapter 7. Control Flow

Control flow statements are used to control the order in which statements are executed in a program. They are used to make decisions and repeat statements. Python supports the following types of control flow statements:

- `if` statement
- `for` statement
- `while` statement
- `break` statement
- `continue` statement
- `pass` statement

7.1. `if` Statement

The `if` statement is used to make decisions in a program. It evaluates a condition and executes a block of code if the condition is true. The `if` statement can also be combined with an `elif` statement to execute a block of code if the condition is false and another condition is true. The `elif` statement can be used multiple times to check multiple conditions. The `if` statement can also be combined with an `else` statement to execute a block of code if all conditions are false.

```
# See chapter 6 for more information about the input() function
var = int(input('Enter value: '))

# Variable is between 1 and 10 (inclusive)
if 1 <= var <= 10:
    print("Variable is in the positive allowed range")

# Variable is between -10 and -1 (inclusive)
elif -10 <= var <= -1:
    print("Variable is in the negative allowed range")

# Variable is outside the allowed range
else:
    print("Condition outside of the allowed range")
```

7.2. `for` Statement

The `for` statement is used to iterate over the items in a collection. It can also be used to iterate over a range of numbers. The `for` statement can be combined with an `else` statement to execute a block of code when the loop is finished without breaking out of the loop.

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
```

```
print(fruit)
```

7.3. while Statement

The **while** statement is used to execute a block of code while a condition is true. The **while** statement can be combined with an **else** statement to execute a block of code when the loop is finished without breaking out of the loop.

```
i = 0
while i <= 10:
    print(i)
    i += 1
```

7.4. break Statement

The **break** statement is used inside a loop to stop the loop from executing any further. The **break** statement can be used inside a **for** or **while** loop.

```
# Example 1: Break statement inside for loop
val = int(input("Enter a number: "))
for i in range(1, 10):
    if val == i:
        print("Number found!")
        break
else:
    print("Number not found!")

# Example 2: Break statement inside while loop
val = int(input("Enter a number: "))
i = 0
while i < 10:
    if val == i:
        print("Number found!")
        break
    i += 1
else:
    print("Number not found!")
```

7.5. continue Statement

Use the **continue** statement inside a loop to skip the rest of the statements in the loop and proceed with the next iteration of the loop. The **continue** statement can be used inside a **for** or **while** loop.

```
numbers = [1, 2, 3, 4, 5, 6]
```

```
# Print all odd numbers in the list until 5
for num in numbers:
    if num % 2 == 0:
        continue
    elif num == 5:
        break
    else:
        print(num)
```

7.6. pass Statement

The **pass** statement is used as a placeholder for code that has not been implemented yet. It is used to prevent a syntax error when a statement is required syntactically, but you do not want to execute any code.

```
val = input("Enter value: ")

# Example 1: Use to avoid getting an error when empty code is not allowed or not
desired.
if val == "A":
    pass
else:
    print(val)

# Example 2" Use as a placeholder for code that is not yet implemented.
if val == "A":
    pass
elif val == "B":
    pass
else:
    pass
```

Chapter 8. Functions

Functions are used to group statements together and give them a name. They are used to make code more readable and reusable. Functions are defined using the `def` keyword. The `def` keyword is followed by the name of the function, a list of parameters in parentheses, and a colon. The statements that make up the body of the function are indented.

```
def function_name(parameter1, parameter2):  
    """ Docstring: description of the function """  
  
    # Code to be executed when the function is called  
    result = parameter1 + parameter2  
    print(result)  
  
    # Return statement (optional)  
    return result
```

Let's break down the different components of a function:

- **Function name:** This is the name you give to your function. It should be descriptive and follow the Python naming conventions (lowercase with words separated by underscores).
- **Parameters:** These are optional input variables that you can define in the function signature. They act as placeholders for values that will be passed into the function when it is called. You can have multiple parameters separated by commas.
- **Docstring:** This is an optional string enclosed in triple quotes immediately after the function definition. It provides a description of the function's purpose, parameters, and return value. It is good practice to include docstrings to document your functions.
- **Function body:** This is the block of code that gets executed when the function is called. It can contain any valid Python code, such as variable assignments, conditional statements, loops, and other function calls.
- **Return statement:** This is an optional statement that specifies the value or values that the function should return when it finishes executing. If no return statement is provided, the function returns `None` by default. You can return a single value or multiple values as a tuple.

8.1. Positional Arguments

Positional arguments are arguments that are passed to a function by position. The first argument is assigned to the first parameter, the second argument is assigned to the second parameter, and so on. Positional arguments are the default type of arguments in Python.

```
def greet(name, age):  
    print("Hello, {0}! You are {1} years old.".format(name, age))  
  
# Calling the greet() function with positional arguments
```

```
greet("Alice", 25)
```

8.2. Named Arguments

Named or also keyword arguments are arguments that are passed to a function by name. They are used to specify which parameter each argument should be assigned to. Keyword arguments are optional and can be specified in any order. They are useful when a function has many parameters and you want to specify only a few of them.

```
def greet(name, age):  
    print("Hello, {0}! You are {1} years old.".format(name, age))  
  
# Calling the greet() function with named arguments  
greet(name="Alice", age=25)
```

8.3. Default Arguments

Default arguments are arguments that have a default value specified in the function signature. They are used when a function has optional parameters. If a default argument is not specified when the function is called, the default value is used instead. Default arguments must be specified after all the required arguments.

```
def greet(name='Nemo', age=42):  
    print("Hello, {0}! You are {1} years old.".format(name, age))  
  
# Call greet() with the default arguments  
greet()
```

8.4. Variable Arguments

Variable number of arguments are arguments that can take a variable number of values. They are used when you do not know how many arguments will be passed to a function. Variable number of arguments are specified using the `*` operator in the function signature. The arguments are collected into a tuple. You can also use the `**` operator to collect keyword arguments into a dictionary.

Python 2

```
def variable_number_of_arguments(a, b, *args, **kwargs):  
    print("a: {a}".format(a=a))  
    print("b: {b}".format(b=b))  
    print("args: {args}".format(args=args))  
    print("kwargs: {kwargs}".format(kwargs=kwargs))
```



```
variable_number_of_arguments(1, 2, 3, c=4)
```

Python 3

```
def variable_number_of_arguments(a, *args, b=1, **kwargs):  
    print(f"a: {a}")  
    print(f"b: {b}")  
    print(f"args: {args}")  
    print(f"kwargs: {kwargs}")
```

```
variable_number_of_arguments(1, 2, 3, c=4)
```

8.5. Unpacking Arguments

Arguments that are unpacked from a list or tuple are used when you want to pass the elements of a list or tuple as arguments to a function. Unpacking arguments is done using the `*` operator in the function call. The elements of the list or tuple are assigned to the parameters in the function signature.

```
def my_function(a, b, c):  
    print(a, b, c)
```

```
args = [1, 2, 3]  
my_function(*args)
```

8.6. Variable Scope

The scope of a variable is the region of the program where the variable can be accessed. Python has two types of variables: global variables and local variables.

Variables defined inside a function are local to that function. They can only be accessed inside the function. Variables defined outside a function are global to the entire program. They can be accessed anywhere in the program.

Local variables can have the same name as global variables. However, they are completely separate variables and are not related in any way. Local variables take precedence over global variables inside a function.

```
# Initialize a variable  
var = 1  
print(var)
```

```
# Define a function that defines a local variable with the same name as the global
variable
def myfunction():
    var = 2
    print(var)

# Call the function
myfunction()

# Check that the function did not change the value of the global variable
print(var)
```

8.7. Nested Functions

Nested functions are functions that are defined inside another function. They can access variables from the enclosing function and are used to implement helper functions that are used mainly inside the enclosing function.

```
def absolute_value(x):
    # Emulate the built-in abs() function, e.g. abs(-1) == 1 and abs(1) == 1

    def negative_value():
        # An inner function can access the variables of the outer function

        return -x

    def positive_value():
        # An inner function can also access the variables of the outer function

        return x

    # Use the inner functions to return the correct value
    return negative_value() if x < 0 else positive_value()

print(absolute_value(-1)) # 1
print(absolute_value(1))  # 1
```

8.8. Closures

A closure is a function object that remembers the values of the variables that were present in the enclosing function when it was defined. The closure can then access and modify the values of those variables. Closures are used to implement data hiding and encapsulation.

```
def greet(message):
    # Enclosing function
```

```

def inner_function(name):
    # The message variable is stored in the inner function context
    return "{} {}".format(message, name)

# Return a closure (a function object)
return inner_function

# Store the closure function object into a variable
welcome = greet("Welcome")

# Use the concrete closure function object
print(welcome('Branko')) # Welcome Branko

# Now call the inner function directly
print(greet("Hello")('John')) # Hello John

```

8.9. Decorator Functions

Function decorators are functions that take a function as an argument and return a modified version of the function. They are used to modify the behavior of a function without changing the function itself.

Function decorators are specified using the `@` operator before the function definition. The function decorator is called when the function is defined.

Example: Simple decorator function

```

def decorator(func):
    def wrapper():
        print('Before function call')
        func()
        print('After function call')
    return wrapper

def welcome():
    print('Welcome to Python')

@decorator
def hello():
    print('Hello world')

# Use the decorator function explicitly
test = decorator(welcome)
test()

```

```
# Let Python do the work for us
hello()
```

Example: Decorator function with arguments

```
def log(message):
    # This is the outer function with the decorator parameters

    # Place a breakpoint here
    print("> Call to log")

    def decorator(func):
        # The standard decorator function with the function as parameter

        # Place a breakpoint here
        print("> Call to decorator")

        def wrapper(*args, **kwargs):
            # This is the wrapper function

            # Place a breakpoint here
            print("> First call to wrapper")

            # This is the wrapper function
            print(message)

            # Return the function
            return func(*args, **kwargs)

        # Return the wrapper function
        return wrapper

    return decorator

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

@log(message="Hello World")
def sub(a, b):
    return a - b

# Decorate function without the "@" operator
add = log("Hello World")(add)
print(sub(1, 2))

# Let Python do the work for us
```

```
print(sub(1, 2))
```

Example: Decorator function used on classes

```
# Example: Function as a decorator for a class

def counter(start_value=1):
    # This is the decorator function with the required interface

    def decorator(cls):
        # This is the decorator function with the class as parameter

        # Add the attribute to the class
        setattr(cls, 'counter', start_value)

        # Return the class
        return cls

    return decorator

# Apply the class decorator with attribute
@counter(start_value=1)
class BaseClass(object):
    pass

# Use the explicit syntax
DecoratedClass = counter(start_value=1)(BaseClass)
obj = DecoratedClass()
print(obj.counter) # Output: 1

# Let Python do the work for us
decorated_class = BaseClass()
print(decorated_class.counter) # Output: 1
```

8.10. Factory Functions

The function factory pattern is a special function used to create other functions dynamically. Typically, a factory function takes a set of parameters and returns a function that uses those parameters. The returned function can then be called with different arguments to get different results.

```
def power_of(n):
    def power(x):
        return x ** n
    return power
```

```
# Square root
sqrt = power_of(0.5)
print(sqrt(100.0))

# Square
sqr = power_of(2)
print(sqr(10.0))
```

8.11. Memoization

Memoization is a technique used to optimize function calls by caching the results of expensive computations and reusing them when the same inputs occur again. Closures can be used to implement memoization by creating a wrapper function that encapsulates the memoization logic. It is important to note that memoization can lead to memory leaks if not implemented correctly. You must ensure that the cache is cleared when it is no longer needed.

```
def fibonacci(n):
    cache = {}

    def memoized_fibonacci(n):
        if n in cache:
            return cache[n]
        if n <= 1:
            result = n
        else:
            result = memoized_fibonacci(n-1) + memoized_fibonacci(n-2)
        cache[n] = result
        return result

    return memoized_fibonacci(n)

print(fibonacci(5))
print(fibonacci(10))
```

8.12. Function Annotations

For Python 3.0 and above, you can specify the types of the parameters and return value of a function using annotations. Function annotations are optional type hints that can be added to function parameters and the return value.

- Use the `:` operator to specify the type of a parameter
- Use the `->` operator to specify the type of the return value

```
# Function annotations for parameters and return value
```

```
def add(x: int, y: int) -> int:
    """ Add two numbers.

    Args:
        x (int): First number.
        y (int): Second number.

    Returns:
        int: Sum of the two numbers.

    """
    return x + y

# Function annotations for parameters and return value
def sub(x: int, y: int) -> int:
    """ Subtract two numbers.

    Args:
        x (int): First number.
        y (int): Second number.

    Returns:
        int: Difference between the two numbers.

    """
    return x - y
```

8.13. Function Attributes

In Python everything is an object, including functions. This means that functions can have attributes just like any other object. Function attributes are used to store information about a function. Some typical use cases for function attributes are:

- Storing metadata about a function (e.g name, author, version, etc.)
- Conditional execution of a function (e.g. `test` attribute in the `unittest` module)
- Storing cached results of a function (e.g. `cache` attribute in the `functools` module)
- Multiple forms of documentation with a method optimized for various platforms

Typically the attributes will be added by a decorator. However, you can also add them manually using the `setattr()` function or directly on the function object.

```
def foo():
    pass

foo.name = "MyFunc"
foo.description = "This is my function"
```

```
foo.author = "Me"

print(foo.author)
print(foo.name)
print(foo.description)
```

CAUTION

It is not recommended to add attributes to functions manually because it can lead to unexpected behavior. It is better to use a decorator to add the attributes or use a class instead.

8.14. Callback Functions

Callback functions are functions that are passed as arguments to other functions. They are used to implement event handlers and asynchronous callbacks. Callback functions are specified using the function name without the parentheses. The function is called when the callback is executed.

```
# Example: Callback function

def button_action(x):
    print("Callback function called with {}".format(x))

def on_pressed(callback):
    print("Test function called")
    callback("ON")

def off_pressed(callback):
    print("Test function called")
    callback("OFF")

on_pressed(callback=button_action)
off_pressed(callback=button_action)
```

8.15. Recursive Functions

Recursion is the process of calling a function from within itself. It is used to solve problems that can be broken down into smaller problems of the same type. It is important to note that recursion can lead to infinite loops if not implemented correctly. You must ensure that the base case is reached at some point. The base case is the case where the function does not call itself. It is used to stop the recursion. The base case is usually the simplest case that can be solved without recursion. The recursive case is the case where the function calls itself.

```
def factorial(n):
```



```

# Base case
if n == 0:
    return 1

# Recursive case
else:
    return n * factorial(n - 1)

test_function = factorial(5)
print(test_function)

```

8.16. Lambda Functions

Lambda functions are anonymous functions that are defined using the `lambda` keyword. They are used when you need a temporary function during the program execution. Lambda functions can only contain a single expression. The expression is evaluated and the result is returned when the function is called. Lambda functions can be assigned to a variable or passed as an argument to another function.

```

# Example : Lambda functions

"""
lambda [param1, param2, ..]: expression

Lambda functions are one-line functions which return an expression using
the pre-defined parameters param1, param2, ... paramN.

Lambda functions are normally used for quick operations on data,
most notably in combination with map, filter, reduce.

"""

# Define a list to iterate over
data_in = [1, 2, 3]

# Use a lambda function to square the input and then map the result to a list
`data_out`
data_out = list(map(lambda x: x * x, data_in))

# Print the result
print(data_in, data_out)

```

```

nop = lambda: None
print(nop())

```

```
# Example : Lambda functions with multiple arguments
```

```
# Multi-parameter lambda
x = lambda a, b, c, d, e: (a + b) * (c + d) / e
print(x(1, 2, 3, 4, 5))
```

```
# Example : Demonstrate that both lamda and def return the same result when compiled
import dis
```

```
func1 = lambda x: x * x
```

```
def func2(x):
    return x * x
```

```
print(dis.dis(func1))
print(dis.dis(func2))
```

```
# Conditional lambda
y = lambda b: 1 if b > 0 else 0
print(y(-1), y(0), y(1))
```

```
# Example: Lambda function with nested conditional lambda
```

```
# The following example demonstrates a lambda function with nested conditional lambda.
z = lambda c: 1 if c > 10 else (1 if c < -10 else 0)
print(-11, -10, -1, 0, 1, 10, 11, sep='\t')
print(z(-11), z(-10), z(-1), z(0), z(1), z(10), z(11), sep='\t')
```

```
# Example: Lambda function with recursion
```

```
x = lambda a: a * x(a - 1) if a > 1 else 1
print(x(5))
```

8.17. Positional-Only Arguments

Positional-only arguments are arguments that can only be passed by position and not by name. They are used when you want to prevent users from passing arguments by name.

- Positional-only arguments are specified using the `/` operator in the function signature.
- The arguments before the `/` operator are positional-only arguments.
- The arguments after the `/` operator can be passed by position or by name.

- This feature was introduced in Python 3.8.

```
# Example: Positional-only arguments

# The arguments a and b are positional-only
def positional_only_arguments(a, b, /):
    return a + b

# The argument a is positional-only, b is positional or keyword
def one_positional_only_argument(a, /, b):
    return a + b
```

8.18. Keyword-Only Arguments

Keyword-only arguments are arguments that can only be passed by name and not by position. They are used when you want to prevent users from passing arguments by position.

- Keyword-only arguments are specified using the `*` operator in the function signature.
- The arguments after the `*` operator are keyword-only arguments.
- This feature was introduced in Python 3.0.

```
# Example: Keyword-only arguments

# The arguments a and b are keyword-only
def keyword_only_arguments(*, a, b):
    return a + b

# The argument a is positional or keyword, b is keyword-only
def one_keyword_only_argument(a, *, b):
    return a + b

# The argument a is positional only, b is keyword-only
def separate_arguments(a, /, *, b):
    return a + b
```

8.19. Function Introspection

Function introspection is the process of examining a function's attributes. It is used to get information about the state variables and code of a function after it is created and initialized. In Python a function is a first class object, which means that it can be handled and manipulated like any other object.

- use `__name__` to get the name of a function

- use `__doc__` to get the docstring of a function
- use `__defaults__` to get the default values of a function's positional parameters
- use `__kwdefaults__` to get the default values of a function's keyword parameters
- use `__globals__` to get the global variables outside the function
- use `__closure__` to get the parent objects of a nested function
- use `__dict__` to get the attributes and their values of a function
- use `__code__` to get the code object of a function

```
def foo(a, b=10, c=20, *args, **kwargs):
    """This is 'foo' function that does nothing."""

    pi = 3.14

    def bar(a=1, b=2, c=3, *args, **kwargs):
        """This is 'bar' function that does nothing."""

        print(pi)

        introspections = {
            '__globals__': bar.__globals__,
            '__name__': bar.__name__,
            '__doc__': bar.__doc__,
            '__code__': bar.__code__,
            '__defaults__': bar.__defaults__,
            '__closure__': bar.__closure__,
            '__dict__': bar.__dict__,
        }

        # A function has a name
        test = introspections['__name__']
        print("Getting function attribute __name__ -> {} ".format(test))

        # A function has a docstring
        test = introspections['__doc__']
        print("Getting function attribute __doc__ -> {} ".format(test))

        # A function has default arguments
        test = introspections['__defaults__']
        print("Getting function attribute __defaults__ -> {} ".format(test))

        # A function has access to the global namespace
        test = introspections['__globals__']
        print("Getting function attribute __globals__ -> {} ".format(test))

        # A function has a closure
        test = introspections['__closure__']
        print("Getting function attribute __closure__ -> {} ".format(test))
```

```
# A function has a variables dictionary
test = introspections['__dict__']
print("Getting function attribute __dict__ -> {} ".format(test))
```

```
# A function has a code object
test = introspections['__code__']
print("Getting function attribute __code__ -> {} ".format(test))
```

```
# A code object has name
print(test.co_name)
```

```
bar()
```

```
# foo(1, 2, 3, 4, 5, c=30, d=40, e=50)
foo(a=1, d=40, e=50)
```

Chapter 9. Classes

A class is a blueprint for creating objects. It defines the properties and methods that all objects of the class will have. An object is an instance of a class. It is a concrete realization of the class. It has well-defined values or all the properties defined in the class.

The class answers the following questions:

- What parts does the object HAVE? (HAVE WHAT?)
- What does the object DO? (DO WHAT?)

The object answers the following questions:

- What is the object? (WHAT IS IT?)
- What is the object's state? (WHAT IS IT LIKE?)

Let's take a look at an example of a class and an object. The class is called `Person` and it defines the properties and methods that all people have. The class answers the question "What does a person have?" and in this case the answer is "A person has a name and an age". The class also answers the question "What does a person do?" and in this case the answer is "A person can say hello".

```
# Example: Class as template

class Person(object):

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_hello(self):
        print("Hello, my name is {} and my age is {}".format(self.name, self.age))
```

The object is called `John` and it is an instance of the `Person` class. It answers the question "What is John?" and in this case the answer is "John is a person". It also answers the question "What is John's state?" and in this case the answer is "John is 30 years old and his name is John".

```
# Example: Object as concrete realization of a class
from class_as_blueprint import Person

John = Person("John", 32)
John.say_hello()
```

9.1. Class structure

The class is defined using the `class` keyword and the name of the class. The class name should

follow the Python naming conventions (CamelCase with the first letter capitalized). It shall always inherit from the `object` or a class that inherits from the `object` due to reasons explained in later chapters.

- Initializer (`__init__`)
- Constructor (`__new__`)
- Instance variables (inside `__init__`)
- Instance methods (inside class body)
- Class attributes (outside any method)
- Class methods (`@classmethod`)
- Static methods (`@staticmethod`)

```
# Example: Class structure

class ClassStructure(object):

    CLASS_VARIABLE = "Hi, I am "

    def __new__(cls, *args, **kwargs):
        print("This is the constructor method")
        return object.__new__(cls)

    def __init__(self):
        print("This is the initialization method")
        self.instance_variable = "John"

    @classmethod
    def class_method(cls):
        print("This is a class method, the class prefix is:
{}").format(cls.class_variable))

    @staticmethod
    def static_method():
        print("This is a static method, the class prefix is:
{}").format(ClassStructure.class_variable))

    def instance_method(self):
        print("This is an instance method, '{} {}'".format(self.class_variable,
self.instance_variable))
```

9.2. Initializer

The `__init__` method is called automatically when the instance is created. It is used to initialize the instance state. It is also sometimes wrongly called the constructor method as it is the most commonly used method when declaring the class. Some of the most common use cases or the `__init__` method are:

- Initialize the instance variables of the class
- Validate the values of the instance variables
- Perform any other initialization steps

Example: Initialization with the `__init__` method

```
class Person(object):

    def __init__(self, name="Branimir", age=40):
        self.name = name
        self.age = age

        print("My name is {0} and I am {1} years old".format(self.name, self.age))

p1 = Person()
p2 = Person("John", 30)
```

Example: Initialization and validation

```
class Person(object):

    def __init__(self, name, age):
        self.name = name
        self.age = age

        if not isinstance(self.name, str):
            raise TypeError("Name must be a string")

        if not isinstance(self.age, int):
            raise TypeError("Age must be an integer")

        if self.age < 0:
            raise ValueError("Age must be a positive integer")

    def say_hello(self):
        print("Hello, my name is {} and my age is {}".format(self.name, self.age))

john = Person("John", 32)
john.say_hello()
```

Example: Initialization in multiple steps

```
class Person(object):
    def __init__(self):
        self.name = self.step_1()
```



```

        self.age = self.step_2()

    @staticmethod
    def step_1():
        name = input("Step 1: Enter the person's name: ")
        return name

    @staticmethod
    def step_2():
        age = input("Step 2: Enter the person's age: ")
        return age

john = Person()
print(john.name, john.age)

```

9.3. Constructor

Constructors are special methods inside a class that create and initialize the objects of a class. In Python the object is created with the special method `__new__` and then initialized with the special method `__init__`.

```

class Calculator(object):

    def __new__(cls):

        # Use the parent class to create the object
        obj = object.__new__(cls)

        # Return the object
        print("__new__ : Created object {}".format(obj))
        return obj

    def __init__(self):

        # Self is the object that was created by __new__
        print("__init__ : Using object {}".format(self))

        # Add attributes to the object
        self.name = "Cool Calculator"

calc = Calculator()
print(calc.name)

```

The method `__new__` is used rarely as Python handles the construction of the object automatically. Some of the most common use cases for the `__new__` method are:

- Create a class that inherits from a different class depending on some condition
- Limit the number of instances that can be created for this class (e.g. Singleton)
- Create a class factory

Example: Singleton using the `__new__` method

```
class Singleton(object):

    __instance = None

    def __new__(cls):
        if cls.__instance is None:
            print("Creating singleton...")
            cls.__instance = object.__new__(cls)

        else:
            print("Singleton already exists...")

        return cls.__instance

s1 = Singleton()
s2 = Singleton()
print(s1 == s2)
print(s1 is s2)
```

Example: Class factory using the `__new__` method

```
class WindowsCalculator(object):

    @staticmethod
    def do():
        print("Do in Windows Calculator")

class LinuxCalculator(object):

    @staticmethod
    def do():
        print("Do in Linux Calculator")

class Calculator(object):

    def __new__(cls, os="windows"):

        # An instance of the WindowsCalculator class is returned
        if os == "windows":
```

```

        return object.__new__(WindowsCalculator)

    # An instance of the LinuxCalculator class is returned
    elif os == "linux":
        return object.__new__(LinuxCalculator)

    # Invalid operating system
    else:
        raise ValueError("Invalid operating system")

# Create a new instance of the Calculator class
calc = Calculator("windows")

# Check if the Calculator class is an instance of the WindowsCalculator class
test = isinstance(calc, WindowsCalculator)
print("Is instance of WindowsCalculator? [{}].format(test))

# Call the do method
calc.do()
```

```

# Example: Conditional inheritance using the __new__ method

class WindowsCalculator(object):
    """ Windows calculator class operations """

    @staticmethod
    def do():
        print("Do in Windows Calculator")

class LinuxCalculator(object):
    """ Linux calculator class operations """

    @staticmethod
    def do():
        print("Do in Linux Calculator")

class Calculator(object):

    def __new__(cls, os="windows"):

        # Windows base class
        if os == "windows":
            parents = (WindowsCalculator, )

        # Linux base class
        elif os == "linux":
            parents = (LinuxCalculator, )
```

```

# Invalid operating system
else:
    raise ValueError("Invalid operating system")

# Create a new class with the given name and bases
cls = type("Calculator", parents, {})

# Return an instance of the new class
return cls()

# Create a new instance of the Calculator class
calc = Calculator("windows")

# Check if the Calculator class is a subclass of the WindowsCalculator class
test = isinstance(calc, WindowsCalculator)
print("Is subclass of WindowsCalculator? [{}].format(test))

# Call the do method
calc.do()

```

9.4. Class instance

The constructor returns a class instance (object) as a concrete realization of the class. This means that the object has well-defined values for all the instance variables. Each instance is unique and has its own identity.

Some of the most common use cases for the class instance are:

- Unique identity of the object (e.g. id)
- Unique state of the object (e.g. name and age)
- Unique behavior of the object based on the state (e.g. say hello)

9.4.1. Instance identity

The identity is a unique number that is assigned to the object when it is created. The identity of an object never changes during its lifetime. It is used mainly to compare objects using the `is` operator. The `is` operator returns `True` if the identity of the two objects is the same and `False` otherwise. The identity of an object is obtained by the `id()` function.

```

# Example: Class instance with concrete values

class Person(object):

    def __init__(self):
        print("Person has ID {}".format(id(self)))

```

```
# The person object has a unique id
p1 = Person()
p2 = Person()
```

9.4.2. Instance variables

The instance has its own set of variables called also attributes. They represent the internal state of the object. The instance variables are defined inside `__init__` method.

```
class Person(object):

    def __init__(self):
        self.name = "Branimir"
        self.age = 40

# Create the instance
p = Person()

# Access to the instance attributes
print(p.name)
print(p.age)
```

9.4.3. Instance methods

Instance methods are functions that are part of the instance of a class. The first parameter of an instance method is always a reference to the current instance. It is usually called `self`. Instance methods can only be used when the instance of the class has been created.

```
# Example: Class instance with instance methods

class Person(object):

    def do_something(self):
        print("{} is doing something".format(self))

    def do_something_with(self, something, someone):
        print("{} is doing {} with {}".format(self, something, someone))

# Create the instance
p = Person()
p.do_something()
p.do_something_with("nothing", "no one")
```

9.5. Class as an object

In Python, everything is an object. This means that classes are also objects. They are instances of the `type` class. This means that classes can be assigned to variables, passed as arguments to functions, and returned from functions. This is a very powerful feature of Python that allows you to write very flexible code. Some of the most common use cases for classes as objects are:

- Assign a class to a variable
- Pass a class as an argument to a function
- Return a class from a function

9.5.1. Class variables

Class attributes are variables that are defined inside a class. They represent the internal state of the class. Class attributes are shared by all instances of the class. They are defined outside the `__init__` method. If you try to access a class attribute that does not exist, Python will look for it in the parent classes until it reaches the top of the inheritance tree. Some of the most common use cases for class attributes are:

- Constants
- Default values for class attributes
- Class attributes that are shared by all instances of the class

Example: Class attributes

```
class Person(object):

    # Class attributes are defined outside of any method and are shared by all
    instances
    MALE_PREFIX = "Mr."
    FEMALE_PREFIX = "Ms."

    def __init__(self, name, sex):
        self.name = name
        self.sex = sex

    def get_prefix(self):
        """ Demonstrate how to access class attributes from instance methods """

        # The person is male
        if self.sex == "male":
            prefix = self.MALE_PREFIX

        # The person is female
        elif self.sex == "female":
            prefix = self.FEMALE_PREFIX

        # Others
```

```

        else:
            prefix = ""

        return prefix

    def get_name(self):
        """ Return the name with the appropriate prefix """
        return "{} {}".format(self.get_prefix(), self.name)

# Create the instances
males = [Person(name="Branimir", sex="male"), Person("Dimitar", sex="male")]

# Class attributes are accessible from the instance methods
print("Default prefix...")
for male in males:
    print(male.get_name())

# Class attributes are accessible from the class itself and a change will affect all
instances
print("Prefix changed...")
Person.MALE_PREFIX = "Sir"
for male in males:
    print(male.get_name())

```

It is important to note that if the class attribute is used with `self` in any of the instance methods, it will be treated as an instance attribute and will be accessible only from the instance of the class. This might introduce bugs in your code if you are not careful.

```

"""
The statement self.<variable> may refer to two things at different times. When no
instance
variable exists for a name, Python will look up the variable in the class. So the
value retrieved
for self.<variable> will be the class variable.

But when setting an attribute via self, Python will always set an instance variable.
So now
self.<variable> is a new instance variable whose value is equal to the class variable
+ 1. This
attribute shadows the class attribute, which you can no longer access via self but
only via the
class.
"""

class TestOp(object):

    immutable = 1
    mutable = [1, ]

```

```

class Test(TestOp):

    def __init__(self):
        super(Test, self).__init__()

    def test_immutable(self):

        print("#" * 80)
        print("Testing immutable class variable")
        print("#" * 80)

        # self references to the class variable
        print("")
        print("Read value through class name and then self...")

        print("CLS => {0}:{1}".format(id(TestOp.immutable), TestOp.immutable))
        print("SELF => {0}:{1}".format(id(self.immutable), self.immutable))

        # self creates and references an instance variable (shadows the class
variable)
        print("")
        print("Change immutable type using self...")

        self.immutable = 2
        print("CLS => {0}:{1}".format(id(TestOp.immutable), TestOp.immutable))
        print("SELF => {0}:{1}".format(id(self.immutable), self.immutable))

        # Changing the value of immutable class variable will create a new object
        print("")
        print("Change the value of the class immutable variable...")

        TestOp.immutable = 3
        print("CLS => {0}:{1}".format(id(TestOp.immutable), TestOp.immutable))
        print("SELF => {0}:{1}".format(id(self.immutable), self.immutable))

        print("")

    def test_mutable(self):
        print("#" * 80)
        print("Testing mutable class variable")
        print("#" * 80)

        # self references to the class variable
        print("")
        print("Read value through class name and then self...")

        print("CLS => {0}:{1}".format(id(TestOp.mutable), TestOp.mutable))
        print("SELF => {0}:{1}".format(id(self.mutable), self.mutable))

```



```

        # self creates and references an instance variable (shadows the class
variable)
        print("")
        print("Change mutable type using self...")

        self.mutable.append(2)
        print("CLS => {0}:{1}".format(id(TestOp.mutable), TestOp.mutable))
        print("SELF => {0}:{1}".format(id(self.mutable), self.mutable))

        # self
        print("")
        print("Change the value of the class mutable variable...")

        TestOp.mutable.append(3)
        print("CLS => {0}:{1}".format(id(TestOp.mutable), TestOp.mutable))
        print("SELF => {0}:{1}".format(id(self.mutable), self.mutable))

        print("")

if __name__ == "__main__":

    test = Test()

    test.test_mutable()
    print("Final value of immutable class variable is
{0}:{1}\n".format(id(Test.immutable), Test.immutable))

    test.test_immutable()
    print("Final value of mutable class variable is
{0}:{1}\n".format(id(Test.mutable), Test.mutable))

```

9.5.2. Class methods

Class methods are functions that are defined inside a class. The first parameter of a class method is always a reference to the class. It is usually called `cls`. Class methods can be used before the instance of the class has been created. Class methods are defined using the `@classmethod` decorator. Some of the most common use cases for class methods are:

- Create new instances of the class
- Modify existing instances of the class
- Modify the class itself

```

# Example: Class method used to create instances

class Person(object):

    NAME_PREFIX = "Mr."

```

```
def __init__(self, name):
    self.name = name

@classmethod
def from_string(cls, name):
    return cls(name)

p = Person.from_string("John")
print(p.name)
```

Example: Class method used to modify the class itself

```
class Person(object):

    NAME_PREFIX = "Mr."

    def __init__(self, name):
        self.name = name

    @classmethod
    def set_prefix(cls, prefix):
        cls.NAME_PREFIX = prefix

p = Person("John")
Person.set_prefix("Dr.")
print(p.NAME_PREFIX)
```

Example: Class method used to modify existing instances of the class

```
class Person(object):

    NAME_PREFIX = "Mr."

    def __init__(self, name):
        self.name = name

    @classmethod
    def set_prefix(cls, prefix):
        cls.NAME_PREFIX = prefix

    @classmethod
    def add_prefix(cls, person):
        person.name = "{} {}".format(cls.NAME_PREFIX, person.name)
```

```
p = Person("John")
Person.set_prefix("Dr.")
Person.add_prefix(p)
print(p.name)
```

9.6. Class as a namespace

A namespace is a mapping from names to objects. It is used to avoid name collisions and to make it easier to find the names you are looking for. In Python, classes are namespaces. This means that you can define variables and functions inside a class. They will be accessible using the class name. Some of the most common use cases for classes as namespaces are:

- Group related variables and functions together
- Avoid name collisions
- Make it easier to find the names you are looking for

Static methods are functions that are defined inside a class. They are not bound to the class or any instances of the class and use the class name as a namespace to be accessed. They are defined using the `@staticmethod` decorator. Static methods can be used before the instance of the class has been created.

```
# Example: Static method are not bound to the class and can be used as utility
functions

class Packet(object):

    def __init__(self, ip_addr='192.168.10.1', mask="255.255.255.0", payload=()):
        self.payload = payload
        self.ip_addr = ip_addr
        self.mask = mask

    @staticmethod
    def dot_to_bytes(val):
        return bytes(map(int, val.split('.')))

    @staticmethod
    def bytes_to_dot(val):
        return '.'.join(map(str, val))

# Convert IP address in dot notation to bytes
addr_bytes = Packet.dot_to_bytes('192.168.1.1')
print(addr_bytes)

# Convert bytes to IP address in dot notation
addr_dot = Packet.bytes_to_dot(addr_bytes)
print(addr_dot)
```

9.7. Properties

The properties are special methods that are used to access and modify the instance attributes of a class. They are defined using the `@property` decorator. The properties encapsulate the internal state of the object and provide a public interface for accessing and modifying it. For example, the setter property can be used to validate the value of an instance attribute before setting it.

When to use properties:

- When you want to validate the value of an instance attribute before setting it
- When you want to compute the value of an instance attribute when it is accessed
- When you want to hide the internal implementation details of an instance attribute

```
# Example: Property used to encapsulate an instance variable
```

```
class Person(object):

    def __init__(self, name):
        self.name = name

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError("Expected a string")
        self.__name = value

p = Person("John")
print(p.name)
```

9.8. Superclass

A superclass in Python is a reference to the parent class of a class. It is used to access the attributes and methods of the parent class. The superclass is specified using the `super()`.

When to use super:

- When you want to call the methods of the parent class
- When you want to access the attributes the parent class

```
# Example: Call super class method and access super class attributes
```

```
class Person(object):
```

```

def __init__(self, name):
    self.name = name

class Employee(Person):

    def __init__(self, name, id_number):
        super(Employee, self).__init__(name)
        self.id_number = id_number

e = Employee("John", 1234)
print(e.name)
print(e.id_number)

```

```

# Example: Call __new__ method of super class

class Person(object):

    def __new__(cls, name):
        return super(Person, cls).__new__(cls)

    def __init__(self, name):
        self.name = name

class Employee(Person):

    def __new__(cls, name, id_number):
        return super(Employee, cls).__new__(cls, name)

    def __init__(self, name, id_number):
        super(Employee, self).__init__(name)
        self.id_number = id_number

e = Employee("John", 1234)
print(e.name)
print(e.id_number)

```

9.9. Class inheritance

Class inheritance is the process of creating a new class from one or more base classes. The new class is called the subclass or the derived class. The base class is called the superclass or the parent class.

The subclass inherits the attributes and methods of the superclass. It can also override the

attributes and methods of the superclass. Class inheritance is used to reuse code and to represent an *is-a* relationship. The subclass is a specialized version of the superclass.

Python supports single inheritance and multiple inheritance. Some of the most common use cases for inheritance are:

- Reuse code (a student inherits name and age from a person)
- Represent an is-a relationship (a student is a person)

```
# Example: Multilevel inheritance
```

```
class A(object):  
  
    @staticmethod  
    def process():  
        print("Root is processing... ")
```

```
class B(A):  
    pass
```

```
class C(B):  
    pass
```

```
# The method process is searched for until the first class having the method is found  
(here A)  
test = C()  
test.process()
```

```
# Example: Multiple inheritance
```

```
class A(object):  
  
    @staticmethod  
    def process():  
        print("Class A is processing... ")
```

```
class B(object):  
  
    @staticmethod  
    def process():  
        print("Class B is processing... ")
```

```
class C(A, B):  
    pass
```

```

class D(B, A):
    pass

# The method process is searched for until the first class having the method is found
(here A)
test = C()
test.process()

# The method process is searched for until the first class having the method is found
(here B)
test = D()
test.process()

```

9.10. Method resolution order

Method Resolution Order (MRO) is a critical concept in Python's object-oriented programming, especially when dealing with inheritance and classes. It defines the order in which methods are searched and executed in a class hierarchy. The MRO ensures that when you call a method on an object, Python knows which method to execute.

The algorithm used to determine the MRO of a class is called the C3 linearization algorithm. It searches the class hierarchy depth first and for each depth level it searches from left to right. The MRO of a class can be accessed using the `mro()` method.

The MRO algorithm is the unique algorithm that achieves several desirable properties:

1. Each ancestor class appears exactly once
2. A derived class always appears before its ancestor ("monotonicity")
3. Direct parents of the same class should appear in the same order as they are listed in class definition ("consistent local precedence order")
4. If children of class A always appear before children of class B, then A should appear before B ("consistent extended precedence order")

For more information about the C3 linearization algorithm, see the following resources:

- https://en.wikipedia.org/wiki/C3_linearization
- <https://www.python.org/download/releases/2.3/mro/>

```

# Example: MRO (Method Resolution Order) - Bottom first

class A(object):

    @staticmethod
    def process():

```

```

        print('A.process()')

class B(object):
    @staticmethod
    def process():
        print('B.process()')

class C(A, B):

    @staticmethod
    def process():
        print('C.process()')

obj = C()
obj.process()
print(C.mro())

```

Example: MRO (Method Resolution Order) - Left first

```

class A(object):

    @staticmethod
    def process():
        print('A.process()')

class B(object):
    @staticmethod
    def process():
        print('B.process()')

class C(A, B):
    pass

class D(B, A):
    pass

```

The method process is searched for until the first class having the method is found (here A)

```

test = C()
print(C.mro())
test.process()

```

The method process is searched for until the first class having the method is found


```
(here B)
test = D()
print(D.mro())
test.process()
```

Example: MRO (Method Resolution Order) - Combined bottom first and left first

```
class A(object):
    @staticmethod
    def process():
        print("A.process()")

class B(object):
    @staticmethod
    def process():
        print("B.process()")

class C(A, B):
    pass

class D(C, B):
    pass

d = D()
print(D.mro())
d.process()
```

Example: MRO (Method Resolution Order) - Diamond problem

```
class A(object):
    @staticmethod
    def process():
        print("A.process()")

class B(A):
    @staticmethod
    def process():
        print("B.process()")

class C(A):
    @staticmethod
    def process():
```

```

        print("C.process()")

class D(B, C):
    pass

d = D()
print(D.mro())
d.process()

```

```

# Example: MRO (Method Resolution Order) - Unresolved

class Player(object):
    pass

class Enemy(Player):
    pass

class GameObject(Player, Enemy):
    pass

# Explanation (see MRO rules in the documentation):
#
# - MRO is GameObject -> Player -> Enemy -> Player (not monotonic as Player appears
# twice)
# - Rule 2 says Enemy should appear before Player
# - Rule 3 says Player should appear before Enemy
#
# Rules 2 and 3 are in conflict, so the MRO algorithm cannot be applied. This is
# called an
# "unresolvable inheritance graph" and Python will raise an exception in this case.

g = GameObject()
print(GameObject.mro())

```

9.11. Abstract class

In Python, an abstract class is a class that cannot be instantiated directly. It is meant to serve as a blueprint for other classes and is designed to be subclassed. Abstract classes often contain abstract methods, which are methods declared but not implemented in the abstract class. Subclasses of the abstract class are then required to provide concrete implementations for these abstract methods.

It is also possible for an abstract class to have concrete methods that are implemented in the abstract class itself. This is useful when you want to provide a default implementation that can be

overridden by subclasses. In this case, the abstract class can be instantiated directly and the class is also called a **template class**.

Python does not have a built-in abstract class keyword. However, you can create an abstract class using the `abc` module. The `abc` module provides a metaclass called `ABCMeta` that allows you to create abstract classes. An abstract class is created by inheriting from the `ABCMeta` metaclass and specifying one or more abstract methods using the `@abstractmethod` decorator.

The `@abstractmethod` decorator can be stacked with other decorators, such as `@classmethod` and `@staticmethod`, on the abstract method.

In case you have to write code that is compatible with both Python 2 and Python 3, you can use the `six` module. The `six` module provides all the functionality of the `abc` module in a way that is compatible with both Python 2 and Python 3. The `six` module is not part of the Python standard library. It must be installed separately. To install the `six` module, run the following command:

```
pip install six
```

Some examples of using the `abc` module to create abstract classes are shown below:

```
# Example: Abstract class using the six library for Python 2
```

```
from six import with_metaclass
from abc import ABCMeta, abstractmethod
```

```
class CalculatorAbc(with_metaclass(ABCMeta)):
```

```
    def __init__(self, mode="basic"):
        self.mode = mode
```

```
    @abstractmethod
    def add(self, *args, **kwargs):
        raise NotImplementedError
```

```
    @abstractmethod
    def subtract(self, *args, **kwargs):
        raise NotImplementedError
```

```
class Calculator(CalculatorAbc):
```

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

```
    def subtract(self, a, b):
        return a - b
```

```
calc = Calculator()
print(calc.add(1, 2))
print(calc.mode)
```

Example: Abstract class in Python 3+

```
from abc import ABC, abstractmethod
```

```
class CalculatorAbc(ABC):
```

```
    def __init__(self, mode="basic"):
        self.mode = mode
```

```
    @abstractmethod
    def add(self, *args, **kwargs):
        raise NotImplementedError
```

```
    @abstractmethod
    def subtract(self, *args, **kwargs):
        raise NotImplementedError
```

```
class Calculator(CalculatorAbc):
```

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

```
    def subtract(self, a, b):
        return a - b
```

```
calc = Calculator()
print(calc.add(1, 2))
print(calc.mode)
```

Example: Stacking decorators

```
from abc import ABCMeta, abstractmethod
from six import with_metaclass
```

```
class DeviceAbc(with_metaclass(ABCMeta)):
    """Example abstract class
```

```
    Usage:
```

```
        # Optional
```

```
@property, @staticmethod, @classmethod
```

```
+
```

```
# Obligatory decorator  
@abstractmethod
```

Example:

```
# Defines and abstract property  
@property  
@abstractmethod  
def prop(self):  
    ...
```

```
"""
```

```
def __init__(self):  
    self._bar = "bar"
```

```
@property  
@abstractmethod  
def bar(self):  
    pass
```

```
@abstractmethod  
def foo(self):  
    pass
```

```
class Samsung(DeviceAbc):
```

```
    @property  
    def bar(self):  
        return self._bar
```

```
    def foo(self):  
        print('foo')
```

```
test = Samsung()  
print(test.bar)  
test.foo()
```

9.12. Interface class

An interface is a class that contains only abstract methods. It is used to define the interface of a class without providing an implementation. An interface is a contract that a class must implement. Python does not offer direct support for interfaces. However, you can create an abstract class that

emulates an interface (see the previous section for more information).

9.13. Mixin class

A mixin is a class that contains methods that can be used by other classes without having to be the parent class of those classes. It is used to provide a common set of methods to multiple classes. Mixins are not meant to be instantiated directly. They are meant to be inherited by other classes. Mixins are usually named with the suffix "Mixin" to indicate that they are not meant to be instantiated directly. Some examples of mixins are shown below.

```
# Example: Mixin class

class RemoteMixin(object):

    def __init__(self, brand=None, volume=0, *args, **kwargs):

        # This syntax is required in order to guarantee that the MRO is not broken
        super(RemoteMixin, self).__init__(*args, **kwargs)

        # Mixin specific attributes
        self.brand = brand
        self.volume = volume

    def volume_up(self):
        self.volume += 1

    def volume_down(self):
        self.volume -= 1

    def status(self):
        print("Brand: {}".format(self.brand))
        print("Volume: {}".format(self.volume))

class JvcRemote(RemoteMixin, object):
    """ Mixins should be always inherited first """

    def __init__(self):
        super(JvcRemote, self).__init__(brand="JVC", volume=10)

    def status(self):
        super(JvcRemote, self).status()

    @staticmethod
    def learn():
        print("Learn button")

class SonyRemote(RemoteMixin, object):
```

```

""" Mixins should be always inherited first """

def __init__(self):
    super(SonyRemote, self).__init__(brand="Sony", volume=5)

    @staticmethod
    def home():
        print("Home button")

remote = JvcRemote()
actions = ["volume_up", "status", "volume_down", "status", "learn", "status"]
for action in actions:
    print("Action: {}".format(action))
    func = getattr(remote, action)
    func()

print("\n")

remote = SonyRemote()
actions = ["volume_up", "status", "volume_down", "status", "home", "status"]
for action in actions:
    print("Action: {}".format(action))
    func = getattr(remote, action)
    func()

```

9.14. Class as a decorator

A class can be used to decorate a function or a class. It is used to add functionality to an existing function or class without modifying the function or class itself. A class decorator is specified using the `@` operator before the function or class definition.

```

# A class decorator for functions
@classDecorator(arg1=..., arg2=...)
def function():
    pass

# A class decorator for classes
@classDecorator(arg1=..., arg2=...)
class MyClass:
    pass

```

Example: Class decorator for functions

```

# Example: Class as a decorator for functions and methods

class Counter(object):

```

```

def __init__(self, init_value=0):
    """ Initialize counter."""
    self._counter = init_value

def __call__(self, function):
    """ Wrapping call to original function. """

    def wrapper(*args, **kwargs):
        """ Wrapper function."""
        try:
            self._counter += 1
            print("{}".format(self._counter))
            return function(*args, **kwargs)

        except Exception as e:
            print(e)

    return wrapper

def f():
    print("Hello World")

@Counter(0)
def g():
    print("Hello World")

print("#" * 80)

# Use the explicit decorator syntax
f = Counter(0)(f)

# Call the decorated functions
for _ in range(10):
    f()

print("#" * 80)

# Use Python's decorator syntax
for _ in range(10):
    g()

```

Example: Class decorator for classes

```

# Example: Class as a decorator for a class
class Counter(object):

```



```

# The constructor accepts the parameter passed to the decorator
def __init__(self, start_value):
    self.counter = start_value

# The __call__ method is called when the class is used as a decorator
def __call__(self, cls):

    # Modify the class by adding an attribute with the specified value
    cls.counter = self.counter

    # Return the modified class
    return cls

# Apply the class decorator with a parameter
@Counter(start_value=1)
class DecoratedClass(object):
    pass

# Use the explicit decorator syntax
DecoratedClass = Counter(start_value=1)(DecoratedClass)
obj = DecoratedClass()
print(obj.counter) # Output: 1

# Use Python's decorator syntax
obj = DecoratedClass()
print(obj.counter) # Output: Custom Value

```

9.15. Nested classes

In Python, you can define a class inside another class, creating what's known as a nested or inner class. Nested classes can be a useful way to organize and encapsulate related functionality and it is very often used to create namespaces for constants, settings and helper classes. A namespace is a dictionary that maps names to objects. It is used to avoid name collisions and to make it easier to find the names you are looking for.

```

# Example: Nested classes for constants, settings, etc.

class Settings:

    LANG = "English"
    THEME = "Light"
    IP_ADDR = "192.168.210.10"
    PORT = 8080

    class AdvancedSettings:

```

```

    ENABLE_LOGGING = False
    MAX_CONNECTIONS = 10

class ExperimentalSettings:

    ENABLE_NEW_FEATURE = False

# Access the basic settings
print(Settings.THEME)

# Access the advanced settings
print(Settings.AdvancedSettings.ENABLE_LOGGING) # Output: False

# Access the experimental settings
print(Settings.ExperimentalSettings.ENABLE_NEW_FEATURE) # Output: False

```

9.16. Named Constructors

Named constructors are special methods inside a class that create and initialize the objects of a the class. They are implemented either using the `@classmethod` decorator or using the `@staticmethod` decorator.

```

class MyClass(object):

    @classmethod
    def named_constructor(cls, arg1, arg2, ...):
        return cls(arg1, arg2, ...)

    @staticmethod
    def named_constructor(arg1, arg2, ...):
        MyClass(arg1, arg2, ...)

```

Named constructors are a popular choice for creating objects in Python because they are more descriptive than the default constructor and they can be used to create objects in different ways.

Example: Create object with an alternative constructor

```

# Example: Named constructors as alternative constructors

class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @classmethod
    def from_diagonal(cls, x1, y1, x2, y2):
        width = abs(x2 - x1)

```

```

        height = abs(y2 - y1)
        return cls(width, height)

# Create a square using the default constructor
rect1 = Rectangle(1, 1)
print(rect1.width, rect1.height)

# Create the same square using the named constructor
rect2 = Rectangle.from_diagonal(1, 1, 2, 2)
print(rect2.width, rect2.height)

```

Example: Reduce the number of parameters of the default constructor

```

# Example: Reduce the number of parameters

class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @classmethod
    def square(cls, size):
        return cls(size, size)

# Create a square using the default constructor
square1 = Rectangle(1, 1)
print(square1.width, square1.height)

# Create the same square using the named constructor
square2 = Rectangle.square(size=1)
print(square2.width, square2.height)

```

Example: Create objects from different sources (file, database, etc.)

```

# Example: Different sources to create and initialize objects
import os

class Person(object):

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def from_file(cls, file):

```

```

        with open(file, 'r') as f:
            data = f.read()
            name, age = data.split(',')

        return cls(name=name, age=age)

# Create a file named profile.txt with the following contents:
# mayank,27

with open('profile.txt', 'w') as f:
    f.write('mayank,27')

person = Person.from_file('profile.txt')
print(person.name, person.age)

# Delete the file profile.txt
os.remove('profile.txt')

```

Example: Create objects with different representations (e.g. JSON, XML, etc.)

```

# Example: Different internal representations of the same object

class Person(object):

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def from_json(cls, data):
        return cls(**data)

    @staticmethod
    def from_csv(data):
        name = data[0]
        age = data[1]
        return Person(name=name, age=age)

# Mock data
json_data = {'name': 'mayank', 'age': 27}
csv_data = [('mayank', 27), ]

person = Person.from_json(json_data)
print(person.name, person.age)

person = Person.from_csv(csv_data[0])
print(person.name, person.age)

```

9.17. Introspection

The class introspection is the process of examining a class's attributes. It is used to get information about a class.

- use `__doc__` to get the docstring of a class
- use `__class__` to get the class of an object
- use `__bases__` to get the base classes of a class
- use `__subclasses__` to get the subclasses of a class
- use `__mro__` to get the method resolution order of a class
- use `__name__` to get the name of a class
- use `__dict__` to get the attributes of a class
- use `__slots__` to get the slots of a class
- use `__annotations__` to get the annotations of a class
- use `__code__` to get the code object of a class

Chapter 10. Docstrings

Docstrings are optional strings enclosed in triple quotes immediately after the class, function, or method definition. They provide a description of the class, function, or method. Docstrings are used to document your code. They can be accessed using the `__doc__` attribute of the class, function, or method.

10.1. Class docstrings

```
# Example: Docstrings in a class

class TestClass(object):
    """ This is a docstring for the class

    Args:
        test (int): This is a docstring for the constructor argument

    Example:

        >>> test_class = TestClass(1)
        >>> test_class.test_function(1)
        'This is a test function with argument 1'

    """

    def __init__(self, test):
        """ This is a docstring for the constructor """
        self.test = test

    def test_function(self, test):
        """ This is a docstring for the function

        Args:
            test (int): This is a docstring for the function argument

        Raises:
            ValueError: If test is None

        Returns:
            str: A string with the argument

        """

        if test is None:
            raise ValueError("test cannot be None")

        return "This is a test function with argument {}".format(test)
```

```
print(TestClass.__doc__)
print(TestClass.test_function.__doc__)
```

10.2. Function docstrings

```
def myfunction(a, b, c):
    """This is a docstring for myfunction

    Args:
        a (int): This is the first argument
        b (int): This is the second argument
        c (int): This is the third argument

    Raises:
        ValueError: If a is less than 0

    Returns:
        int: This is the return value

    Example:
        >>> myfunction(1, 2, 3)
        6

    """

    if a < 0:
        raise ValueError('a must be greater than 0')

    return a + b + c

print(myfunction.__doc__)
```

10.3. Module docstrings

A python module is a file that contains python code. It is used to organize related code into a single file. A module typically contains classes, functions, and variables. The docstring of a module is the first string in the module.

```
# Example: Docstrings for modules
# =====
# Copyright 2023 by <Author>
#
#                               All Rights Reserved
#
# Permission to use, copy, modify, and distribute this software and
```

```
# its documentation for any purpose and without fee is hereby
# granted, provided that the above copyright notice appear in all
# copies and that both that copyright notice and this permission
# notice appear in supporting documentation, and that the name of
# <AUTHOR> not be used in advertising or publicity pertaining to
# distribution of the software without specific, written prior
# permission.
#
# <AUTHOR> DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
# INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
# NO EVENT SHALL <AUTHOR> BE LIABLE FOR ANY SPECIAL, INDIRECT OR
# CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
# NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
# CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
# =====
```

```
"""
```

Math Operations Module

This module provides a set of mathematical operations, including addition, subtraction, multiplication, and division.

Usage:

- Import this module using 'import math_operations'.
- Call functions using 'math_operations.add()', 'math_operations.subtract()', 'math_operations.multiply()', and 'math_operations.divide()'.

Example:

```
>>> import math
>>> "{:.2f}".format(math.sin(math.radians(90)))
'1.00'
>>> "{:.2f}".format(math.cos(math.radians(90)))
'0.00'
```

```
"""
```

```
class HelperClass(object):
```

```
    """This is a helper class for the module"""
    pass
```

```
def add(x, y):
```

```
    """Add two numbers."""
    return x + y
```

```
def subtract(x, y):
```

```
    """Subtract one number from another."""
    return x - y
```



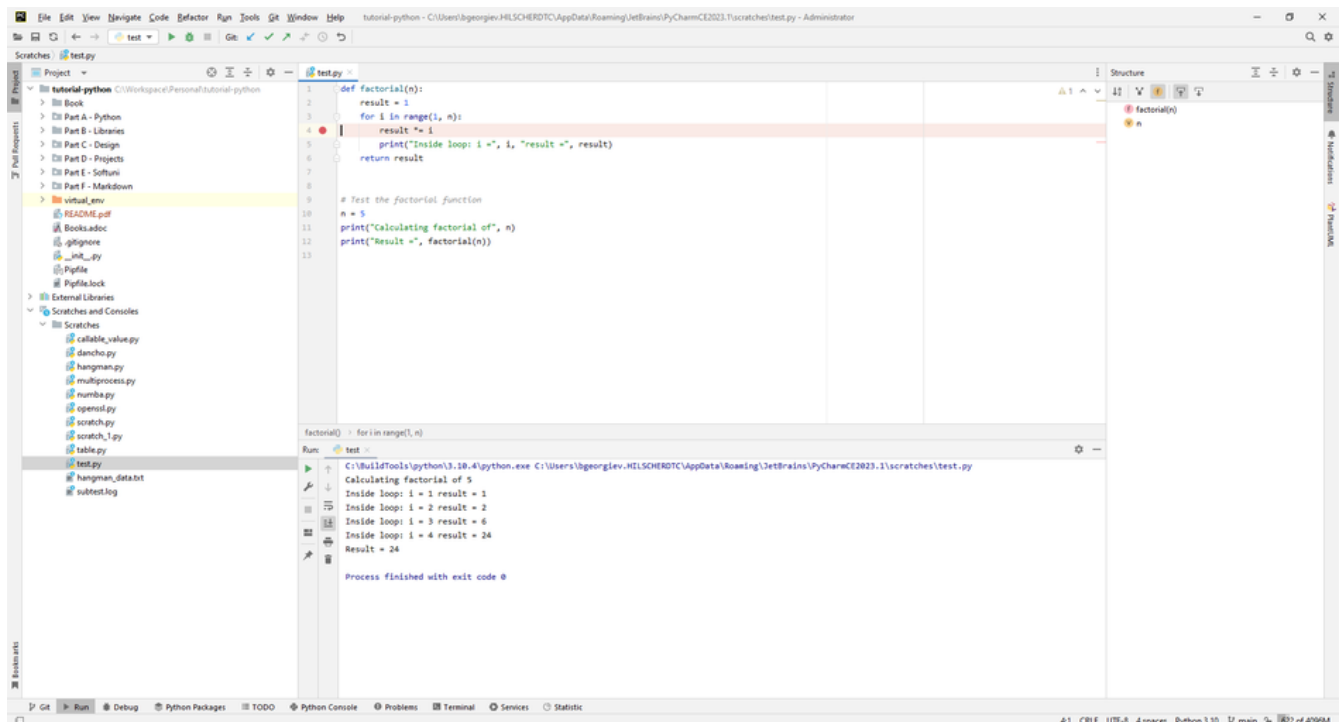
```
def multiply(x, y):
    """Multiply two numbers."""
    return x * y

def divide(x, y):
    """Divide one number by another."""
    if y == 0:
        raise ValueError("Division by zero is not allowed.")
    return x / y

if __name__ == "__main__":
    # Example usage of the module's functions
    result = add(5, 3)
    print("Result of addition:", result)
```

Chapter 11. Debugging

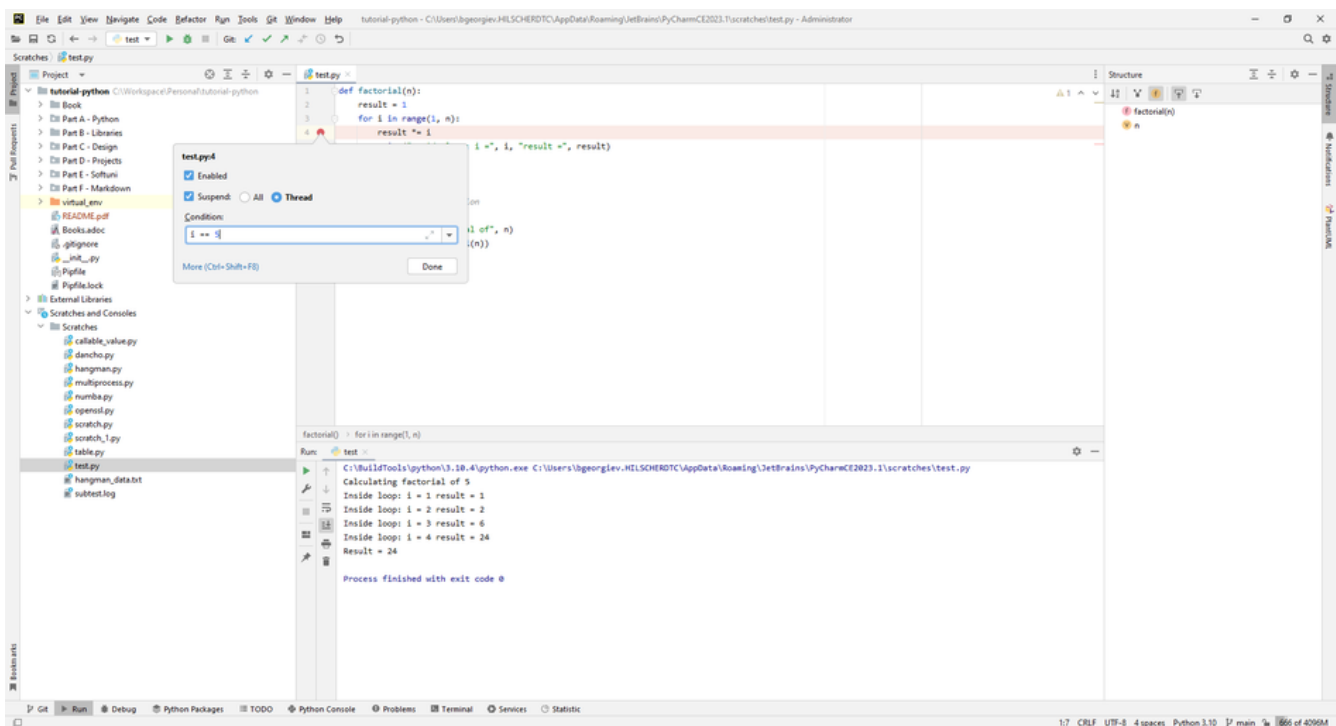
11.1. Breakpoints



A breakpoint is a point in the program where the execution stops. It is used to pause the program and inspect the state of the program at that point. Breakpoints are used to debug programs. They are used to find bugs in the program. Breakpoints can be set in the following ways:

- Using any IDE (see red dot in the image above)
- Using the `breakpoint()` function (only in Python 3.7 and above)
- Using the `pdb` module

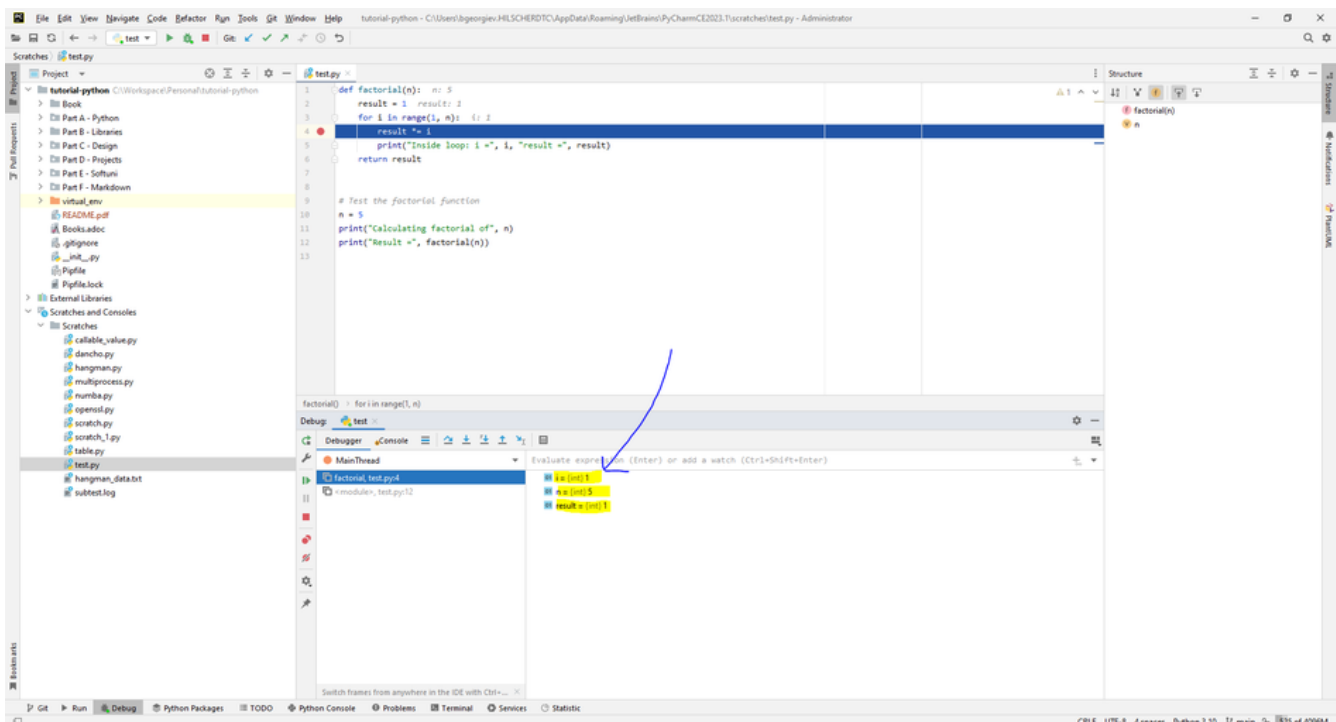
11.2. Conditional breakpoints



A conditional breakpoint is a breakpoint that is only triggered when a condition is met. It is used to pause the program and inspect the state of the program when a condition is met. Conditional breakpoints are used to debug programs. They are used to find bugs in the program. Conditional breakpoints can be set in the following ways:

- Using an IDE (right click on the breakpoint and edit the conditions)
- Using the `breakpoint()` function (only in Python 3.7 and above)
- Using the `pdb` module

11.3. Watchlists



A watchlist is a list of variables that are monitored while the program is running. It is used to inspect the values of variables while the program is running. Watchlists are used to debug programs. They are used to find bugs in the program. Watchlists can be set in the following ways:

- Using any IDE (while in debug mode)
- Using the `pdb` module

Chapter 12. Linting

The static code analysis process (linting for short) is the process of checking the source code for type errors, syntax errors, and styling errors. It is used to ensure that the code is correct and follows the Python style guide. In this tutorial we will use the integrated tools in PyCharm but further reading is recommended on the topic.

The following resources are a good starting point:

- <https://realpython.com/python-code-quality/>
- <https://realpython.com/python-type-checking/>

The following list contains some of the most popular code analysis tools:

- **mypy** (type checking)
- **pylint** (error and style checking)
- **flake8** (error, style and complexity checking)
- **pycodestyle** (style checking)
- **mccabe** (complexity checking)
- **bandit** (security checking)
- **pyroma** (project checking)
- **pydocstyle** (docstring checking)
- **docformatter** (docstring formatting)
- **pydocstringformatter** (docstring formatting)

PART B: Advanced Python

Chapter 1. Requirements

1.1. Python Installation

- [Python 3.7 or higher](#) (*recommended*)
- [Python 2.7.18](#)

1.2. Python IDE

- [PyCharm](#) (*recommended*)
- [Visual Studio Code](#)

1.3. Version Control

- [Git Client](#) (*recommended*)
- [GitHub Account](#) (*recommended*)
- [SVN Client](#)

1.4. Style Guide

- [PEP-08](#) (*mandatory*)
- [Google Style Guide](#) (*mandatory*)

Chapter 2. OOP Principles

The OOP principles are a set of principles that are used to maximize the quality of the software codebase. Using OOP programming will result in a more flexible, maintainable, and reusable code.

2.1. Abstraction (Modeling)

Abstraction is a fundamental concept in object-oriented programming (OOP) that refers to the process of exposing only the relevant and essential data and behavior to the users without showing the internal implementation details. It is used to manage complexity and reduce dependencies between classes.

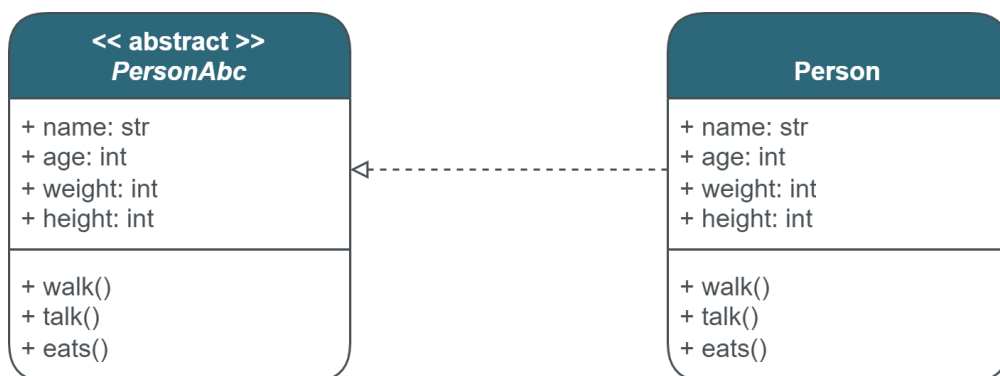
Now let's look at an example of abstraction in Python. Let's say you want to model a person. You look around you and see that each person is unique, e.g. name, age, height, weight, etc. However, there are some common characteristics that all people share, e.g. they can walk, talk, eat, etc. You can use abstraction to model a person by focusing on the common characteristics and ignoring the unique characteristics.

In abstract terms, a person is an object that has a name, an age, a height, a weight, etc. It can walk, talk, eat, etc. The name, age, height, weight, etc. are the attributes of the person. The walk, talk, eat, etc. are the methods of the person. The attributes and methods of a person are called the interface of the person. The interface of a person defines the ways in which you can interact with a person.

In complex software development abstraction is achieved by using **abstract classes** or **interfaces**. The latter is not available in Python but can be emulated using abstract classes with abstract methods only.

They key points about abstraction are:

- Focuses on the "what an object does" rather than "how it does it"
- Allows you to define a clear and high-level interface for interacting with objects
- Uses abstract classes or interfaces provide a blueprint for other classes to follow
- Manages complexity and enhances code readability



Example: Abstraction with abstract classes


```

from abc import ABCMeta, abstractmethod
from six import with_metaclass

class PersonAbc(with_metaclass(ABCMeta)):
    """ The abstract base class for a person answers what a person shall be able to
    do. """

    def __init__(self):

        # WHAT DOES THE PERSON HAVE?

        self.name = 'Bob'
        self.age = 42
        self.weight = 80
        self.height = 180

        # WHAT DOES THE PERSON DO?

    @abstractmethod
    def walk(self):
        # Still abstract, because we don't know how a person walks.
        pass

    @abstractmethod
    def talk(self):
        # Still abstract, because we don't know how a person talks.
        pass

    @abstractmethod
    def eats(self):
        # Still abstract, because we don't know how a person eats.
        pass

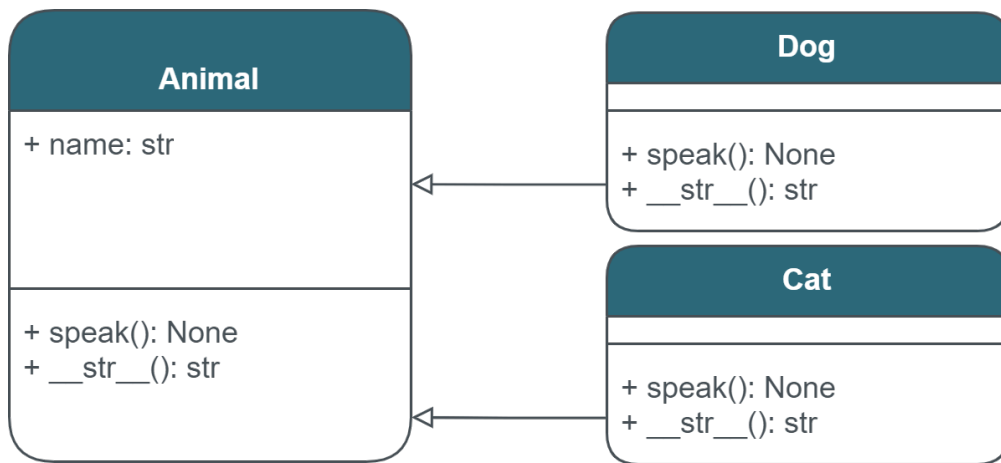
```

2.2. Inheritance (Code Reuse)

Inheritance is the process of creating a new class from an existing class. The subclass inherits the attributes and methods of the superclass and can add its own attributes and methods. The inheritance is the opposite process of abstraction and is used to represent an *is-a* relationship, e.g a student is a person.

Key points about inheritance:

- Reuses code from an existing class
- Represents an *is-a* relationship
- Focuses on the "what" an object is and how it behaves



Example: Inheritance

```
class Animal(object):

    def __init__(self, name):
        self.name = name

    def speak(self):
        print("I am an animal")

    def __str__(self):
        return "Animal: {}".format(self.name)

class Dog(Animal):

    def speak(self):
        print("I am a dog")

    def __str__(self):
        return "Dog: {}".format(self.name)

class Cat(Animal):

    def speak(self):
        print("I am a cat")

    def __str__(self):
        return "Cat: {}".format(self.name)

animal = Animal("Animal")
animal.speak()

dog = Dog("Dog")
dog.speak()
```

```
cat = Cat("Cat")
cat.speak()
```

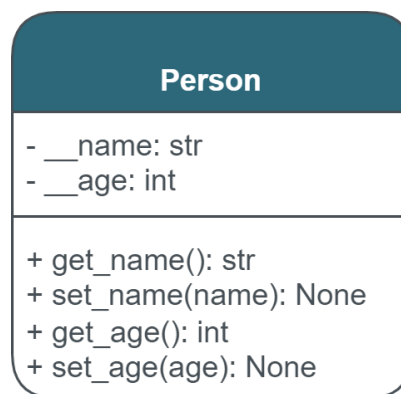
2.3. Encapsulation (Data Hiding)

This is a fundamental concept in object-oriented programming (OOP) that refers to the hiding of data through the use of access modifiers and special methods called getters and setters. It is used to protect the internal state of an object from the outside world.

Python offers no mechanism for making attributes private and relies on naming conventions to distinguish between public, protected and private attributes. Protected attributes are prefixed with a single underscore and private attributes are prefixed with a double underscore.

Key points about encapsulation:

- Uses access modifiers to hide the internal state of an object
- Provides getters and setters to access and modify the internal state of an object
- Defines protection logic for invalid values in the setters



```
# Example: Encapsulation with getters and setters

class Person(object):
    """ The abstract base class for a person answers
        what a person shall be able to do.
    """

    def __init__(self, name, age):

        # Modify the access to the internal state
        self.__name = name
        self.__age = age

    @property
    def name(self):
        """ The getter for the name """
        return self.__name
```

```

@name.setter
def name(self, name):
    """ The setter for the name """

    # Protection logic
    if name is None:
        raise ValueError("name cannot be None")

    self.__name = name

@property
def age(self):
    """ The getter for the age """

    return self.__age

@age.setter
def age(self, age):
    """ The setter for the age """

    # Protection logic
    if self.__age is None:
        raise ValueError("age cannot be None")

    elif self.__age < 0:
        raise ValueError("age cannot be negative")

    elif self.__age > 150:
        raise ValueError("age cannot be greater than 150")

    self.__age = age

person = Person("John", 30)
print(person.name)
print(person.age)

```

Public attributes and methods are accessible from outside the class. They are used to define the interface of the class. Typically all getter and setter methods are public. Public attributes and methods are not prefixed with an underscore.

Protected attributes and methods are accessible from outside the class but only in a subclass. They may or may not have a getter or setter method. Protected attributes and methods are prefixed with a single underscore.

Private attributes and methods are not accessible from outside the class. They are used to hide the internal implementation details of a class. Private attributes and methods are prefixed with a double underscore.

```
# Example: Access modifiers
```

```

class AccessModifiers(object):

    def __init__(self):

        # Public: Accessible from anywhere
        self.public = "public"

        # Protected: Accessible from the class and subclasses
        self._protected = "protected"

        # Private: Accessible only from the class
        self.__private = "private"

class AccessModifiersChild(AccessModifiers):

    def __init__(self):
        super(AccessModifiersChild, self).__init__()

        # Public: Accessible from anywhere
        print(self.public)

        # Protected: Accessible from the class and subclasses
        print(self._protected)

        # Private: Not accessible from the class
        try:
            print(self.__private)

        except AttributeError as e:
            print(e)

test = AccessModifiersChild()

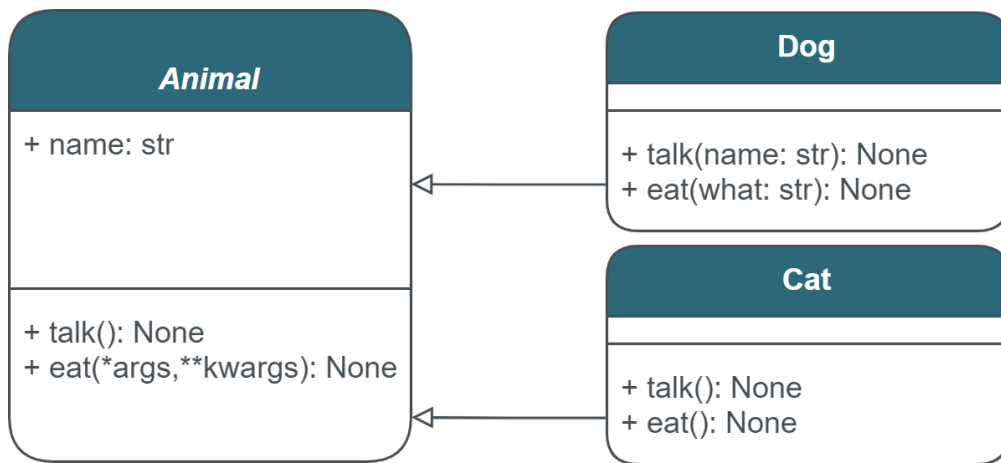
```

2.4. Polymorphism (Flexibility)

Polymorphism is the ability of an object to take on many forms. It is used to represent an *is-a* relationship. Polymorphism is achieved by using inheritance and overriding or overloading the methods of the superclass.

Overriding is the process of redefining a method in the subclass. The method in the subclass has the same name and signature as the method in the superclass.

Overloading is the process of defining multiple methods with the same name but different signatures in the same class.



Example: Polymorphism

```
class Animal(object):
    def __init__(self, name):
        self.name = name

    # Abstract method that supports overriding, but the linter will complain on
    # overloading
    def talk(self):
        raise NotImplementedError

    # Abstract method that supports both overriding and overloading
    def eat(self, *args, **kwargs):
        raise NotImplementedError

class Cat(Animal):

    # Override the talk method
    def talk(self):
        print("Meow!")

    def eat(self):
        print("Cat is eating")

class Dog(Animal):

    # Overload the talk method
    def talk(self, name):
        print("Woof!")

    def eat(self, what):
        print("Dog is eating {}".format(what))

cat = Cat("Kitty")
cat.talk()
```

```
cat.eat()
```

```
dog = Dog("Doggy")  
dog.talk("Doggy")  
dog.eat("bone")
```

2.5. Association (Relationships)

Association represents a relationship between classes. It describes how objects from different classes interact and collaborate to achieve a certain functionality. Associations define the ways in which objects can relate to each other and exchange information.

Associations may have also a direction. For example, a car has an engine, but an engine does not have a car. This is a unidirectional association. Possible directions for associations are:

- non-directional
- uni-directional
- bi-directional

Associations may also have a multiplicity. For example, a car has one engine, but an engine may belong multiple cars. In this case, the association is one-to-many. Possible multiplicities for associations are:

- one-to-one
- one-to-many
- many-to-many

Key points about associations:

- Defines the ways in which objects can relate to each other and exchange information
- May have a direction (non-directional, uni-directional, bi-directional)
- May have a multiplicity (one-to-one, one-to-many, many-to-many)

2.5.1. Dependency

Dependency is a relationship between two classes in which one class depends on the other. It is used to represent a "uses-a" relationship. For example a car can be ignited only if the owner has the right key. In abstract terms the key is the interface and if its implementation is changed then the car will not be able to start.

Key points about dependency:

- "Uses-A" Relationship
- Unidirectional
- Weak Coupling
- Short Lifetime

2.5.2. Composition

Composition allows you to build complex objects by combining simpler objects. It involves creating a class that contains one or more objects of other classes as its members. In composition, the containing class is responsible for creating and managing the objects it contains, and it can delegate certain responsibilities to those contained objects.

While both composition and inheritance facilitate code reuse, composition is often preferred over inheritance when you want to avoid tight coupling between classes or when the "is-a" relationship isn't appropriate. Composition offers more flexibility and allows you to create more maintainable code in complex scenarios.

Key points about composition:

- "Has-A", "Composed-Of" Relationship
- Delegate Responsibilities (the whole delegates responsibilities to the parts)
- Strong Coupling (the whole is responsible for the parts)
- Single Ownership (if the whole is destroyed, the parts are destroyed)
- Shared Lifetime (if one part is destroyed, the whole is destroyed)

2.5.3. Aggregation

Aggregation is another concept in object-oriented programming (OOP) that involves a "whole-part" relationship between classes. It's a form of association where one class represents a larger structure (the whole), and it contains or is composed of other classes (the parts). Aggregation is a more specialized form of composition, emphasizing a looser relationship between the whole and its parts.

While both aggregation and composition involve relationships between classes, composition implies a stronger relationship, where the contained objects are owned and managed by the containing object. Aggregation, on the other hand, represents a looser relationship where the parts can exist independently.

- "Has-A" Relationship
- Delegate Responsibilities (the whole delegates responsibilities to the parts)
- Looser Coupling (the whole is not responsible for the parts)
- Shared Ownership (if the whole is destroyed, the parts are not destroyed)
- Independent Lifetimes (if one part is destroyed, the whole is not destroyed)

Chapter 3. SOLID Design

The SOLID principles enhance the OOP programming by defining a set of design principles that are used to minimize the change in the software codebase.

They were introduced by Robert C. Martin (Uncle Bob) in his 2000 paper, "Design Principles and Design Patterns". The SOLID acronym was introduced later by Michael Feathers.

The SOLID principles are:

- Single Responsibility Principle (focus one thing at a time)
- Open/Closed Principle (extend instead of modify)
- Liskov Substitution Principle (children shall not break on using parents behavior)
- Interface Segregation Principle (many small interfaces are better than one big interface)
- Dependency Inversion Principle (depend on abstract interfaces, not on concrete objects)

3.1. Single Responsibility Principle

Reduce the interdependencies by reducing the number of attributes and methods in the class. A change in one class shall not break the code in another class. This minimizes the number of reasons to change a class.

The single responsibility principle states that a class or function should have a single reason to change. By splitting a class or function into several classes or functions, you can reduce the number of reasons to change it. Each part can be developed and tested independently.

Despite being one of the easiest principles to understand, the single responsibility principle is also one of the most difficult to apply. The reason for this is that it is not always clear what constitutes a single responsibility and how feasible it is to split a class or function into several classes or functions.

Example: Breaking the single responsibility principle

This is a simple example of a class that violates the single responsibility principle. The class too many responsibilities and a change might break the code that uses it. For example, if you change the data format from JSON to XML, you will have to modify the class to handle the new data format. This is a clear violation of the single responsibility principle.

MyCustomFileFormat

+ filename: str
+ data: str

+ read(): str
+ write(text): None
+ process(): str

Example : An example that violates the Single Responsibility Principle

```
class MyCustomFileFormat(object):
    """ This class violates the Single Responsibility Principle because:

        1. It reads the file and stores the data
        2. It writes data to the file
        3. It processes the stored data

    """

    def __init__(self, filename):
        self.filename = filename
        self.data = ""

    def read(self):
        # Responsibility: Read the file
        with open(self.filename, "r") as f:
            self.data = f.read()
        return self.data

    def write(self, text):
        # Responsibility: Write to the file
        with open(self.filename, "w") as f:
            f.write(text)

    def process(self):
        # Responsibility: Process the data
        self.data = self.data.upper()
        return self.data

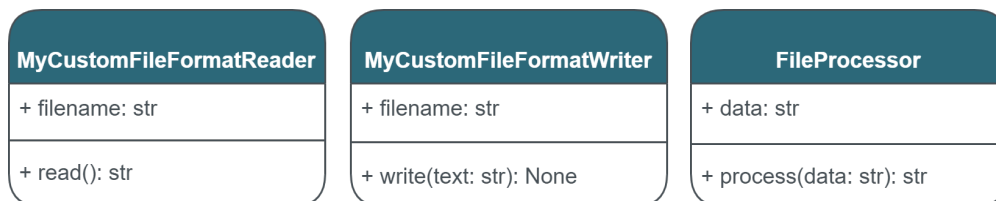
# Create the file reader
reader = MyCustomFileFormat("test_input.myformat")
print(reader.read())

# Process the data
print(reader.process())
```

```
# Create the file writer
writer = MyCustomFileFormat("test_output.myformat")
writer.write(reader.data)
```

Example: Following the single responsibility principle

This is an implementation of the single responsibility principle. The class has been split into three classes, one for reading data from a file, one for processing the data, and one for writing the data to a file. This way, each class has a single responsibility and a change will not break the code using the other classes. Each class can be developed, enhanced, and tested independently.



```
# Example : An example that follows the Single Responsibility Principle

# By splitting the class into three classes, we can now reuse the classes in
# other parts of the program. A change in one class will not affect the other
# classes. This makes the code more maintainable and easier to understand.
```

```
class MyCustomFileFormatReader(object):
    # Responsibility: Read the file

    def __init__(self, filename):
        self.filename = filename

    def read(self):
        with open(self.filename, "r") as f:
            return f.read()

class MyCustomFileFormatWriter(object):
    # Responsibility: Write to the file

    def __init__(self, filename):
        self.filename = filename

    def write(self, text):
        with open(self.filename, "w") as f:
            f.write(text)

class FileProcessor(object):
    # Responsibility: Process the data

    def __init__(self):
        self.data = ""
```

```

def process(self, data):
    self.data = data.lower()
    return self.data

# Create the file reader
reader = MyCustomFileFormatReader("test_input.myformat")
content = reader.read()
print(content)

# Process the data
processor = FileProcessor()
new_content = processor.process(content)
print(new_content)

# Create the file writer
writer = MyCustomFileFormatWriter("test_output.myformat")
writer.write(new_content)

```

3.2. Open/Closed Principle

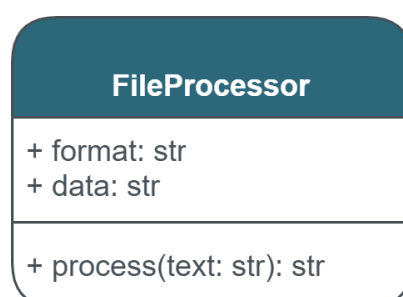
Never touch a working system unless it is to fix a bug. Replace or extend the existing functionality without modifying the existing code. This minimizes the change in the old codebase.

A module, class or function should be open for extension but closed for modification. This means that a module, class or function shall be extended without having to modify its source code, except for bug fixes.

In Python, you can use inheritance and composition to achieve this principle. This can be achieved through:

- Composition (adding new functionality through composition)
- Subclassing and Polymorphism (extending existing functionality of a parent class)
- Mixins (adding new functionality to a class through multiple inheritance)
- Monkey patching (adding new functionality at runtime)

Example: Breaking the open/closed principle



Example : A bad example that violates the Open/Closed Principle

```
class FileProcessor(object):

    def __init__(self, file_format="upper"):
        self.format = file_format
        self.data = ""

    def process(self, text):

        # This violates the Open/Closed Principle because a new file format
        # cannot be added without modifying the FileProcessor class.

        if self.format == "upper":
            # Responsibility: Process the data
            self.data = text.upper()

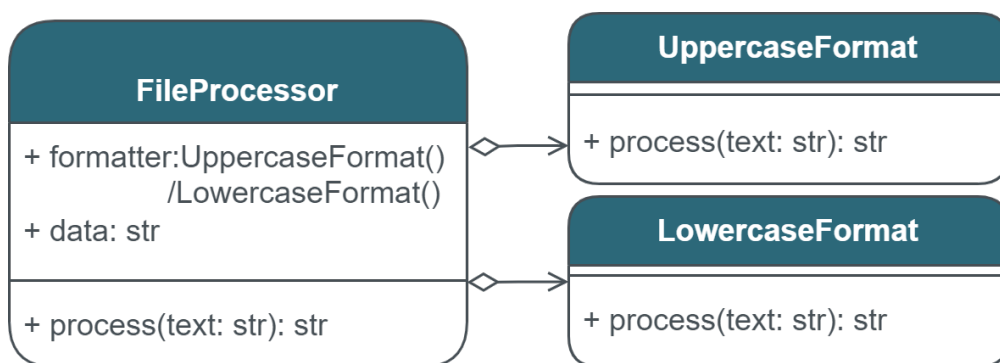
        elif self.format == "lower":
            # Responsibility: Process the data
            self.data = text.lower()

        else:
            raise ValueError("Invalid format: {}".format(format))

        return self.data

process = FileProcessor(file_format="upper")
print(process.process("Hello World!"))
```

Example: Following the open/closed principle



Example : A good example that follows the Open/Closed Principle

```
class LowercaseFormat(object):

    @staticmethod
    def process(text):
        return text.lower()
```

```

class UppercaseFormat(object):
    # Responsibility: Process the data

    @staticmethod
    def process(text):
        return text.lower()

# Add more file formats here...

class FileProcessor(object):

    # Use composition to extend the functionality of the FileProcessor class
    # without modifying the class itself.

    def __init__(self, file_format):
        self.formatter = file_format
        self.data = ""

    def process(self, text):
        self.data = self.formatter.process(text)
        return self.data

processor = FileProcessor(LowercaseFormat())
print(processor.process("Hello World!"))

```

3.3. Liskov Substitution Principle

A method shall be implemented and used only there where it makes sense. This minimizes changes accross the class hierarchy.

This principle states that children shall not break on using the parents behavior. If you have a class hierarchy, then you should be able to use any subclass in place of its parent class without an unexpected behavior. If this is not the case, then you are violating the Liskov substitution principle.

A very common example of violating this principle is when a subclass overrides a method of its parent class to do nothing. Or when the method has to check the type of an object to determine what to do with it.

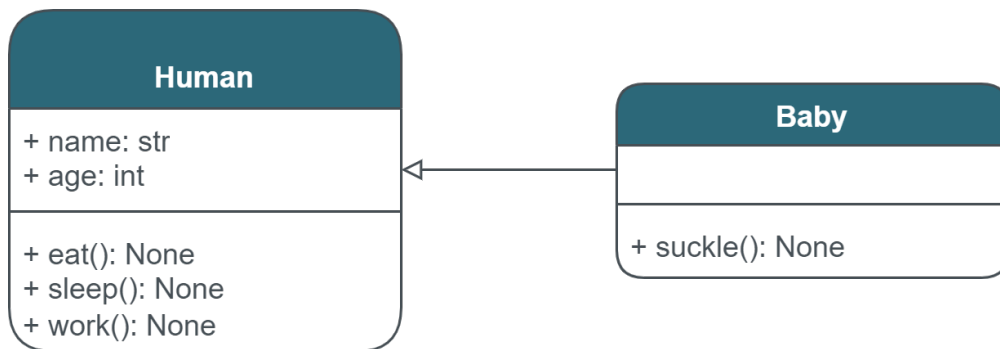
The Liskov substitution is used to ensure that class hierarchies are well designed and to prevent unexpected behavior when using polymorphism. A single method should be implemented only there where it makes sense.

Some code smells showing the violation of this principle:

- Override a method of a base class to do nothing

- Overridden methods change their behavior (e.g. return type, arguments, exceptions, etc.)
- The parent or the child has to check the type of an object to determine what to do with it

Example: Violating the Liskov substitution principle



Example: A bad example that violates the Liskov Substitution Principle

```

class Human(object):

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def eat(self):
        print("{} eating".format(self.name))

    def sleep(self):
        print("{} sleeping".format(self.name))

    def work(self):

        # Code smell: Type checking or conditional logic to determine the
        # behaviour and thus the child class and the parent class are not
        # substitutable. We have a divergent behaviour for Human and Baby
        # when the work method is called.

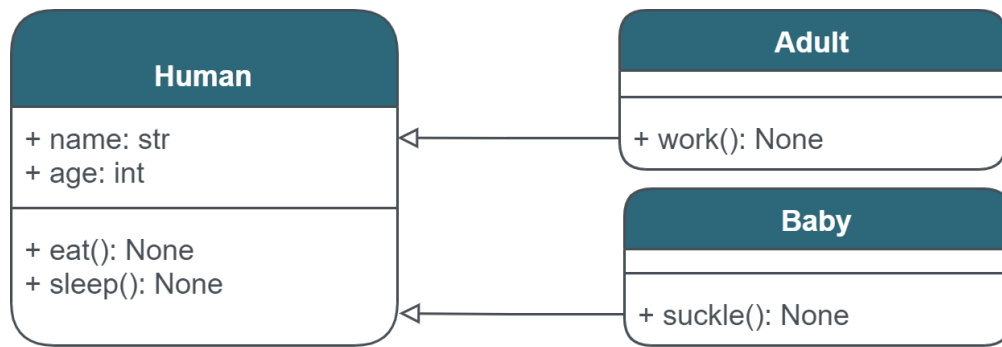
        if type(self) == Baby:
            raise RuntimeError("Too young to work")

        print("{} working".format(self.name))

class Baby(Human):

    def suckle(self):
        print("{} suckling".format(self.name))
  
```

Example: Following the Liskov substitution principle



Example: A good example that follows the Liskov Substitution Principle

```
class Human(object):

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def eat(self):
        print("{} eating".format(self.name))

    def sleep(self):
        print("{} sleeping".format(self.name))

class Adult(Human):

    # GOOD: Adult can be substituted for Human because it implements all the
    # methods of the Human class and it does not violate the Liskov
    # Substitution Principle.

    def work(self):
        print("{} working".format(self.name))

class Baby(Human):

    # GOOD: Baby can be substituted for Human because it implements all the
    # methods of the Human class and it does not violate the Liskov
    # Substitution Principle.

    def suckle(self):
        print("{} suckling".format(self.name))
```

3.4. Interface Segregation Principle

A method that is used by several but not all subclasses shall be in a separate interface or class and be used or implemented only by the subclasses that

need it. This minimizes changes accross the class hierarchy.

The interface segregation principle states that many small interfaces are better than one big interface. This means that a class should not be forced to implement methods that it does not need.

This principles is applied when a parent class has many subtypes and sometimes only a few of them need to use a particular method from the parent. In this case, it is better to split the interface into several smaller interfaces so that each subtype only implements the methods that it needs.

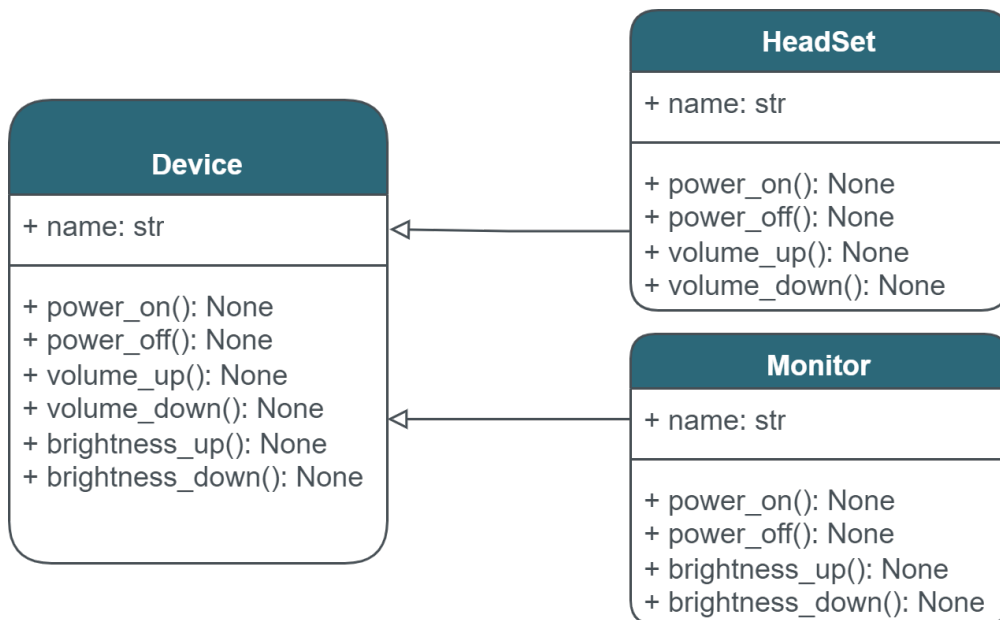
Let's say we have an abstract class for a vehicle that has a method for driving and a method for flying. We also have a class for a car that implements the driving method and a class for a plane that implements the flying method. This is a clear violation of the interface segregation principle because the car class does not need to implement the flying method and the plane class does not need to implement the driving method.

In this case, it is better to split the vehicle class into two smaller interfaces, one for driving and one for flying. This way, the car class only needs to implement the driving interface and the plane class only needs to implement the flying interface.

Some common examples of violating this principle are:

- A class implements a large number of methods that are not used by all subtypes
- A class has many abstract methods that are shall not be implemented by all subtypes

Example: Violating the interface segregation principle



Example: A bad example that violates the Interface Segregation Principle

```
class Device(object):

    def __init__(self, name):
        self.name = name
```

```

def power_on(self):
    pass

def power_off(self):
    pass

def volume_up(self):
    # Code smell: Used only by Headset
    pass

def volume_down(self):
    # Code smell: Used only by Headset
    pass

def brightness_up(self):
    # Code smell: Used only by Monitor
    pass

def brightness_down(self):
    # Code smell: Used only by Monitor
    pass

class HeadSet(Device):

    def __init__(self, name):
        super(HeadSet, self).__init__(name)

    def power_on(self):
        print("Headset powered on")

    def power_off(self):
        print("Headset powered off")

    def volume_up(self):
        print("Headset volume up")

    def volume_down(self):
        print("Headset volume down")

class Monitor(Device):

    def __init__(self, name):
        super(Monitor, self).__init__(name)

    def power_on(self):
        print("Monitor powered on")

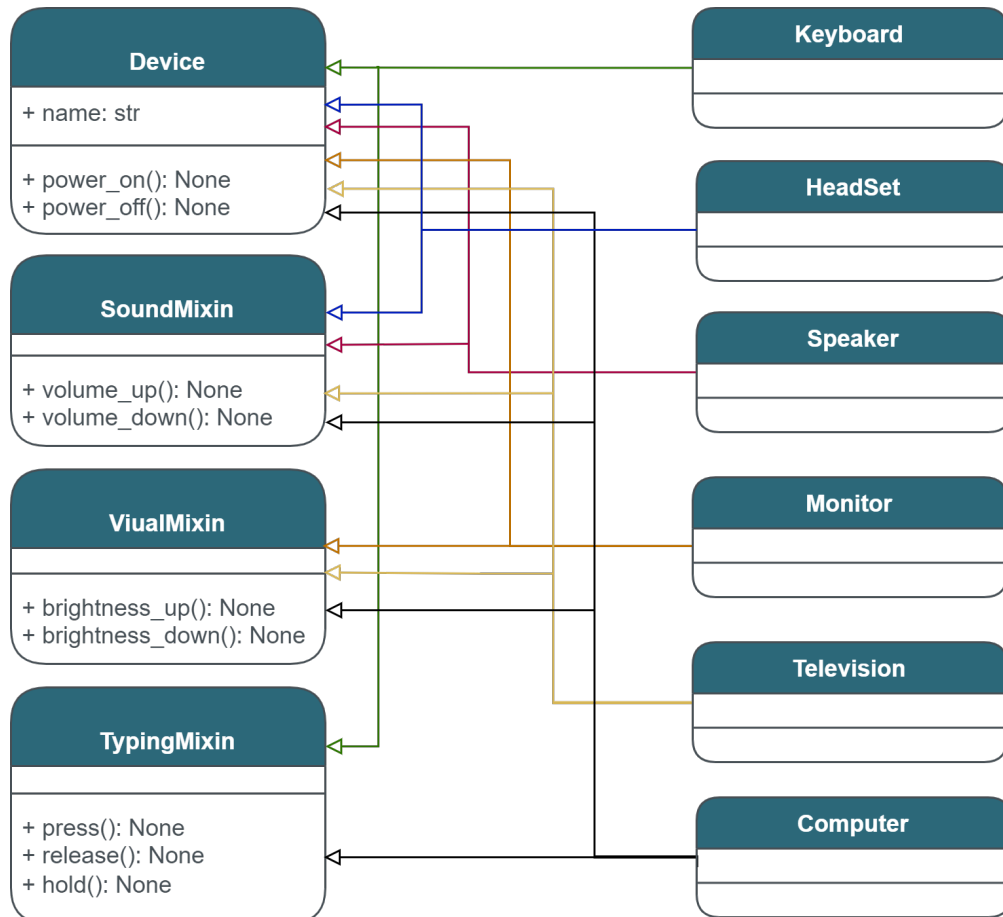
    def power_off(self):
        print("Monitor powered off")

```

```
def brightness_up(self):
    print("Monitor brightness up")

def brightness_down(self):
    print("Monitor brightness down")
```

Example: Following the interface segregation principle



Example: A good example that follows the Interface Segregation Principle

```
class Device(object):
    # Defines only the common methods for all devices

    def __init__(self, name, *args, **kwargs):
        super(Device, self).__init__(*args, **kwargs)
        self.name = name

    def power_on(self):
        pass

    def power_off(self):
        pass

class SoundMixin(object):
```

```

# Defines only the methods for all sound devices

def __init__(self, *args, **kwargs):
    super(SoundMixin, self).__init__(*args, **kwargs)

def volume_up(self):
    pass

def volume_down(self):
    pass

class VisualMixin(object):
    # Defines only the methods for all visual devices

    def __init__(self, *args, **kwargs):
        super(VisualMixin, self).__init__(*args, **kwargs)

    def brightness_up(self):
        pass

    def brightness_down(self):
        pass

class TypingMixin(object):
    # Defines only the methods for all typing devices

    def __init__(self, *args, **kwargs):
        super(TypingMixin, self).__init__(*args, **kwargs)

    def press(self):
        pass

    def release(self):
        pass

    def hold(self):
        pass

class Keyboard(Device, TypingMixin):
    pass

class HeadSet(Device, SoundMixin):
    pass

class Speaker(Device, SoundMixin):
    pass

```

```
class Monitor(Device, VisualMixin):
    pass

class Television(Device, SoundMixin, VisualMixin):
    pass

class Computer(Device, SoundMixin, VisualMixin, TypingMixin):
    pass
```

3.5. Dependency Inversion Principle

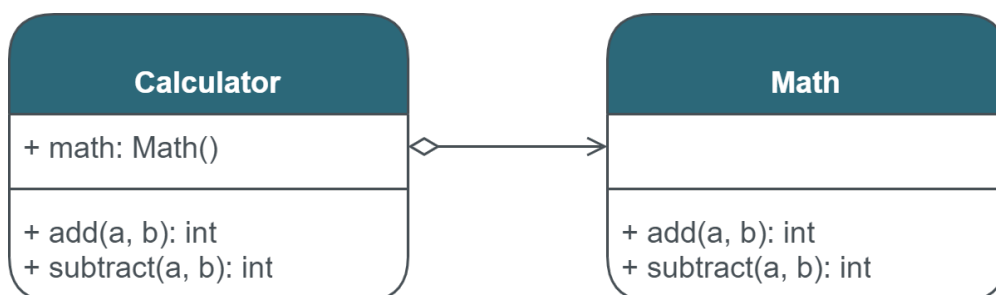
Define an agreement between the client and the server. The client is obliged to use the agreement and the server is obliged to implement it. This minimizes changes accross the product layers.

The dependency inversion principle is used to decouple high-level modules from low-level modules. It is used to reduce the coupling between modules and to make them more reusable. It is also used to make modules more testable.

As an example, let's say you have a class that depends on a database class. You can decouple the two classes defining an interface on how to access the database. The client class is obliged to use this interface and the data access class is obliged to implement it. This way, the two classes are decoupled from each other and can be used independently.

The client class gets the required interface either by using **dependency injection** or a factory method. Dependency injection is a technique where the required interface is passed to the client class through its constructor or through a setter method. A factory method is a method that returns an object of a class and can be called at runtime.

Example: Violating the dependency inversion principle



Example: A bad example that violates the Dependency Inversion Principle

```
class Math(object):
```

```

    @staticmethod
    def add(a, b):
        return a + b

    @staticmethod
    def subtract(a, b):
        return a - b

class Calculator(object):

    def __init__(self):

        # Code smell: Dependency relationship is hard-coded (dependency)
        # and not abstracted (injection)

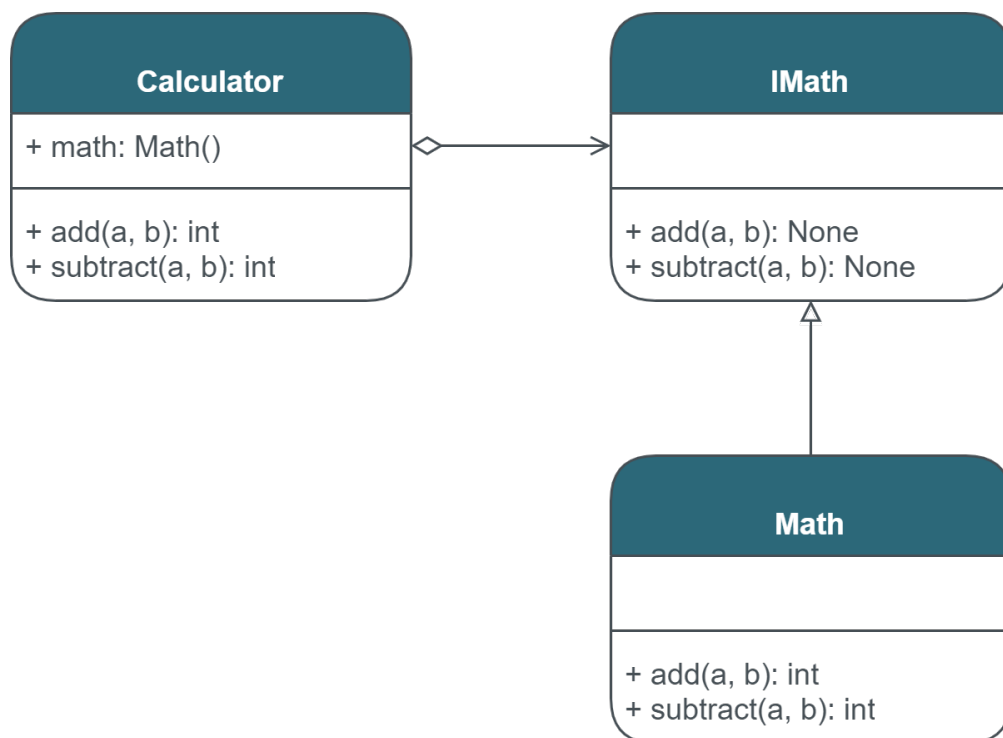
        self.math = Math()

    def add(self, a, b):
        result = self.math.add(a, b)
        return result

    def subtract(self, a, b):
        result = self.math.subtract(a, b)
        return result

```

Example: Following the dependency inversion principle



Example : A good example that follows the Dependency Inversion Principle

```

class IMath(object):
    """ Simplified interface for a math class """

    # GOOD: IMath is an abstraction that defines the contract for Math and
    # Calculator (the interface). It has no implementation details.

    @staticmethod
    def add(a, b):
        raise NotImplementedError()

    @staticmethod
    def subtract(a, b):
        raise NotImplementedError()

class Math(IMath):
    # GOOD: Both Math and Calculator depend on abstraction (MathAbc)

    @staticmethod
    def add(a, b):
        return a + b

    @staticmethod
    def subtract(a, b):
        return a - b

class Calculator(object):
    """A simple calculator class

    Args:
        math (IMath): An object that implements the IMath interface

    """

    def __init__(self, math):
        # GOOD: Both Math and Calculator depend on abstraction (MathAbc)
        self.math = math

    def add(self, a, b):
        result = self.math.add(a, b)
        return result

    def subtract(self, a, b):
        result = self.math.subtract(a, b)
        return result

```

Chapter 4. Design Patterns

The design patterns are a set of patterns that are used to solve common problems in software development. They are used to minimize the time and effort required to develop software based on a good quality codebase combining SOLID and best practices based on the experience of other developers.

4.1. Creational Patterns

The creational patterns are a set of design patterns that deal with the creation of objects. They are used to create objects in a systematic way that promotes reusability and flexibility. The creational patterns are:

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

4.1.1. Factory Method

The factory method pattern is a creational pattern that uses factory methods to create objects. A factory method is a method that returns an object of a class and can be called at runtime. It provides a way to hide the creation logic of an object. In Python the factory method is usually implemented as a static method or a class method.

```
# Example: Factory Method as a static Method

class Transport(object):
    def __init__(self):
        pass

    def deliver(self):
        pass

class Truck(Transport):
    def deliver(self):
        print("Delivering by land in a box")

class Ship(Transport):
    def deliver(self):
        print("Delivering by sea in a container")
```



```

class Logistic(object):
    def __init__(self, transport):
        self.transport = transport

    def deliver(self):
        self.transport.deliver()

class App(object):

    @staticmethod
    def select_transport():
        # Factory method

        # Get transport type
        token = input("Select transport type: ")

        # Convert to lowercase and remove whitespace
        transport_type = token.lower().strip(" \t\n\r")

        # Factory logic
        if transport_type == "truck":
            return Truck()

        elif transport_type == "ship":
            return Ship()

        else:
            raise ValueError("Invalid transport type")

    def run(self):

        # Select transport
        transport = self.select_transport()

        # Deliver
        logistic = Logistic(transport)
        logistic.deliver()

if __name__ == "__main__":
    app = App()
    app.run()

```

Example: Factory Method as a class method

```

class Transport(object):
    def __init__(self):

```

```

        pass

    def deliver(self):
        pass

class Truck(Transport):
    def deliver(self):
        print("Delivering by land in a box")

class Ship(Transport):
    def deliver(self):
        print("Delivering by sea in a container")

class Logistic(object):
    def __init__(self, transport):
        self.transport = transport

    def deliver(self):
        self.transport.deliver()

    @classmethod
    def from_json(cls, json):
        # Factory method

        # Get transport type
        token = json.get("transport_type")

        # Convert to lowercase and remove whitespace
        transport_type = token.lower().strip(" \t\n\r")

        # Factory logic
        if transport_type == "truck":
            return Truck()

        elif transport_type == "ship":
            return Ship()

        else:
            raise ValueError("Invalid transport type")

class App(object):

    @staticmethod
    def run():

        # Select transport
        json = {"transport_type": "truck"}

```

```

        transport = Logistic.from_json(json)

        # Deliver
        logistic = Logistic(transport)
        logistic.deliver()

if __name__ == "__main__":
    app = App()
    app.run()

```

4.1.2. Abstract Factory

The abstract factory pattern is a creational pattern that extends the factory method pattern. It is an abstract class that provides an interface consisting of a set of factory methods for creating objects that are somehow related. The methods themselves are abstract and are specified to return abstract objects.

```

# Example: Abstract Factory

from abc import ABC, abstractmethod

class Button(ABC):
    # Abstract interface for buttons

    @abstractmethod
    def paint(self):
        raise NotImplementedError

class WinButton(Button):
    # Concrete product for Windows buttons

    def paint(self):
        print("WinButton")

class LinuxButton(Button):

    def paint(self):
        print("LinuxButton")

class Menu(ABC):
    # Abstract interface for menus

    @abstractmethod
    def paint(self):

```

```

        raise NotImplementedError

class WinMenu(Menu):
    # Concrete product for Windows menus

    def paint(self):
        print("WindowsMenu")

class LinuxMenu(Menu):
    # Concrete product for Linux menus

    def paint(self):
        print("LinuxMenu")

class GUIFactory(ABC):

    # Abstract factory that declares a set of methods for creating each of the
    # products. These methods must return abstract product types represented by
    # the abstract interfaces Button and Menu.

    @abstractmethod
    def create_button(self) -> Button:
        raise NotImplementedError

    @abstractmethod
    def create_menu(self) -> Menu:
        raise NotImplementedError

class WinFactory(GUIFactory):

    # The concrete factory for Windows product variants.

    def create_button(self):
        return WinButton()

    def create_menu(self):
        return WinMenu()

class LinuxFactory(GUIFactory):

    # The concrete factory for Linux product variants.

    def create_button(self):
        return LinuxButton()

    def create_menu(self):

```

```

        return LinuxMenu()

if __name__ == "__main__":

    os = input("Select OS: ")

    if os == "win":
        factory = WinFactory()

    elif os == "linux":
        factory = LinuxFactory()

    else:
        raise ValueError("Invalid GUI")

    # Create the GUI
    button = factory.create_button()
    menu = factory.create_menu()
    gui = [button, menu]

    # Paint the GUI
    for item in gui:
        item.paint()

```

4.1.3. Builder

The builder pattern is a creational pattern that is used to create complex objects. It provides a way to create an object step by step and allows you to produce different types and representations of an object using the same construction code.

In Python the builder pattern is usually implemented using a class with a ***fluent interface***. A fluent interface is an object-oriented API that aims to provide more readable code.

Typically it involves method chaining and replaces multi-parameter functions, such as complex constructors, with multiple methods that each take a single parameter.

```

# Complex constructor
person = Person(
    age = 40,
    name = "John Doe",
)

# Builder with fluent interface
person = Person().name("John Doe").age(40)

```

It is implemented by using method chaining to invoke multiple methods on the same object. The methods return the object itself which allows the calls to be chained together in a single statement.

```
# Example: Builder Pattern
```

```
from abc import ABC, abstractmethod
```

```
class Pizza(object):
```

```
    def __init__(self):
```

```
        self.crust = None
```

```
        self.cheese = None
```

```
        self.toppings = []
```

```
    def __str__(self):
```

```
        return "Crust: {0}\nCheese: {1}\nToppings: {2}".format(
```

```
            self.crust,
```

```
            self.cheese,
```

```
            ", ".join(self.toppings)
```

```
        )
```

```
class PizzaBuilderAbc(ABC):
```

```
    @abstractmethod
```

```
    def set_crust(self, crust):
```

```
        pass
```

```
    @abstractmethod
```

```
    def add_cheese(self, cheese):
```

```
        pass
```

```
    @abstractmethod
```

```
    def add_topping(self, topping):
```

```
        pass
```

```
    @abstractmethod
```

```
    def build(self):
```

```
        pass
```

```
class MyPizzaBuilder(object):
```

```
    def __init__(self):
```

```
        self.pizza = Pizza()
```

```
    def set_crust(self, crust):
```

```
        self.pizza.crust = crust
```

```
        return self
```

```
    def add_cheese(self, cheese):
```

```
        self.pizza.cheese = cheese
```

```

        return self

    def add_topping(self, topping):
        self.pizza.toppings.append(topping)
        return self

    def build(self):
        return self.pizza

pizza = (
    MyPizzaBuilder()
    .set_crust("thin")
    .add_cheese("mozzarella")
    .add_topping("pepperoni")
    .add_topping("mushrooms")
    .build()
)

print(pizza)

```

4.1.4. Prototype

The prototype pattern is a creational pattern that is used to create objects by cloning existing objects. The new object will have the same properties as the cloned object but will be a separate instance. It provides a way to hide the creation logic of an object.

In Python the prototype may be implemented using the `copy` module or the `copy()` and `deepcopy()` methods. The `copy()` method creates a shallow copy of an object. A **shallow copy** of an object is a copy of the object without cloning any of its internal references. The `deepcopy()` method creates a deep copy of an object. A **deep copy** of an object is a copy of the object that clones all of its internal references.

For simplicity the following example will not use the `copy` methods and instead will show the logic of the prototype pattern.

```

# Example: Prototype Pattern

from abc import ABC, abstractmethod

# Prototype interface
class Prototype(ABC):

    @abstractmethod
    def clone(self):
        pass

```

```

# Concrete Prototype
class Pizza(Prototype):

    def __init__(self, crust, cheese, toppings):
        self.crust = crust
        self.cheese = cheese
        self.toppings = toppings

    def __str__(self):
        return "Crust: {0}\nCheese: {1}\nToppings: {2}".format(
            self.crust,
            self.cheese,
            ", ".join(self.toppings)
        )

    def clone(self):
        return Pizza(self.crust, self.cheese, self.toppings)

if __name__ == "__main__":

    pizza = Pizza("thin", "mozzarella", ["pepperoni", "mushrooms"])

    pizza_clone = pizza.clone()
    print(pizza_clone)

    print(pizza == pizza_clone)
    print(pizza is pizza_clone)

```

4.1.5. Singleton

The singleton pattern is a creational pattern that is used to create a single instance of a class. It provides a way to hide the creation logic of an object.

In Python the singleton pattern can be implemented in many ways, for example by using the *new* method.

```

# Example: Singleton Pattern with __new__

class Singleton(object):

    __instance = None

    def __new__(cls):
        if cls.__instance is None:
            cls.__instance = super(Singleton, cls).__new__(cls)

        return cls.__instance

```



```
s1 = Singleton()
s2 = Singleton()

print(s1 is s2)
```

4.2. Structural Patterns

Structural patterns are a set of design patterns that deal with the composition of classes and objects. They are used to create larger structures from smaller ones. The structural patterns are:

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

4.2.1. Adapter

The adapter pattern is a structural pattern that is used to adapt the interface of a class to another interface. It provides a way to hide the incompatibility between two interfaces. The adapter pattern is also called the wrapper pattern.

There are two types of adapters:

- Class adapter
- Object adapter

The class adapter uses multiple inheritance to adapt one interface to another. It is implemented by creating a subclass of the target class and the adaptee class. The subclass inherits the target class and the adaptee class. It provides a way to adapt the interface of the adaptee class to the interface of the target class.

The object adapter uses composition to adapt one interface to another. It is implemented by creating a class that contains an instance of the adaptee class. It provides a way to adapt the interface of the adaptee class to the interface of the target class.

```
class ChannelV1(object):

    @staticmethod
    def applyConfig():
        # method name from actual legacy code in the company (Python 2.7)
        print('Configuration method of the old device class!')
```

```

class ChannelV2(object):

    @staticmethod
    def configure():
        print('Configuration method of the new device class!')

class ChannelAdapter(ChannelV1, ChannelV2):
    # Class adapter (with inheritance)

    def applyConfig(self):
        # The adapter's applyConfig method calls the new class's configure method
        self.configure()

def host_app(channel):
    # The host_app function works with the ChannelV1 interface
    channel.applyConfig()

if __name__ == "__main__":
    # Original code using the old service (ChannelV1)
    host_app(channel=ChannelV1())

    # Use adapter for the new service (ChannelV2) adapted to the old interface
    (ChannelV1)
    host_app(channel=ChannelAdapter())

```

Example: Adapter Pattern (Object Adapter)

```

class ChannelV1(object):

    @staticmethod
    def applyConfig():
        # method name from actual legacy code in the company
        print('Configuration method of the old device class!')

class ChannelV2(object):

    @staticmethod
    def configure():
        print('Configuration method of the new device class!')

class ChannelAdapter(ChannelV1):
    # Class adapter (with composition)

    def __init__(self, adaptee):

```

```

        self.adaptee = adaptee

    def applyConfig(self):
        self.adaptee.configure()

def host_app(channel):
    channel.applyConfig()

if __name__ == "__main__":
    # Original code using the old service
    host_app(channel=ChannelV1())

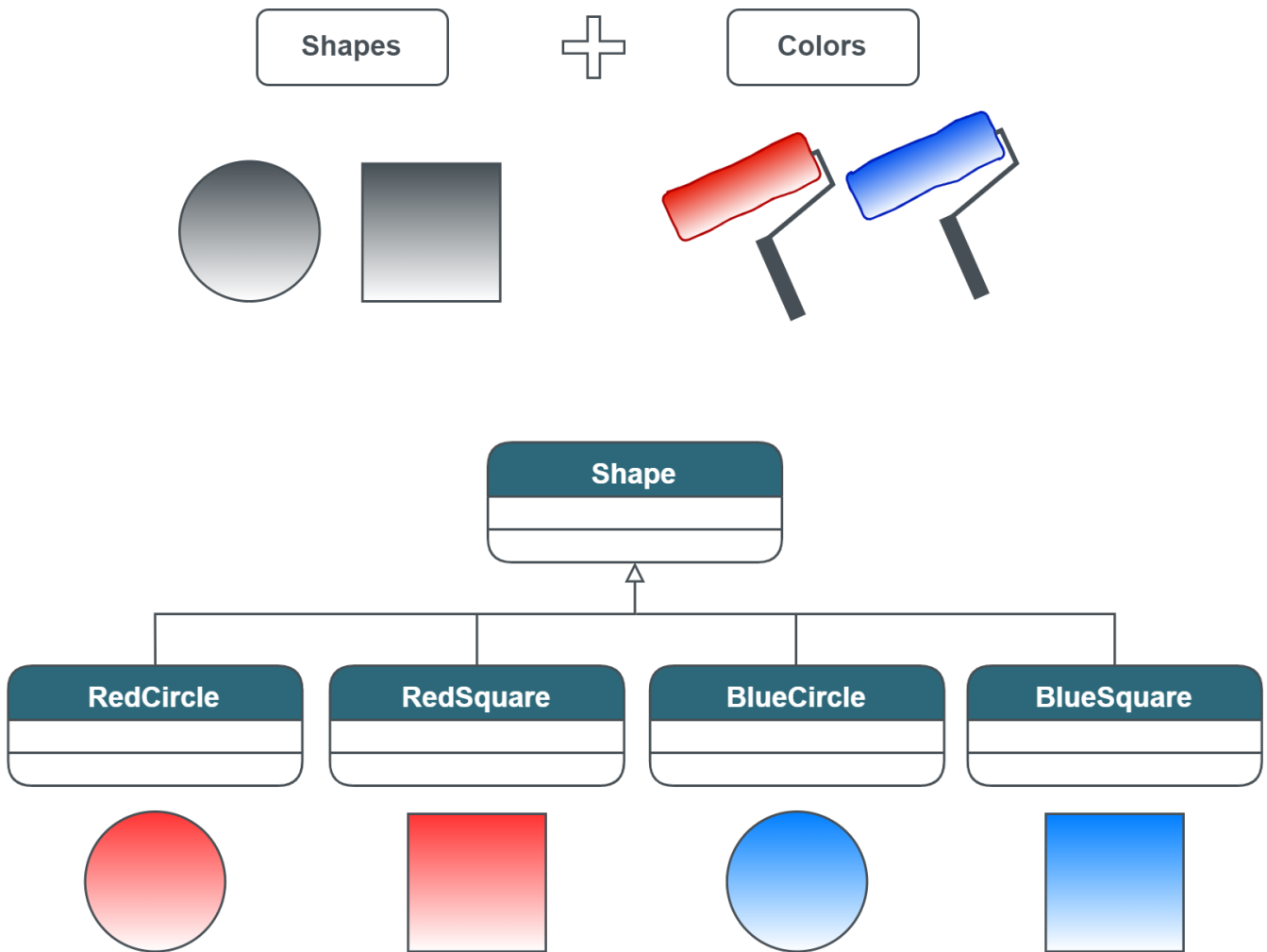
    # Use adapter for the new service adapted to the old interface
    host_app(channel=ChannelAdapter(adaptee=ChannelV2()))

```

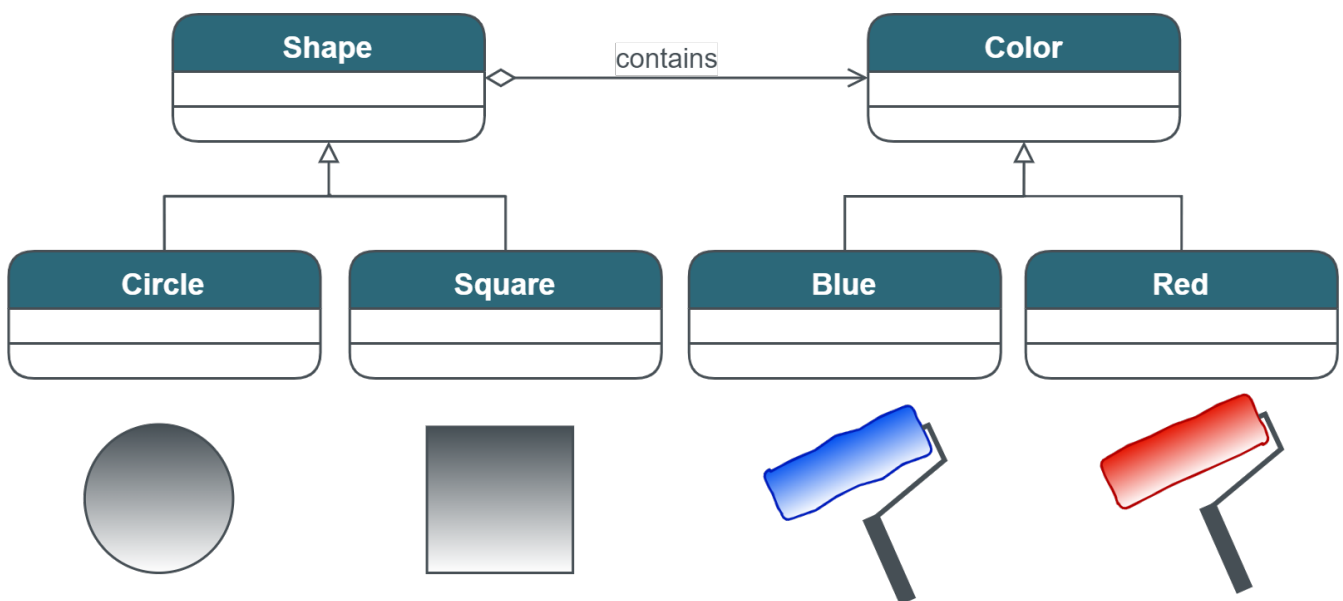
4.2.2. Bridge

The bridge pattern provides a way to hide the implementation details of a class from the clients using it. It is used to decouple an abstraction from its implementation so that the two can vary independently. It provides a way to separate the interface of a class from its implementation.

Let's say you have a class that can draw shapes with different colors. You can implement this class by using inheritance. You can create a subclass for each shape and color combination. For example, you can create a subclass for drawing a red circle, a red square, a blue circle, and a blue square. This approach works well if you have a small number of shapes and colors. However, it becomes difficult to manage when you have a large number of shapes and colors.



The bridge pattern can be used to solve this problem. It provides a way to separate the interface of a class from its implementation. It allows you to create a class that can draw shapes with different colors without having to create a subclass for each shape and color combination.



Example: Bridge Pattern with shapes and colors

```

class Color(object):
    # Interface for Implementation
  
```

```

def __init__(self, name):
    self.name = name

def paint(self, shape):
    raise NotImplementedError

class Red(Color):
    # Concrete implementation for red color

    def __init__(self):
        super(Red, self).__init__('red')

    def paint(self, shape):
        print('Painting the {} with red color'.format(shape))

class Blue(Color):
    # Concrete implementation for blue color

    def __init__(self):
        super(Blue, self).__init__('blue')

    def paint(self, shape):
        print('Painting the {} with blue color'.format(shape))

class Shape(object):
    # This is the abstraction used by the client

    def __init__(self, color: Color):
        self.color = color

    def draw(self):
        raise NotImplementedError

class Circle(Shape):
    # Refined abstraction for circles

    def draw(self):
        print('Drawing a circle')
        self.color.paint(shape='circle')

class Square(Shape):
    # Refined abstraction for squares

    def draw(self):
        print('Drawing a square')

```

```

        self.color.paint(shape='square')

class DrawApp(object):
    # This is the client class

    @staticmethod
    def draw(shape: Shape):
        shape.draw()

if __name__ == "__main__":

    # Draw a red circle
    app = DrawApp()

    # Draw shapes
    app.draw(shape=Circle(color=Red()))
    app.draw(shape=Square(color=Blue()))

```

4.2.3. Composite

The composite pattern is a structural pattern that is used to compose objects into tree structures to represent part-whole hierarchies. It provides a way to treat both individual objects and groups of objects in a uniform way. It is used to create a tree structure of objects where each node in the tree structure can be either an individual object or a group of objects.

```

# Example: Composite pattern

from abc import abstractmethod

class IComponent(object):
    # Interface used by both the leaf and composite classes

    def __init__(self, name):
        self.name = name

    @abstractmethod
    def execute(self):
        raise NotImplementedError

class Leaf(IComponent):

    def execute(self):
        return "{} running".format(self.name)

```

```

# Concrete implementation of the composite class
class Composite(IComponent):
    # This is the composite class (tree is composed of leafs)

    def __init__(self, name="Root"):
        super(Composite, self).__init__(name)
        self.components = []

    def add(self, c):
        self.components.append(c)
        return self

    def remove(self, c):
        self.components.remove(c)
        return self

    def get_children(self):
        return self.components

    def execute(self):

        # Delegate work to all the children
        for child in self.components:
            print("{} / {}".format(self.name, child.execute()))

        return "{} running".format(self.name)

if __name__ == "__main__":

    # Define composite product
    product = Composite()
    product.add(Leaf("Leaf 1"))
    product.add(Leaf("Leaf 2"))
    print(product.execute())

    print()

    # Nested composite product
    product.add(
        Composite("Subsystem A")
        .add(Leaf("Leaf 1"))
        .add(Leaf("Leaf 2"))
    )
    print(product.execute())

```

4.2.4. Decorator

The decorator pattern is a structural pattern that is used to add new functionality to an existing object without changing its structure. It provides a way to extend the functionality of an object at

runtime.

```
# Example: Decorator Pattern

from abc import ABC, abstractmethod

class NotifierAbc(ABC):

    @abstractmethod
    def notify(self):
        pass

class Notifier(NotifierAbc):

    def notify(self):
        print("Print on screen ...")

class NotifierDecorator(NotifierAbc):

    def __init__(self, component):
        self._component = component

    @abstractmethod
    def notify(self):
        pass

class EmailNotifier(NotifierDecorator):

    @staticmethod
    def send_email():
        print("Sending email ...")

    def notify(self):
        self.send_email()
        self._component.notify()

class SMSNotifier(NotifierDecorator):

    @staticmethod
    def send_sms():
        print("Sending SMS ...")

    def notify(self):
        self.send_sms()
        self._component.notify()
```



```
def main():
    notifier = Notifier()
    notifier_with_email = EmailNotifier(notifier)
    notifier_with_email_and_sms = SMSNotifier(notifier_with_email)
    notifier_with_email_and_sms.notify()

if __name__ == "__main__":
    main()
```

4.2.5. Facade

The facade pattern is a structural pattern that is used to create a simplified interface to a complex system. It provides a way to hide the complexity of a system and provide clients with a simple interface to it.

```
# Example: Facade Pattern

class PumpSystem(object):

    @staticmethod
    def prepare():
        print("SubsystemA prepare ...")

    @staticmethod
    def run():
        print("SubsystemA run ...")

class VentilationSystem(object):

    @staticmethod
    def prepare():
        print("SubsystemB prepare ...")

    @staticmethod
    def run():
        print("SubsystemB run ...")

class ComplexSystemFacade(object):

    def __init__(self):
        self._subsystemA = PumpSystem()
        self._subsystemB = VentilationSystem()

    def run(self):
```

```

        self._subsystemA.prepare()
        self._subsystemB.prepare()
        self._subsystemA.run()
        self._subsystemB.run()

def main():
    system = ComplexSystemFacade()
    system.run()

if __name__ == "__main__":
    main()

```

4.2.6. Flyweight

The Flyweight Pattern is a structural design pattern that's used to minimize memory usage or computational expenses by sharing as much as possible with related objects. It's especially useful when you have a large number of similar objects that can be made more efficient by sharing common state.

The shared state is immutable and can be shared between multiple objects. The non-shared state is mutable and is stored by the flyweight object. The non-shared state must be passed to the flyweight object by the client when it is needed.

Some typical use cases for the flyweight pattern are:

- Text editors
- Word processors
- eCommerce applications
- Graphics applications
- Computer games

```

# Exercise: Flyweight Pattern

# Shared state (flyweight object)
class CoffeeFlavor(object):

    def __init__(self, flavor):
        self._flavor = flavor

    def get_flavor(self):
        return self._flavor

# Unique state (context object)
class CoffeeOrder(object):

```

```

def __init__(self, table_number, flavor):
    self._table_number = table_number
    self._flavor = flavor

def serve(self):
    print(f"Serving coffee to table {self._table_number} with flavor {self._flavor.get_flavor()}")

# Flyweight factory (manages the flyweight and context objects)
class CoffeeShop(object):

    def __init__(self):

        # Cache for the orders and flavors
        self._orders = {}
        self._flavors = {}

    def take_order(self, table_number, flavor_name):

        # Check if the flavor instance is already cached
        flavor = self._flavors.get(flavor_name)

        # If not, create a new flavor instance and cache it
        if not flavor:
            flavor = CoffeeFlavor(flavor_name)
            self._flavors[flavor_name] = flavor

        # Create a new order with the shared flavor and unique table number
        order = CoffeeOrder(table_number, flavor)
        self._orders[table_number] = order

    def serve_orders(self):
        for table_number, order in self._orders.items():
            order.serve()

# Client code
if __name__ == "__main__":

    # Create the coffee shop
    coffee_shop = CoffeeShop()

    # Take orders from the customers
    coffee_shop.take_order(1, "Cappuccino")
    coffee_shop.take_order(2, "Espresso")
    coffee_shop.take_order(3, "Cappuccino")
    coffee_shop.take_order(4, "Espresso")

    # Serve the orders

```

4.2.7. Proxy

The proxy pattern is a structural pattern that is used to provide a surrogate or placeholder for another object to control access to it. It provides a way to add extra functionality to an object without changing its structure.

```
# Example: Proxy Pattern

class Server(object):

    def request(self):
        raise NotImplementedError

class RealServer(Server):

    def request(self):
        print("RealServer: Handling request.")

class ProxyServer(Server):

    def __init__(self, server: Server = None):
        self._server = server

    def request(self):
        if self.check_access():
            self._server.request()
            self.log_access()

    @staticmethod
    def check_access():
        print("ProxyServer: Checking access prior to firing a real request.")
        return True

    @staticmethod
    def log_access():
        print("ProxyServer: Logging the time of request.", end="")

class Client(object):

    def __init__(self, server: Server):
        self._server = server

    def execute(self):
        self._server.request()
```

```

if __name__ == "__main__":

    print("Client: Executing the client code with a real server")
    real_server = RealServer()
    client = Client(server=real_server)
    client.execute()

    print("")

    print("Client: Executing the same client code with a proxy server")
    client = Client(server=ProxyServer(real_server))
    client.execute()

```

4.3. Behavioral Patterns

Behavioral patterns are a set of design patterns that deal with the communication between classes and objects. They are used to define how objects interact with each other. The behavioral patterns are:

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor
- Interpreter

4.3.1. Chain of Responsibility

The Chain of Responsibility is a behavioral design pattern that allows you to pass requests along a chain of handlers. Each handler decides either to process the request or to pass it to the next handler in the chain. Here's a simple example of the Chain of Responsibility pattern in Python:

Let's say we have a purchase approval system where purchase requests are handled by different managers based on the amount. If the purchase amount is below a certain threshold, a team leader can approve it. If it's above the team leader's limit, it's passed to a manager, and so on.

When to use the chain of responsibility pattern:

- Execute multiple handlers in a specific order
- Change the set or the order of the handlers at runtime

```
# Example: Chain of Responsibility Pattern

# Handler interface

class Approver(object):
    def __init__(self, successor=None):
        self.successor = successor

    def set_successor(self, successor):
        self.successor = successor

    def approve(self, purchase):
        pass

# Concrete handlers

class TeamLeader(Approver):
    def approve(self, purchase):
        if purchase <= 1000:
            print(f"Team Leader approves the purchase of ${purchase}")
        elif self.successor:
            self.successor.approve(purchase)

class Manager(Approver):
    def approve(self, purchase):
        if 1000 < purchase <= 5000:
            print(f"Manager approves the purchase of ${purchase}")
        elif self.successor:
            self.successor.approve(purchase)

class Director(Approver):
    def approve(self, purchase):
        if 5000 < purchase <= 10000:
            print(f"Director approves the purchase of ${purchase}")
        elif self.successor:
            self.successor.approve(purchase)

class President(Approver):
    def approve(self, purchase):
        if purchase > 10000:
            print(f"President approves the purchase of ${purchase}")
```

```

        elif self.successor:
            self.successor.approve(purchase)

# Client code

if __name__ == "__main__":

    # Define the roles in the chain of responsibility
    team_leader = TeamLeader()
    manager = Manager()
    director = Director()
    president = President()

    # Set the successors
    team_leader.set_successor(manager)
    manager.set_successor(director)
    director.set_successor(president)

    # Create purchases
    purchases = [500, 1000, 2000, 5000, 10000, 20000]

    # Process the purchases
    for purchase in purchases:
        team_leader.approve(purchase)

```

4.3.2. Command

The command pattern is a behavioral pattern that is used to encapsulate a request as an object. It provides a way to decouple the sender of a request from its receiver by giving multiple objects a chance to handle the request.

When to use the command pattern:

- Parametrize objects with operations
- Specify, queue, and execute requests at different times
- Support reversible operations

```

# Example: Command Pattern

```

```

# Command interface
class Command(object):
    def execute(self):
        pass

    def undo(self):
        pass

```

```

# Concrete commands
class TypeCommand(Command):
    def __init__(self, text, document):
        self.text = text
        self.document = document

    def execute(self):
        self.document.add_text(self.text)

    def undo(self):
        self.document.remove_text(self.text)

# Receiver class
class Document(object):
    def __init__(self):
        self.content = ""

    def add_text(self, text):
        self.content += text

    def remove_text(self, text):
        if text and text in self.content:
            self.content = self.content.replace(text, "", 1)

    def get_content(self):
        return self.content

# Invoker class (TextEditor)
class TextEditor(object):
    def __init__(self):
        self.history = []

    def execute(self, command):
        command.execute()
        self.history.append(command)

    def undo_last_command(self):
        if self.history:
            command = self.history.pop()
            command.undo()

# Client code
if __name__ == "__main__":
    document = Document()

```



```

editor = TextEditor()

print("Initial content:", document.get_content())

# Type some text
type_action1 = TypeCommand("Hello, ", document)
editor.execute(type_action1)
print("After typing:", document.get_content())

type_action2 = TypeCommand("world!", document)
editor.execute(type_action2)
print("After typing:", document.get_content())

# Undo the last command
editor.undo_last_command()
print("After undo:", document.get_content())

```

4.3.3. Iterator

The Iterator Pattern is a behavioral design pattern that allows sequential access to the elements of an composite object without exposing its underlying representation. The composite object may be a list, a tree, a graph or any other complex data structure. The iterator pattern decouples the algorithm for traversing the data structure from the data structure itself.

Python offers a built-in iterator protocol that allows you to iterate over all the elements of a any composite object that implements the ***iter()*** method. The ***iter()*** method should return an iterator object that implements the ***next()*** method. The ***next()*** method should return the next element in the sequence (see chapter about dunder methods).

When to use the iterator pattern:

- Access the elements of a composite object sequentially without exposing its underlying representation
- Provide a uniform interface for traversing different aggregate structures

```

# Example: Iterator Pattern

# Iterator interface
class Iterator(object):
    def has_next(self):
        pass

    def next(self):
        pass

# TreeNode represents a node in a tree
class TreeNode(object):

```

```

def __init__(self, data):
    self.data = data
    self.children = []

def add_child(self, child):
    self.children.append(child)

# Concrete Iterator for tree traversal
class TreeIterator(Iterator):
    def __init__(self, root):
        self.stack = [root]

    def has_next(self):
        return len(self.stack) > 0

    def next(self):
        if not self.has_next():
            raise StopIteration()

        node = self.stack.pop()
        for child in reversed(node.children):
            self.stack.append(child)
        return node.data

# Client code
if __name__ == "__main__":

    # Create a sample tree structure
    root = TreeNode("Root")
    child1 = TreeNode("Child 1")
    child2 = TreeNode("Child 2")
    child3 = TreeNode("Child 3")

    # Add nodes to the tree
    root.add_child(child1)
    root.add_child(child2)
    child2.add_child(child3)

    # Create a tree iterator
    iterator = TreeIterator(root)

    # Traverse and print tree nodes
    while iterator.has_next():
        node = iterator.next()
        print(node)

```

4.3.4. Mediator

Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

When to use the mediator pattern:

- There are tight interconnections between several components
- Use when it gets hard to maintain relationships between multiple objects
- A component becomes hard to reuse in a different context

```
# Example: Mediator Pattern

# Mediator interface
class Mediator(object):

    def forward(self, message, component):
        raise NotImplementedError

# Concrete Mediator
class Dialog(Mediator):

    def __init__(self):
        self.components = []

    def set_component(self, component):
        self.components.append(component)

    def forward(self, message, sender):
        for component in self.components:
            if sender is not component:
                component.receive(message)

# Component interface
class Component(object):

    def __init__(self, mediator):
        self.mediator = mediator

    def send(self, message):
        raise NotImplementedError

    def receive(self, message):
        raise NotImplementedError
```

```

# Concrete Component
class Button(Component):

    def send(self, message):
        print(f"Button sends: {message}")
        self.mediator.forward(message, self)

    def receive(self, message):
        print(f"Button receives: {message}")

# Concrete Component
class Textbox(Component):

    def send(self, message):
        print(f"Textbox sends: {message}")
        self.mediator.forward(message, self)

    def receive(self, message):
        print(f"Textbox receives: {message}")

# Concrete Component
class Checkbox(Component):

    def send(self, message):
        print(f"Checkbox sends: {message}")
        self.mediator.forward(message, self)

    def receive(self, message):
        print(f"Checkbox receives: {message}")

# Client code
if __name__ == "__main__":

    # Create the mediator
    mediator = Dialog()

    # Add components to the mediator
    button = Button(mediator)
    mediator.set_component(button)

    # Add more components to the mediator
    textbox = Textbox(mediator)
    mediator.set_component(textbox)

    # Add more components to the mediator
    checkbox = Checkbox(mediator)
    mediator.set_component(checkbox)

```

```
# Send messages
button.send("Hello from Button!")
```

4.3.5. Memento

The Memento Pattern is a behavioral design pattern that allows you to save and restore the previous state of an object without revealing the details of its implementation. It provides a way to save the internal state of an object without violating encapsulation. The pattern is also known as the snapshot pattern.

When to use the memento pattern:

- Produce snapshots of an object's state to be able to restore it later
- Use when direct access to the object's fields/getters/setters violates its encapsulation
- Use in combination with the command pattern to implement undoable operations (save, execute, restore)

```
# Example: Memento Pattern

# Originator class
class TextEditor:
    def __init__(self):
        self.content = ""

    def write(self, text):
        self.content += text

    def show_content(self):
        print(f"Editor Content: {self.content}")

    def create_memento(self):
        return TextEditorMemento(self.content)

    def restore_from_memento(self, memento):
        self.content = memento.get_state()

# Memento class
class TextEditorMemento:
    def __init__(self, content):
        self.content = content

    def get_state(self):
        return self.content

# Caretaker class
class History:
    def __init__(self):
```

```

        self.states = []

    def push(self, memento):
        self.states.append(memento)

    def pop(self):
        if self.states:
            return self.states.pop()
        return None

# Client code
if __name__ == "__main__":

    text_editor = TextEditor()
    history = History()

    text_editor.write("Hello, ")
    history.push(text_editor.create_memento())

    text_editor.write("world!")
    history.push(text_editor.create_memento())

    text_editor.show_content()

    text_editor.write(" Hello, hello!")
    text_editor.show_content()

    # Undo the last action
    last_state = history.pop()
    if last_state:
        text_editor.restore_from_memento(last_state)

    text_editor.show_content()

```

4.3.6. Observer

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing. The producer of the events is called the **Subject**. The object that consumes the events is called the **Observer**. The subject and the observer are loosely coupled.

When to use the observer pattern:

- Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.
- Use the pattern when some objects in your app must observe others, but only for a limited time or in specific cases.

```

# Example: Observer Pattern

# Subject Interface (produces data)
class Publisher(object):

    def attach(self, observer):
        pass

    def detach(self, observer):
        pass

    def notify(self):
        pass

# Concrete Subject
class WeatherData(Publisher):

    def __init__(self):
        self.subscribers = []
        self.temperature = 0.0
        self.humidity = 0.0
        self.pressure = 0.0

    def attach(self, observer):
        self.subscribers.append(observer)

    def detach(self, observer):
        self.subscribers.remove(observer)

    def notify(self):
        for subscriber in self.subscribers:
            subscriber.update(self.temperature, self.humidity, self.pressure)

    def set(self, temperature, humidity, pressure):
        self.temperature = temperature
        self.humidity = humidity
        self.pressure = pressure
        self.notify()

# Observer interface (consumes data)
class Subscriber(object):

    def update(self, temperature, humidity, pressure):
        pass

# Concrete Observer
class DisplayDevice(Subscriber):

```

```

def __init__(self, name):
    self.name = name

def update(self, temperature, humidity, pressure):
    print(
        f"{self.name} Display - Temperature: {temperature}°C, Humidity:
{humidity}%, Pressure: {pressure} hPa")

# Client code
if __name__ == "__main__":

    # Create a weather station
    weather_station = WeatherData()

    # Attach a display device to the weather station
    display1 = DisplayDevice("Display 1")
    weather_station.attach(display1)

    # Attach another display device to the weather station
    display2 = DisplayDevice("Display 2")
    weather_station.attach(display2)

    # Simulate changes in weather data
    weather_station.set(25.0, 60.0, 1013.0)
    weather_station.set(26.5, 55.0, 1010.5)

```

4.3.7. State

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class. The pattern encapsulates the states into separate classes and delegates the state transitions to the objects representing the states.

When to use the state pattern:

- Use the State pattern when you have an object that behaves differently depending on its current state, the number of states is enormous, and the state-specific code changes frequently.
- Use the pattern when you have a class polluted with massive conditionals that alter how the class should behave according to the current values of the class's fields.
- Use the pattern when you have a lot of duplicate code across similar states and transitions of a condition-based state machine.

[finite state machine] | [part_b/ch_04/finite_state_machine.png](#)

```

# State interface
class State(object):

    def connect(self):

```



```

        raise NotImplementedError

    def disconnect(self):
        raise NotImplementedError

    def send_data(self, data):
        raise NotImplementedError

# Concrete State: Disconnected
class DisconnectedState(State):

    def connect(self):

        # Action
        print("Connecting to the server...")

        # Transition to the Connected state
        return ConnectedState()

    def disconnect(self):
        print("Already disconnected.")
        return self

    def send_data(self, data):

        print("Cannot send data while disconnected.")
        return self

# Concrete State: Connected
class ConnectedState(State):

    def connect(self):

        # Action
        print("Already connected.")

        # Transition
        return self

    def disconnect(self):
        # Action
        print("Disconnecting from the server...")

        # Transition
        return DisconnectedState()

    def send_data(self, data):
        # Action
        print(f"Sending data to the server: {data}")

```

```

        # Transition
        return self

# Context class
class Client(object):

    def __init__(self):
        # Initial state
        self.state = DisconnectedState()

    def connect(self):
        self.state = self.state.connect()

    def disconnect(self):
        self.state = self.state.disconnect()

    def send_data(self, data):
        self.state.send_data(data)

# Client code
if __name__ == "__main__":

    client = Client()

    client.send_data("Hello, server!") # Try sending data while disconnected

    client.connect()
    client.send_data("Hello, server!") # Send data after connecting

    client.connect() # Try connecting again

    client.disconnect()
    client.send_data("Goodbye, server!") # Send data after disconnecting

```

4.3.8. Strategy

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

When to use the strategy pattern:

- Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.
- Use the pattern when your class has a massive conditional operator that switches between different variants of the same algorithm.
- Use the pattern when you have a lot of similar classes that only differ in the way they execute

some behavior.

```
# Example: Strategy Pattern

# Strategy interface
class TextFormatter(object):

    def format_text(self, text):
        raise NotImplementedError

# Concrete Strategy: Uppercase formatting
class UppercaseFormatter(TextFormatter):

    def format_text(self, text):
        return text.upper()

# Concrete Strategy: Lowercase formatting
class LowercaseFormatter(TextFormatter):
    def format_text(self, text):
        return text.lower()

# Concrete Strategy: Title case formatting
class TitleCaseFormatter(TextFormatter):

    def format_text(self, text):
        return text.title()

# Context class
class TextEditor(object):
    def __init__(self, formatter):
        self.formatter = formatter

    def set_formatter(self, formatter):
        self.formatter = formatter

    def format_text(self, text):
        return self.formatter.format_text(text)

# Client code
if __name__ == "__main__":

    text = "This is a simple example of the Strategy Pattern."

    # Create text editor with the default uppercase formatting strategy
    editor = TextEditor(UppercaseFormatter())
```

```

result = editor.format_text(text)
print("Uppercase Formatting:")
print(result)

# Change the formatting strategy to lowercase
editor.set_formatter(LowercaseFormatter())
result = editor.format_text(text)
print("\nLowercase Formatting:")
print(result)

# Change the formatting strategy to title case
editor.set_formatter(TitleCaseFormatter())
result = editor.format_text(text)
print("\nTitle Case Formatting:")
print(result)

```

4.3.9. Template Method

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

When to use the template method pattern:

- Use the Template Method pattern when you want to let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure.
- Use the pattern when you have several classes that contain almost identical algorithms with some minor differences. As a result, you might need to modify both classes when the algorithm changes.

```

# Example: Template Method Pattern

# Abstract with template methods
class Notification(object):

    # Template method
    def send_notification(self, message):
        self.authenticate()
        self.format_message(message)
        self.send_message()

    # Template method
    def authenticate(self):
        print("Authentication successful")

    # Template method
    def format_message(self, message):
        print(f"Formatting message: {message}")

```

```

# Abstract method
def send_message(self):
    raise NotImplementedError

# Concrete Notification subclass for email
class EmailNotification(Notification):

    def send_message(self):
        print("Sending email...")

# Concrete Notification subclass for SMS
class SMSNotification(Notification):

    def send_message(self):
        print("Sending SMS...")

# Client code
if __name__ == "__main__":
    email_notification = EmailNotification()
    sms_notification = SMSNotification()

    text = "This is a notification message."

    print("Email Notification:")
    email_notification.send_notification(text)

    print("\nSMS Notification:")
    sms_notification.send_notification(text)

```

4.3.10. Visitor

The Visitor Pattern is a behavioral design pattern that allows you to separate algorithms from the objects on which they operate. It provides a way to add new operations to existing classes without modifying them. This pattern is especially useful when you have a complex structure of objects and want to perform various operations on them without modifying their source code.

When to use the visitor pattern:

- Use the Visitor pattern when you need to perform an operation on all elements of a complex object structure (for example, an object tree).
- Use the Visitor to clean up the business logic of auxiliary behaviors.
- Use the pattern when a behavior makes sense only in some classes of a class hierarchy, but not in others.

```
# Example: Visitor Pattern
```

```

class ExportVisitor(object):

    def visit(self, element):
        pass

class XMLExportVisitor(ExportVisitor):

    def visit(self, element):
        print('XML exporter visiting element of type
{0}'.format(type(element).__name__))

class Node(object):

    def accept(self, visitor):
        pass

class City(Node):

    def accept(self, visitor):
        visitor.visit(self)

class Industry(Node):

    def accept(self, visitor):
        visitor.visit(self)

class NavigationMap(object):

    def __init__(self):
        self.nodes = []

    def add(self, element):
        self.nodes.append(element)

    def accept(self, visitor):
        for element in self.nodes:
            element.accept(visitor)

if __name__ == "__main__":
    exporter = XMLExportVisitor()
    object_structure = NavigationMap()
    object_structure.add(City())
    object_structure.add(Industry())

```

4.3.11. Interpreter

The Interpreter Pattern is a behavioral design pattern that defines a language grammar and provides an interpreter to interpret sentences in that language. It is often used when you have a domain-specific language (DSL) and you want to implement an interpreter to execute statements in that language.

When to use the interpreter pattern:

- Define a grammar for a simple language
- Implement an interpreter for this language

```
# Example: Interpreter Pattern

# Abstract Expression
class Expression(object):

    def interpret(self, context):
        raise NotImplementedError

# Terminal Expression
class Number(Expression):

    def __init__(self, value):
        self.value = value

    def interpret(self, context):
        return self.value

# Non-terminal Expression
class Add(Expression):

    def __init__(self, left, right):
        self.left = left
        self.right = right

    def interpret(self, context):
        return self.left.interpret(context) + self.right.interpret(context)

# Context
class Context(object):

    def __init__(self):
        self.variables = {}
```

```
def set(self, variable, value):
    self.variables[variable] = value

def get(self, variable):
    return self.variables.get(variable, 0)

# Client code
if __name__ == "__main__":

    context = Context()
    context.set("x", 10)
    context.set("y", 5)

    expression = Add(
        Number(context.get("x")),
        Number(context.get("y"))
    )

    result = expression.interpret(context)
    print(f"Result: {result}")
```


Chapter 5. System Architecture

The system architecture defines the framework on how to manage the software development process and organize the software system. It provides a set of guidelines and best practices for building and maintaining the software system. This helps to minimize the overall costs and risks of the software development process.

The System Architecture defines the framework on how to manage the software development process and organize the software system. It provides a set of guidelines and best practices for building and maintaining the software system.

Here is a typical set of architectural artifacts that are used to define the software architecture of a software system:

1. Development Environment (e.g version control, issue tracking, CI/CD, IDE, etc.)
2. Deployment Strategy (e.g. Standalone, Cloud, etc.)
3. Functional Requirements (Use Cases, User Stories, Components, Class Diagrams, etc.)
4. Non-Functional Requirements (e.g. Performance, Scalability, etc.)
5. Architectural Style (e.g. Monolithic, Microservices, etc.)
 - Development Process (e.g. Waterfall, Agile, etc.)
6. Quality Assurance (Test Automation)

5.1. Development Environment

The development infrastructure is the set of tools and processes that are used to develop and maintain the software system. For example a web application may use the following tools and processes:

- Technologies (e.g. Python, Django, PostgreSQL, Redis, etc.)
- Integrated Development Environment (e.g. PyCharm, Eclipse, Github Codespaces)
- Coding standards (e.g. PEP 8, Google Python Style Guide)
- Version control (e.g. Git, SVN, Mercurial)
- Release system (e.g. semantic versioning)
- Issue tracking (e.g. Jira, Github Issues)

5.2. Deployment Strategy

The deployment and infrastructure are the processes and tools that are used to deploy and run the software system. This could also define the infrastructure requirements such as the operating system, the database, the web server, etc. For example a web application may use the following deployment infrastructure:

- as a standalone application on a server or a virtual machine.
- as a container in a container orchestration platform such as Kubernetes.
- on a cloud platform such as AWS, Azure, or Google Cloud.
- on a serverless platform such as AWS Lambda or Google Cloud Functions.
- continuous integration and continuous deployment (CI/CD) tools

5.3. Functional Requirements

The functional requirements define the functions and features of the software system. For example a web application may have the following functional requirements:

- User management (e.g. login, logout, register, forgot password)
- User profile management (e.g. update profile, change password)
- Product management (e.g. list products, add product to cart, checkout)
- Order management (e.g. list orders, cancel order, return order)

5.4. Non-Functional Requirements

The quality attributes are typically the non-functional requirements of a software system. They define the quality characteristics of the software system. For example a web application may have the following quality attributes:

- **Performance:** the responsiveness of the system under various workloads and data volumes (e.g. response time, throughput, latency, resource usage)
- **Scalability:** the ability of the system to handle increasing workloads and data volumes. Scaling can be done by adding more resources such as CPU or memory (vertical scaling) or by adding more instances of the system (horizontal scaling).
- **Security:** the ability of the system to protect data and resources from unauthorized access (e.g. authentication, authorization, encryption, auditing)
- **Reliability:** the ability of the system to perform its functions under abnormal conditions (e.g. error handling, fault tolerance, failure recovery)
- **Availability:** the ability of the system to be operational and accessible when needed (e.g. uptime, downtime, mean time between failures)
- **Maintainability:** the ability of the system to be modified and extended (e.g. loose coupling, modular design, technical documentation)
- **Usability:** the ability of the system to be used by users (e.g. easy of use, learning curve, user documentation)
- **Testability:** the ability of the system to be tested frequently and automatically (e.g. with a regression test suite that can be run after each software change)
- **Compatibility:** the ability of the system to be compatible with other systems (e.g. backward compatibility, forward compatibility)

- **Compliance:** the ability of the system to comply with laws, regulations, and standards (e.g. GDPR, PCI DSS, HIPAA, ISO 27001)

5.5. Architecture and Design

The architectural patterns are recurring styles that are used to structure a software system. For example a web application may use the MVC pattern to structure the frontend, and the microservices pattern to structure the backend of the application.

5.5.1. Architecture Styles

Multi-Layered

A multi-layered pattern can be used to structure programs that can be decomposed into groups of subtasks, each of which is at a particular level of abstraction. Each layer provides services to the next higher layer.

Typically the layers are organized in a hierarchical manner, with each layer providing services to the next higher layer. Almost all software applications are multi-layered. The most common layers are:

- Presentation layer (the frontend layer)
- Application layer (the backend layer)
- Data layer (the data access layer)

Protocols are also a form of multi-layered architecture. The OSI model is a seven-layered model that describes the different layers of a network protocol stack. The TCP/IP model is a four-layered model that describes the different layers of the Internet protocol suite.

Monolithic

A monolithic architecture is a traditional software architecture where all components and functionalities of an application are tightly integrated into a single codebase and executed as a single process.

The process may be further divided into multiple threads. The monolithic architecture is the traditional architecture used by most software applications. It is the opposite of a distributed architecture. It is also called a monolithic application or a monolithic kernel.

One of the main advantages of a monolithic architecture is that it is simple to develop, test and deploy. However, it is difficult to scale and maintain. It is also difficult to adopt new technologies and programming languages. This means that all of the components must be written in the same programming language.

Microservices

A service-oriented architecture can be used to structure programs that can be decomposed into a set of services that are independently deployable and scalable. Each service is self-contained and implements a single business capability. This allows each service to be developed, tested, deployed,

scaled and maintained independently, without affecting the other services. Each service can be implemented using a different programming language and technology stack.

Services communicate with each other using a protocol such as HTTP/HTTPS or gRPC. Services can be deployed as containers in a container orchestration platform such as Kubernetes. Microservices are typically implemented using a framework such as Spring Boot.

Client-Server

This pattern can be used to structure programs that can be decomposed into two parts: a server that provides a service, and a client that requests services from the server. The client and server communicate with each other using a dedicated protocol. The client and server may be implemented using different programming languages and technology stacks.

Peer-to-Peer

This pattern can be used to structure programs that can be decomposed into a set of peers that are symmetric in terms of their functionality, and each peer may act as both a client and a server. Blockchains or torrents are examples of peer-to-peer networks.

Pipe-Filter

This pattern can be used to structure programs that can be decomposed into a set of filters that process a stream of data, and a set of pipes that connect the filters together. Each filter has a single input and a single output. Pipes can be used to connect the filters together and to connect a filter to a data source or data sink. This allows the filters to be developed, tested, deployed, scaled and maintained independently, without affecting each other.

Message Queue

- Asynchronous point-to-point communication
- Only one consumer can process the message (whoever consumes it first)
- Allows the consumer to process the messages at its own pace (decoupling)
- The data is ordered (FIFO) and stored until it is processed (persistence)
- Not scalable for large number of consumers (e.g. millions of users)
- Useful for spike protection (e.g. when the producer is faster than the consumer)

Publish-Subscribe

- Asynchronous broadcast communication
- A client must subscribe to a topic to receive messages from it
- Publishers send messages to an intermediary called a message broker
- The message broker forwards the messages to the subscribers instantly
- Ordering and persistence are not guaranteed
- Better scalability than message queues (e.g. millions of users)

Message Stream

Similar to the publish-subscribe pattern with the following differences:

- The messages are published to a topic
- The messages are stored in a log
- The messages are delivered to the subscribers in the same order as they were published
- The subscribers can consume the messages at their own pace
- The subscribers can rewind the log and reprocess the messages
- Suitable for real-time data processing (e.g. analytics, monitoring, etc.)

5.5.2. System Design

- API Design
- Data Model
- UML Diagrams
- Wireframes (Mockup)

5.6. Development Process

The development process is the set of activities that are used to develop and maintain the software system. For example almost all software applications undergo the following development steps:

- Requirements analysis
- Design
- Implementation
- Testing
- Deployment
- Maintenance

These steps are also known as the software development life cycle (SDLC). The SDLC is a process used by software development teams to design, develop, test, and deploy software applications.

Depening on how the steps of the development process are performed, the development process may be classified as follows:

- **Sequential development** is a process where the steps of the development process are performed strictly one after the other.
- **Iterative development** is a process where the system is typically developed in multiple iterations. Each iteration involves the steps of the development process performed sequentially, builds on the previous iteration and refines the product until it meets the customer requirements. It is also know as the horizontal development process and it is the preferred process for monolithich and high-risk applications.

- **Incremental development** is a process where each component is developed independently and then integrated into the system. The component is divided into multiple **slices**, called also **increments**. Each new increment builds on top of the existing released functionality. It is also known as the vertical development process and it is the preferred process for distributed or applications with strict time constraints.

Depending on how the steps of the development process are performed, the development process may be classified as follows:

- Waterfall Model (sequential)
- V-Model Model (sequential)
- Prototyping Model (iterative)
- Spiral Model (iterative)
- Agile Model (iterative, incremental)

5.7. Quality Assurance

This section describes the quality assurance process. It involves the practice of verifying that the software meets its **functional** and **non-functional** requirements. It is typically done by writing automated tests that can be run frequently and automatically.

The quality assurance process is typically implemented by the quality assurance team. It involves the following activities:

- Testing strategy selection
- Testing tools selection
- Documentation

5.7.1. Testing Strategy

The testing strategy defines the approach that will be used to test the software system and typically will require a tight communication between the development and quality assurance teams.

- What to test? (e.g. unit, integration, end-to-end, etc.)
- How to test? (e.g. manual, automated, etc.)
- When to test? (e.g. continuous, discrete, etc.)
- Who will test? (e.g. QA engineers, developers, etc.)

5.7.2. Testing Tools

After the testing strategy is defined, the testing tools can be selected. The testing tools are the tools that are used to implement the testing strategy. They can be divided into the following categories:

- Test frameworks (e.g. pytest, unittest, nose, etc.)
- Test coverage tools (e.g. coverage.py, pytest-cov, etc.)

- Test reporting tools (e.g. pytest-html, pytest-json, etc.)

5.7.3. Documentation

The documentation is an important part of the quality assurance process. It involves the practice of writing documentation that describes the testing strategy and the testing tools. It is typically done by the quality assurance team. It involves the following activities:

- Test plan
- Test cases
- Test reports

Chapter 6. Logging

Logging is a crucial concept in software development and system administration. It involves the practice of recording, tracking, and storing events, actions, and messages generated by a software application or system for various purposes, including troubleshooting, monitoring, and auditing. Here are the key aspects of logging:

- Log levels (debug, info, warning, error, critical)
- Log formatting (with context, e.g thread, timestamp, etc.)
- Log output (console, file, network, etc.)
- Thread safety

6.1. Logging Hierarchy

The logging module provides a set of classes that can be used to log messages. The `Logger` class is the main class that is used to log messages. It provides methods for logging messages at the different log levels. The `LoggerAdapter` class is used to add contextual information to log messages.

The logging is implemented using a hierarchy of loggers. Each logger has a name and a level. At the top of the hierarchy is the root logger. The **root logger** is implemented as a singleton and all loggers inherit their properties from the root logger.

The basic configuration is done using the `basicConfig()` function. It is used to configure the root logger. It should be called before any other logging calls are made. It is typically called in the main module of an application.

The user has access to the logging hierarchy through the class variable `manager` that is an instance of the `Manager()` class. The `Manager()` class is a singleton that provides access to the logging hierarchy and shows all the children of the root logger. It also provides methods for creating and retrieving loggers.

```
# Example: Logging Objects with hierarchy

import logging

# Basic logging configuration
logging.basicConfig(level=logging.DEBUG)

# Create the root logger (no name is provided)
root = logging.getLogger()
root.setLevel(logging.DEBUG)
root.debug('DEBUG message from the root logger')

# Create a child logger (name is provided)
child = logging.getLogger('child')
child.info('INFO message from the child logger')
```



```
# Create a grandchild logger (parent and name are provided)
grandchild = logging.getLogger('child.grandchild')
grandchild.error('CRITICAL message from the grandchild logger')

# Get the logging hierarchy
root.info('Logger hierarchy: {}'.format(root.manager.loggerDict))
```

6.2. Logging Levels

The logging levels are organized in a hierarchy. The NOTSET level is the lowest and the logger will log message levels above. The CRITICAL level is the highest and the logger will log only critical messages as it is the highest level. The following table shows the logging levels with their names and numeric values:

- NOTSET (0)
- DEBUG (10)
- INFO (20)
- WARNING (30)
- ERROR (40)
- CRITICAL (50)

```
# Example: Logging Levels

import logging

# Basic logging configuration
logging.basicConfig(level=logging.DEBUG)

# Logging levels
logging.debug('Hello world!')
logging.info('Hello world!')
logging.warning('Hello world!')
logging.error('Hello world!')
logging.critical('Hello world!')
```

6.3. Basic Configuration

The root logger is configured using the `basicConfig()` function. It should be called before any other logging calls are made. It is typically called in the main module of an application. All loggers inherit their properties from the root logger. Once created the root logger cannot be reconfigured.

The `basicConfig()` function accepts the following parameters:

- `level`: The logging level of the root logger. The default level is WARNING.
- `format`: The format string of the log messages. The default format is

"%(levelname)s:%(name)s:%(message)s".

- **datefmt**: The format string of the date and time. The default format is "%Y-%m-%d %H:%M:%S".
- **filename**: The name of the log file. The default is None.
- **filemode**: The mode of the log file. The default is "a".
- **stream**: The stream to which the log messages are written. The default is sys.stderr.

The following example shows how to configure the root logger using the `basicConfig()` function:

```
# Example: Basic and Advanced Logging Configuration

import logging

# Root logger configuration
logging.basicConfig(level=logging.DEBUG)

# Create the root logger (no name is provided)
root = logging.getLogger()
root.info('DEBUG message from the root logger')

# Create a child logger (name is provided)
child = logging.getLogger('child')
child.info('INFO message from the child logger')
```

6.4. Advanced Configuration

The following section discusses the advanced configuration of the logging module. It introduces the concepts of loggers, handlers, filters, and formatters. It also discusses the best practices for configuring the logging module.

6.4.1. Logging Format

The logging format is a string that is used to format the log messages. The logging module provides a set of built-in format strings that can be used to format the log messages. The following table shows the built-in format strings:

Format	Description
%(asctime)s	The date and time of the log message. The default format is "%Y-%m-%d %H:%M:%S".
%(created)f	The timestamp of the log message. It is the number of seconds since the epoch.
%(filename)s	The filename of the source file where the logging call was made.
%(funcName)s	The name of the function where the logging call was made.

Format	Description
%(levelname)s	The logging level of the log message.
%(levelno)s	The numeric logging level of the log message.
%(lineno)d	The line number of the source file where the logging call was made.
%(message)s	The log message.
%(module)s	The name of the module where the logging call was made.
%(name)s	The name of the logger where the logging call was made.
%(pathname)s	The full pathname of the source file where the logging call was made.
%(process)d	The process ID of the process where the logging call was made.
%(processName)s	The name of the process where the logging call was made.
%(relativeCreated)d	The number of milliseconds since the logging module was loaded.
%(thread)d	The thread ID of the thread where the logging call was made.
%(threadName)s	The name of the thread where the logging call was made.

For example the root logger can be configured to use the following format string:

```
# Configure the root logger
logging.basicConfig(
    level = logging.DEBUG,
    format = "%(asctime)s : %(name)s : %(levelname)s - %(message)s",
)
```

6.4.2. Logging Handlers

The logging handlers are used to send log messages to different destinations. The logging module provides a set of built-in handlers that can be used to send log messages to the console, a file, the network, etc. Some of the most common handlers are:

Handler	Description
NullHandler	The NullHandler is used to disable logging in a library. It is used to prevent the "No handlers could be found for logger" warning message from being displayed when a library is used in an application that does not use logging.
StreamHandler	The StreamHandler is used to send log messages to the console. It is used to send log messages to the console when the application is run in interactive mode. It is the default handler of the root logger.
FileHandler	The FileHandler is used to send log messages to a file. It is used to send log messages to a file when the application is run in non-interactive mode. It is the default handler of the root logger.
RotatingFileHandler	The RotatingFileHandler is used to send log messages to a file that is rotated when it reaches a certain size. It is useful when you want to limit the size of the log file.
TimedRotatingFileHandler	The TimedRotatingFileHandler is used to send log messages to a file that is rotated at certain intervals. It is useful when you want to limit the size of the log file and rotate it at certain intervals.
SocketHandler	The SocketHandler is used to send log messages to a network socket. It is used to send log messages to a remote server over the network using TCP.
DatagramHandler	The DatagramHandler is used to send log messages to a UDP socket. It is used to send log messages to a remote server over the network using UDP.
SysLogHandler	The SysLogHandler is used to send log messages to the system log. It is used to send log messages to the system log on Unix systems. It is used to send log messages to the Windows event log on Windows systems.
SMTPHandler	The SMTPHandler is used to send log messages to an email address. It is used to send log messages to an email address using SMTP. It is useful when you want to send log messages to an email address when an error occurs.

Handler	Description
HTTPHandler	The HTTPHandler is used to send log messages to a web server using HTTP POST requests. It is useful when you want to send log messages to a web server when an error occurs.

6.4.3. Logging Exceptions

When an exception occurs it is a good practice to log some additional information about the error and provide a suggestion on how to resolve it.

```
import logging

a = 5
b = 0

try:
    c = a / b
except Exception as e:
    logging.error(e, exc_info=True)
    logging.error("Please check the values of a and b")
```

6.4.4. Configuration File

Typically the logging module is configured using the provided API. However, it can also be configured using a configuration file.

The configuration file **MUST** follow a specific format. It is divided into sections, with each section corresponding to one of the following classes:

- logger
- handler
- formatter
- filter

Each section has a name that is enclosed in square brackets. The name of the section is the name of the class. The name of the section is used to create an instance of the class.

Each section contains a set of key-value pairs. The key is the name of the attribute of the class. The value is the value of the attribute. The value can be a string, an integer, a float, a boolean, a list, or a dictionary.

The configuration file is useful when you want to configure the logging module without having to modify the source code of the application.

```
# Example: Configure the logging module with a configuration file
```

```

import logging.config

# Configure the logging module with a configuration file
logging.config.fileConfig('logging.conf')

# Get the root logger instance
root = logging.getLogger()

# Check the logger dictionary
root.info(f"After fileConfig(): {logging.Logger.manager.loggerDict} ")

# Get the test logger instance
test = logging.getLogger('test')

# Check the logger dictionary
root.info(f"After getLogger('test'): {logging.Logger.manager.loggerDict} ")

# Log a message from the test logger
test.info('INFO message from the test logger')

```

6.4.5. Configuration Dictionary

The logging module can also be configured using a configuration dictionary. The configuration dictionary **MUST** follow a specific format. It is divided into sections, with each section containing a set of key-value pairs. The key is the name of the attribute of the class. The value is the value of the attribute. The value can be a string, an integer, a float, a boolean, a list, or a dictionary.

The dictionary configuration is useful when you want to configure the logging module using a JSON, YAML, or TOML configuration file that can be easily parsed into a dictionary.

```

# Example: Logging Configuration with a Dictionary

import logging.config

# Define the logging configuration as a dictionary
config = {

    # Version of the logging configuration
    'version': 1,

    # Define the formatters
    'formatters': {

        # Standard formatter
        'standard': {
            'format': '%(asctime)s [%(levelname)s] %(name)s: %(message)s',
        },
    },
}

```

```

# Define the handlers
'handlers': {

    # Console handler
    'console': {
        'class': 'logging.StreamHandler',
        'formatter': 'standard',
    },

},

# Root logger configuration
'root': {
    'handlers': ['console', ],
    'level': 'DEBUG',
},

# User logger configuration
'loggers': {

    # Logger myapp
    'myapp': {
        'handlers': ['console', ],
        'level': 'INFO',
        'propagate': False,
    },

},

}

# Configure logging using the dictionary-based configuration
logging.config.dictConfig(config)

# Create loggers
root_logger = logging.getLogger()
app_logger = logging.getLogger('myapp')

# Log messages
root_logger.debug('This is a debug message from the root logger')
app_logger.info('This is an info message from the app logger')

```

6.5. Best Practices

The following list is a set of best practices for using the logging module:

- Use the logging module instead of the print() function
- Start logging as soon as possible in the development cycle
- Design the logging hierarchy before writing the code

- Use unique identifiers to correlate log messages across different log files
- Use the logging levels to control the verbosity of the log messages
- Avoid excessive logging, especially sensitive or irrelevant information
- Use the logging format to add context to the log messages (e.g. thread, timestamp, etc.)
- Never hardcode sensitive data (e.g., passwords, API keys) in log messages
- Use log rotation to manage log file sizes and prevent disk space issues
- On errors log the stack trace to help with debugging and if possible, suggest resolutions
- Logging is thread-safe and thus can slow down the application if used excessively in a multi-threaded application
- Use a logging framework that supports structured logging (e.g. JSON, XML, etc.)

Chapter 7. Exceptions

Exceptions are a way to handle and manage errors or exceptional situations that can occur during the execution of a program. In Python, exceptions are objects that are raised (or thrown) when an error occurs, and they can be caught and handled using exception-handling mechanisms.

Key points:

- An exception is an object generated in an abnormal situation
- If the exception is not handled by the program it will stop
- Answer the question "WHAT TO DO ON AN ERROR?" and not "WHAT ERROR HAPPENED?"
- Solves the **semi-predicate** problem

7.1. Handling Exceptions

The *try-except* statement is used to handle exceptions. It is used to catch and handle exceptions that occur in a block of code. The *try-except* statement consists of a *try* block and one or more *except* blocks. The *try* block contains the code that may raise an exception. The *except* blocks contain the code that handles the exception.

The following table shows all the possible statements that can be used in a try-except statement:

Statement	Description
try	The try block contains the code that may raise an exception. It must be followed by at least one except block.
except	The except block contains the code that handles the exception. It must be preceded by a try block.
else	The else block contains the code that is executed if no exception is raised in the try block. It must be preceded by a try-except block.
finally	The finally block contains the code that is always executed, regardless of whether an exception is raised in the try block. It must be preceded by a try-except block.
raise	The raise statement is used to raise an exception. It must be followed by an exception object or an exception class.

```
# Example: Exception Handling with try-except statements

import time
```

```

# Custom exception
class OverheatError(Exception):
    pass

def pump(time_on):

    if time_on < 0:
        # Raise builtin exception
        raise ValueError("ERROR: Incorrect value for operation time...")

    for i in range(time_on):
        print("Pumping...")
        if i > 3:
            # Raise custom exception
            raise OverheatError("ERROR: The device overheated")
        time.sleep(1)

def operate_pump(time_on):

    print("START")

    try:
        pump(time_on)

    # First check expected errors
    except ValueError as e:
        print(e)

    except OverheatError as e:
        print(e)

    # Last check unexpected errors
    except Exception as e:
        # Forward to the caller, maybe it knows what to do with it
        print("Caught exception {}".format(e))

    # What to do in case no error occurred
    else:
        print("STOP")

    # Cleanup operations, executed in all cases
    finally:
        print("CLEANUP")

# Test with negative values
operate_pump(-1)
print()

```

```
# Test with valid values
operate_pump(3)
print()

# Overheat the pump
operate_pump(5)
print()
```

7.2. Program Flow

The typical program flow from procedural programming languages is to execute a sequence of statements such as function calls and check the result of each statement. If the result is successful, the program continues with the next statement. If the result is unsuccessful, the program stops and an error message is displayed.

This can lead to complex verification code that is difficult to read and maintain, especially if the program flow is complex and the number of possible errors is large. The following example shows how to check the result of a function call and handle the different errors:

```
# Example of exception flow without try/except statements

def func_a(x):
    # Function uses and returns only positive values
    if x < 0:
        # Simulate error condition
        return -1
    else:
        return 1

def func_b(x):
    # Functions uses and returns only negative values
    if x > 0:
        # Simulate error condition
        return 1
    else:
        return -1

def app(value):

    # Each function must have a check of the error condition
    # as there is no standard definition of an error

    # Check A
    error = func_a(value)
    if error < 0:
        print("STEP A: ERROR")
```

```

    else:
        print("STEP A: OK")

    # Check B
    error = func_b(value)
    if error > 0:
        print("STEP B: ERROR")
    else:
        print("STEP B: OK")

app(1)
app(0)
app(-1)

```

The same program flow can be implemented using exceptions. The following example shows how to implement the same program flow using exceptions:

```

# Example: Exception Flow with try-except statements
def func_a(x):
    # Function uses and returns only positive values
    if x < 0:
        # Simulate error condition
        raise ValueError("STEP A: ERROR")
    else:
        return 1

def func_b(x):
    # Functions uses and returns only negative values
    if x > 0:
        # Simulate error condition
        raise ValueError("STEP B: ERROR")
    else:
        return -1

def app(value):

    # Try to execute STEP A and STEP B
    try:
        func_a(value)
        func_b(value)

    # On error print something
    except ValueError as e:
        print(e)

    # Try to execute STEP B
    try:

```

```

        func_b(value)
        print("STEP B: OK")

    # On error print something
    except ValueError as e:
        print(e)

app(1)
app(0)
app(-1)

```

The advantage of the exceptions is that the program flow is not interrupted or polluted with *if-else* statements. The program flow is more readable and maintainable as the exceptions are handled in a separate block of code.

7.3. Chaining Exceptions

The exception chaining is a mechanism that allows you to chain exceptions together. It is used to propagate exceptions from one part of the program to another. It is useful when you want to propagate an exception from a low-level function to a high-level function. It is also useful when you want to add additional information to an exception. The following example shows how to chain exceptions together:

```

# Example: Exception Chaining

class MyException(Exception):
    pass

def main():
    try:
        print("main")
        func_a()
    except Exception as e:
        print(e)

def func_a():
    # Explicit chaining using 'raise'
    try:
        print("func_a")
        func_b()
    except Exception as e:
        # Convert to another exception type
        raise MyException(e)

def func_b():

```

```

# Unhandled exceptions sent to the caller by default
print("func_b")
read_file()

def read_file():
    print("read_file")
    raise Exception("Error raised in read_file()")

main()

```

7.4. Organizing Exceptions

The exceptions can be organized into a hierarchy. The hierarchy is useful when you want to catch multiple exceptions with a single except block. The following rules apply when the exceptions are organized into a hierarchy:

- Each package should have a file with its exception definitions (e.g. errors.py)
- Organize exceptions in categories (e.g. core, io, etc.)
- Define always a base exception class for the product that inherits from the `Exception` class

The folder structure of the project with the exception definitions is shown in the following example:

```

src/
├── socket
│   ├── __init__.py
│   ├── settings.py
│   └── errors.py    # <-- Exceptions for the core package
└── io
    ├── __init__.py
    ├── settings.py
    ├── handlers.py
    └── errors.py    # <-- Exceptions for the api package

```

Each of the exception files in a package should contain the exception class hierarchy. The developer is required to define a base exception class for the package that inherits from the `Exception` and implement all other exceptions as subclasses of the base exception class. The following example shows the exception definitions for the `core` package:

```

# Example: Exception Hierarchy in a Package

class SocketError(Exception):

    def __init__(self, message):

```

```

        self.message = message

    def __str__(self):
        return self.message

class ConnectError(SocketError):

    def __init__(self, message, status):
        super(ConnectError, self).__init__(message)
        self.status = status

    def __str__(self):
        return '{}: {}'.format(self.status, self.message)

class DataTransferError(SocketError):

    def __init__(self, message, status, source, destination):
        super(DataTransferError, self).__init__(message)
        self.status = status
        self.source = source
        self.destination = destination

    def __str__(self):
        return '{}: {} from {} to {}'.format(self.status, self.message, self.source,
                                              self.destination)

class DisconnectError(SocketError):

    def __init__(self, message, status):
        super(DisconnectError, self).__init__(message)
        self.status = status

    def __str__(self):
        return '{}: {}'.format(self.status, self.message)

if __name__ == "__main__":

    try:
        raise ConnectError('Connection failed', 100)

    except ConnectError as e:
        print(e)

    try:
        raise DataTransferError(
            message='Data transfer failed',
            status=200,

```

```

        source='myhost',
        destination='remotehost',
    )

except DataTransferError as e:
    print(e)

try:
    raise DisconnectError('Connection failed', 300)

except DisconnectError as e:
    print(e)

```

The following examples shows how to catch multiple exceptions with a single except block:

```

# Example: Catch Multiple Exceptions with a Single Except Block

from exception_hierarchy import *

def main():

    # Example 1: Handling all exceptions inherited from SocketError
    try:
        raise ConnectError('Connection failed', 100)

    except SocketError as e:
        print(e)

    # Example 2: Handling a group of exceptions with a single except block
    try:
        raise DisconnectError('Connection failed', 100)

    except (ConnectError, DataTransferError, DisconnectError) as e:
        print(e)

main()

```

7.5. Built-in Exceptions

The Python interpreter provides a set of built-in exceptions that can be used to handle common errors. The following table shows the most common built-in exceptions:

7.5.1. System Exceptions

Class	Description
BaseException	The base class for all built-in exceptions. It is inherited by all other exception classes.
Exception	The base class for all built-in, non-system-exiting exceptions. It is inherited by all other exception classes.
SystemExit	The exception that is raised when the program is terminated using the <code>sys.exit()</code> function. It is inherited from the BaseException class.
KeyboardInterrupt	The exception that is raised when the program is interrupted by the user (e.g. CTRL+C). It is inherited from the BaseException class.
GeneratorExit	The exception that is raised when a generator or coroutine is closed. It is inherited from the BaseException class.

7.5.2. Non-System Exceptions

Class	Description
Exception	The base class for all built-in, non-system-exiting exceptions. It is inherited by all other exception classes.
AssertionError	The exception that is raised when an assert statement fails. It is inherited from the Exception class.
StopIteration	The exception that is raised when the <code>next()</code> method of an iterator does not return any more values. It is inherited from the Exception class.
StopAsyncIteration	The exception that is raised when the <code>anext()</code> method of an asynchronous iterator does not return any more values. It is inherited from the Exception class.
AttributeError	The exception that is raised when an attribute reference or assignment fails. It is inherited from the Exception class.
BufferError	The exception that is raised when a buffer related operation fails. It is inherited from the Exception class.
EOFError	The exception that is raised when the <code>input()</code> function hits an end-of-file condition (e.g. CTRL+D). It is inherited from the Exception class.

Class	Description
MemoryError	The exception that is raised when an operation runs out of memory. It is inherited from the Exception class.
ReferenceError	The exception that is raised when a weak reference proxy is used to access a garbage collected referent. It is inherited from the Exception class.
SystemError	The exception that is raised when the interpreter detects an internal error. It is inherited from the Exception class.
TypeError	The exception that is raised when an operation or function is applied to an object of an inappropriate type. It is inherited from the Exception class.
ValueError	The exception that is raised when an operation or function receives an argument that has the right type but an inappropriate value. It is inherited from the Exception class.

7.5.3. Arithmetic Errors

Class	Description
ArithmeticError	The base class for all built-in arithmetic errors. It is inherited from the Exception class.
FloatingPointError	The exception that is raised when a floating point operation fails. It is inherited from the ArithmeticError class.
OverflowError	The exception that is raised when an arithmetic operation exceeds the largest possible value for a numeric type. It is inherited from the ArithmeticError class.
ZeroDivisionError	The exception that is raised when the second argument of a division or modulo operation is zero. It is inherited from the ArithmeticError class.

7.5.4. Module Errors

Class	Description
ImportError	The exception that is raised when an <code>import</code> statement fails. It is inherited from the Exception class.

Class	Description
ModuleNotFoundError	The exception that is raised when a module could not be found. It is inherited from the ImportError class.

7.5.5. Name Errors

Class	Description
NameError	The exception that is raised when a local or global name is not found. It is inherited from the Exception class.
UnboundLocalError	The exception that is raised when a local variable is referenced before it has been assigned a value. It is inherited from the NameError class.

7.5.6. OS Errors

Class	Description
OSError	The base class for all built-in system-exiting exceptions. It is inherited from the Exception class.
BlockingIOError	The exception that is raised when an operation would block on an object (like a socket) set for non-blocking operation. It is inherited from the OSError class.
ChildProcessError	The exception that is raised when an operation on a child process fails. It is inherited from the OSError class.
ConnectionError	The base class for all built-in connection errors. It is inherited from the OSError class.
BrokenPipeError	The exception that is raised when a broken pipe error occurs. It is inherited from the ConnectionError class.
ConnectionAbortedError	The exception that is raised when a connection attempt is aborted by the peer. It is inherited from the ConnectionError class.
ConnectionRefusedError	The exception that is raised when a connection attempt is refused by the peer. It is inherited from the ConnectionError class.

Class	Description
ConnectionResetError	The exception that is raised when a connection is reset by the peer. It is inherited from the ConnectionError class.
FileExistsError	The exception that is raised when a file or directory cannot be created because it already exists. It is inherited from the OSError class.
FileNotFoundError	The exception that is raised when a file or directory cannot be found. It is inherited from the OSError class.
InterruptedError	The exception that is raised when a system call is interrupted by an incoming signal. It is inherited from the OSError class.
IsADirectoryError	The exception that is raised when a file operation (e.g. <code>os.remove()</code>) is requested on a directory. It is inherited from the OSError class.
NotADirectoryError	The exception that is raised when a directory operation (e.g. <code>os.listdir()</code>). It is inherited from the OSError class.
PermissionError	The exception that is raised when a file operation is not permitted. It is inherited from the OSError class.
ProcessLookupError	The exception that is raised when a process lookup fails. It is inherited from the OSError class.
TimeoutError	The exception that is raised when a system function times out. It is inherited from the OSError class.

7.5.7. Runtime Errors

Class	Description
RuntimeError	The base class for all built-in runtime errors. It is inherited from the Exception class.
NotImplementedError	The exception that is raised when an abstract method is not implemented. It is inherited from the RuntimeError class.
RecursionError	The exception that is raised when the maximum recursion depth is exceeded. It is inherited from the RuntimeError class.

Class	Description
SystemError	The exception that is raised when the interpreter detects an internal error. It is inherited from the Exception class.

7.5.8. Syntax Errors

Class	Description
SyntaxError	The exception that is raised when a syntax error occurs. It is inherited from the Exception class.
IndentationError	The base class for all built-in indentation errors. It is inherited from the SyntaxError class.
TabError	The exception that is raised when indentation contains tabs and spaces. It is inherited from the IndentationError class.

7.5.9. Unicode Errors

Class	Description
UnicodeError	The base class for all built-in Unicode errors. It is inherited from the ValueError class. It is inherited from the Exception class.
UnicodeDecodeError	The exception that is raised when a Unicode-related error occurs during decoding. It is inherited from the UnicodeError class.
UnicodeEncodeError	The exception that is raised when a Unicode-related error occurs during encoding. It is inherited from the UnicodeError class.
UnicodeTranslateError	The exception that is raised when a Unicode-related error occurs during the translation of Unicode to another character set. It is inherited from the UnicodeError class.

7.5.10. Warning Errors

Class	Description
Warning	The base class for all built-in warnings. It is inherited from the Exception class. It is inherited from the Exception class.
DeprecationWarning	The warning that is raised when a deprecated feature is used. It is inherited from the Warning class.

Class	Description
PendingDeprecationWarning	The warning that is raised when a feature is pending deprecation. It is inherited from the Warning class.
RuntimeWarning	The warning that is raised when a runtime warning occurs. It is inherited from the Warning class.
SyntaxWarning	The warning that is raised when a syntax warning occurs. It is inherited from the Warning class.
UserWarning	The warning that is raised when a user warning occurs. It is inherited from the Warning class.
FutureWarning	The warning that is raised when a future warning occurs. It is inherited from the Warning class.
ImportWarning	The warning that is raised when an import warning occurs. It is inherited from the Warning class.
UnicodeWarning	The warning that is raised when a Unicode warning occurs. It is inherited from the Warning class.
BytesWarning	The warning that is raised when a bytes warning occurs. It is inherited from the Warning class.
ResourceWarning	The warning that is raised when a resource warning occurs. It is inherited from the Warning class.

7.5.11. Lookup Errors

Class	Description
LookupError	The base class for all built-in lookup errors. It is inherited from the Exception class.
IndexError	The exception that is raised when a sequence index is out of range. It is inherited from the LookupError class.
KeyError	The exception that is raised when a mapping key is not found in a dictionary. It is inherited from the LookupError class.

7.6. Best Practices

- Whenever possible use exceptions instead of return codes

- Always wrap the production code in a try-except block
- Handle first the known and then unknown errors
- Prefer custom errors over builtin errors
- Have always a custom base exception class

Chapter 8. Dunder Methods

Dunder methods are special methods that are defined using the double underscore notation. They are used to emulate the behavior of built-in types and provide a way to customize the behavior of classes. Dunder methods are also called magic methods. Some of the most common use cases for dunder methods are:

- Emulate the behavior of built-in types
- Customize the behavior of classes
- Object representation
- Implement operator overloading
- Implement iterators
- Implement context managers

8.1. Object representation

Representation	Dunder method	Notes
Human-readable	<code>__str__</code>	If you want to provide a human-readable string representation of an object, use the <code>__str__</code> method. This method should return a string that is easy for a human to understand.
Unambiguous	<code>__repr__</code>	If you want to provide an unambiguous string representation of an object, use the <code>__repr__</code> method. This method should return a string that can be used to recreate the object.
Custom formatting	<code>__format__</code>	If you want to provide custom string formatting of an object, use the <code>__format__</code> method. This method should return a string that is formatted according to the format specification.

Representation	Dunder method	Notes
Raw byte string	<code>__bytes__</code>	If you want to provide a raw byte string representation of an object, use the <code>__bytes__</code> method. This method should return a byte string that can be used to recreate the object. This method is used by the <code>bytes()</code> function.
Unique	<code>__hash__</code>	If you want to compute the hash of an object, use the <code>__hash__</code> method. This method should return an integer that is used to compare objects in dictionaries and sets. This method is used by the <code>hash()</code> function.

```
# Example: Dunder methods for object representation

import struct

class Point(object):

    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def __repr__(self):
        return "Point(x={}, y={})".format(self.x, self.y)

    def __str__(self):
        return "({}, {})".format(self.x, self.y)

    def __format__(self, format_spec):
        fmt = "({:}" + format_spec + "}" + "{:}" + format_spec + "}"
        fmt = fmt.format(self.x, self.y)
        return fmt

    def __hash__(self):
        return hash((self.x, self.y))

    def __bytes__(self):
        result = []
        for item in (self.x, self.y):
            result.extend(struct.pack("!f", item))
```

```
return bytes(result)
```

```
p = Point(1, 2)
print(hash(p))
print(repr(p))
print(str(p))
print("{:.3f}".format(p))
print(bytes(p))
```

8.2. Custom behavior of classes and instances

Behavior	Dunder method	Notes
Customize creation	<code>__new__</code>	If you want to customize the creation of an instance, use the <code>__new__</code> method. This method should return an instance of the class. It is used to create the instance before it is initialized.
Customize initialization	<code>__init__</code>	If you want to customize the initialization of an instance, use the <code>__init__</code> method. This method is called right after the instance has been created. It is used to initialize the instance variables of the class.
Customize calling	<code>__call__</code>	If you want to customize the behavior of calling an instance, use the <code>__call__</code> method. This method is called when the instance is called like a function.
Customize deletion	<code>__del__</code>	If you want to customize the deletion of an instance, use the <code>__del__</code> method. This method is called when the instance is deleted. It is used by the <code>del</code> statement.

```
# Example: Dunder methods for customizing object behavior

class TestClass(object):

    def __init__(self):
        print("I'm initialized!")
```

```

        self.name = "DunderClass"

    def __new__(cls, *args, **kwargs):
        print("I'm created!")
        return super(TestClass, cls).__new__(cls)

    def __call__(self, *args, **kwargs):
        print("I'm called! My name is {}".format(self.name))

    def __del__(self):
        print("I'm deleted!")

# Create and initialize an instance of DunderClass
test = TestClass()

# Call the instance
test()

# Delete the instance
del test

```

8.3. Custom behavior of built-in types

Built-in type	Dunder method	Notes
<code>int()</code>	<code>__int__</code>	If you want to convert an object to an integer, use the <code>__int__</code> method. This method should return an integer. It is used by the <code>int()</code> function.
<code>float()</code>	<code>__float__</code>	If you want to convert an object to a float, use the <code>__float__</code> method. This method should return a float. It is used by the <code>float()</code> function.
<code>complex()</code>	<code>__complex__</code>	If you want to convert an object to a complex number, use the <code>__complex__</code> method. This method should return a complex number. It is used by the <code>complex()</code> function.
<code>bool()</code>	<code>__bool__</code>	If you want to evaluate the truth value of an object, use the <code>__bool__</code> method. This method should return a boolean. It is used by the <code>bool()</code> function.

Built-in type	Dunder method	Notes
<code>bytes()</code>	<code>__bytes__</code>	If you want to convert an object into bytes object use the <code>__bytes__</code> method. It is used by the <code>bytes()</code> function.

Example: Dunder methods for object representation

```
import struct
import math
```

```
class Point(object):

    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)
        self.__length = self._calc_length()

    def _calc_length(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)

    def __bool__(self):
        return bool(self.x or self.y)

    def __int__(self):
        return int(self._calc_length())

    def __float__(self):
        return float(self._calc_length())

    def __complex__(self):
        return complex(self.x, self.y)

    def __bytes__(self):
        return bytes(int(self.__length))
```

```
c = Point(1, 2)
print("Point({}, {})".format(c.x, c.y))
print("bool(c)      : {}".format(bool(c)))
print("int(c)       : {}".format(int(c)))
print("float(c)      : {}".format(float(c)))
print("complex(c)   : {}".format(complex(c)))
print("bytes(c)     : {}".format(bytes(c)))
```

8.4. Operator overloading

Operator	Dunder method
<code>==</code>	<code>__eq__</code>
<code>!=</code>	<code>__ne__</code>
<code><</code>	<code>__lt__</code>
<code><=</code>	<code>__le__</code>
<code>></code>	<code>__gt__</code>
<code>>=</code>	<code>__ge__</code>
<code>+</code>	<code>__add__</code>
<code>-</code>	<code>__sub__</code>
<code>*</code>	<code>__mul__</code>
<code>/</code>	<code>__truediv__</code>
<code>//</code>	<code>__floordiv__</code>
<code>%</code>	<code>__mod__</code>
<code>**</code>	<code>__pow__</code>
<code>&</code>	<code>__and__</code>
<code> </code>	<code>__or__</code>
<code>^</code>	<code>__xor__</code>
<code><<</code>	<code>__lshift__</code>
<code>>></code>	<code>__rshift__</code>
<code>+=</code>	<code>__iadd__</code>
<code>-=</code>	<code>__isub__</code>
<code>*=</code>	<code>__imul__</code>
<code>/=</code>	<code>__itruediv__</code>
<code>//=</code>	<code>__ifloordiv__</code>
<code>%=</code>	<code>__imod__</code>
<code>**=</code>	<code>__ipow__</code>
<code>&=</code>	<code>__iand__</code>
<code> =</code>	<code>__ior__</code>
<code>^=</code>	<code>__ixor__</code>
<code><<=</code>	<code>__ilshift__</code>
<code>>>=</code>	<code>__irshift__</code>
<code>-</code>	<code>__neg__</code>
<code>+</code>	<code>__pos__</code>
<code>~</code>	<code>__invert__</code>

```
# Example: Dunder methods for operator overloading
```

```
class Point(object):
```

```

def __init__(self, x, y):
    self.x = float(x)
    self.y = float(y)

def __add__(self, other):
    return Point(
        x=self.x + other.x,
        y=self.y + other.y
    )

def __sub__(self, other):
    return Point(
        x=self.x - other.x,
        y=self.y - other.y
    )

def __mul__(self, other):
    return Point(
        x=self.x * other.x,
        y=self.y * other.y
    )

def __divmod__(self, other):
    return Point(
        x=self.x % other.x,
        y=self.y % other.y
    )

def __truediv__(self, other):
    return Point(
        x=self.x / other.x,
        y=self.y / other.y
    )

def __floordiv__(self, other):
    return Point(
        x=self.x // other.x,
        y=self.y // other.y
    )

```

```

a = Point(1, 2)
b = Point(3, 4)

z = a + b
print(z.x, z.y)

```

8.5. Iterator protocol

Function	Dunder method	Notes
Create iterator	<code>__iter__</code>	If you want to iterate over an object, use the <code>__iter__</code> method. This method should return an iterator. It is used by the <code>iter()</code> function.
Get next item	<code>__next__</code>	If you want to get the next value of an iterator, use the <code>__next__</code> method. This method should return the next value of the iterator. It is used by the <code>next()</code> .

```
# Example: Dunder methods for the iterator protocol

class Stack(object):

    def __init__(self):
        self._items = []
        self._index = -1

    def push(self, item):
        self._items.append(item)

    def pop(self):
        try:
            return self._items.pop()
        except IndexError:
            raise IndexError("Pop from an empty stack")

    def __iter__(self):
        return self

    def __next__(self):
        try:
            next_item = self._items[self._index]
            self._index += 1
            return next_item

        except IndexError:
            raise StopIteration

s = Stack()
s.push(1)
s.push(2)
s.push(3)
```

```
for item in s:
    print(item)
```

8.6. Context manager protocol

Function	Dunder method	Notes
Enter context	<code>__enter__</code>	If you want to customize the behavior of the <code>with</code> statement when entering the block, use the <code>__enter__</code> method. This method should return an object that will be assigned to the variable in the <code>with</code> statement.
Exit context	<code>__exit__</code>	If you want to customize the behavior of the <code>with</code> statement when exiting the block, use the <code>__exit__</code> method. This method should return a boolean that indicates whether the exception was handled or not.

```
# Example: Dunder methods for context manager

class Complex(object):

    def __init__(self, real, imag):
        self.real = float(real)
        self.imag = float(imag)

    def __enter__(self):
        print(">>> Inside __enter__")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(">>> Inside __exit__")
        print(" Execution type:", exc_type)
        print(" Execution value:", exc_val)
        print(" Traceback:", exc_tb)

        if exc_type is not None:
            print("\nException occurred")
            return True
        else:
            print("\nNo exception occurred")
            return False
```



```
with Complex(1, 0) as c:  
    print(">>> Inside with block")  
    print(c.real, c.imag)  
    print(c.real / 0)
```

8.7. Best Practices

- Document what dunder methods do and what they are expected to return. This helps other developers understand the purpose and behavior of these methods.
- Dunder methods should be simple and focused. If a dunder method becomes too complex, consider refactoring it into smaller, more manageable methods.
- Avoid overloading too many dunder methods in a single class. Overloading too many can make your code complex and less maintainable.
- Include tests for dunder methods in your test suite to ensure they behave as expected. Test cases for equality (*eq*) and string representation (*str*) are commonly used.
- Overload dunder methods for common operators (e.g., +, -, *) when it makes sense for the class. This allows objects of the class to be used in intuitive ways.

Chapter 9. Advanced Data

9.1. Data Manipulation

9.1.1. filter

The `filter()` function returns an iterator that contains the elements of an iterable for which a given function returns true. For example the following code returns an iterator that contains the even numbers of a list:

```
# Example: Filter Data using the filter() function

def is_even(x):
    return x % 2 == 0

# Sample data
sample = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Filter using a filtering function (first) and an iterable (second)
iterator = filter(is_even, sample)
print(list(iterator))

# Filter using a lambda function (first) and an iterable (second)
iterator = filter(lambda x: x % 2 == 0, sample)
print(list(iterator))
```

9.1.2. map

The `map()` function returns an iterator that contains the results of applying a given function to each element of an iterable.

```
# Example: Filter Data using the filter() function

def sqr(x):
    return x * x

# Sample data
sample = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Map using a mapping function (first) and an iterable (second)
iterator = map(sqr, sample)
print(list(iterator))
```

```
# Map using a lambda function (first) and an iterable (second)
iterator = map(lambda x: x * x, sample)
print(list(iterator))
```

9.1.3. reduce

The `reduce()` function returns a single value that is the result of applying a given function to each element of an iterable. This function is useful when you want to reduce a list of values to a single value. You have to import the `reduce()` function from the `functools` module in order to use it.

```
# Example: Reduce Data using the reduce() function

from functools import reduce

def total(x, y):
    return x + y

# Sample data
sample = [1, 1, 1]

# Map using a mapping function (first) and an iterable (second)
value = reduce(total, sample)
print(value)

# Map using a lambda function (first) and an iterable (second)
iterator = reduce(lambda x, y: x + y, sample)
print(value)
```

In the example above the `reduce()` function is used to calculate the sum of a list of numbers. The `reduce` function will take the current and next element and apply the function to them. The result of the function will be used as the current element for the next iteration.

9.1.4. reversed

The `reversed()` function returns an iterator that contains the elements of an iterable in reverse order (i.e. from last to first).

```
# Example: Filter Data using the filter() function

# Sample data
sample_1 = [1, 2, 5, 4, 3]
sample_2 = {1: 'a', 2: 'b', 5: 'd', 4: 'c', 3: 'e'}

# Reverse a list
iterator = reversed(sample_1)
print(list(iterator))
```

```
# Reverse a dictionary
iterator = reversed(sample_2.items())
print(list(iterator))
```

9.1.5. sorted

The `sorted()` function returns an iterator that contains the elements of an iterable in sorted order. The sample can be sorted in ascending or descending order. Not to be mistaken with the `sort()` method of the list class.

```
# Example: Sort Data using the sorted() function

class Book(object):

    def __init__(self, title, author, year):
        self.title = title
        self.author = author
        self.year = year

    def __repr__(self):
        return "Book({title}, {author}, {year})".format(
            title=self.title, author=self.author, year=self.year)

# List of books
books = [
    Book('The Great Gatsby', 'F. Scott Fitzgerald', 1925),
    Book('To Kill a Mockingbird', 'Harper Lee', 1960),
    Book('1984', 'George Orwell', 1949),
    Book('Brave New World', 'Aldous Huxley', 1932),
    Book('The Catcher in the Rye', 'J.D. Salinger', 1951),
]

# Sort the list of dictionaries based on the 'year' key in each dictionary
sorted_books = sorted(books, key=lambda book: book.year)

# Print the sorted list
for b in sorted_books:
    print(b)
```

9.1.6. zip

The `zip()` function returns an iterator that contains tuples of elements from each of the iterables passed as arguments. The pairing is done in order, so the first element of the first iterable is paired with the first element of the second iterable, and so on.

If the iterables passed as arguments have different lengths, the resulting iterator will contain tuples of the shortest length.

```
# Example: Zipping Data using the zip() function
```

```
# Sample data
```

```
numbers = [1, 2, 3]
```

```
letters = ['a', 'b', 'c']
```

```
# Zip two lists
```

```
zipped = zip(numbers, letters)
```

```
zipped_list = list(zipped)
```

```
print(zipped_list)
```

```
# Unzip into two lists
```

```
unzipped = zip(*zipped_list)
```

```
numbers, letters = map(list, unzipped)
```

```
print(numbers)
```

```
print(letters)
```

9.1.7. unpack

The unpacking is done by using the `*` operator. The `*` operator is used to unpack an iterable into a sequence of arguments. Typically this is used on a list or tuple to unpack it into a sequence of arguments for a function call.

```
# Example: Unpacking Arguments using the * operator and ** operator
```

```
# A list of arguments
```

```
pos_args = [1, 2, 3, 4]
```

```
keyword_args = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

```
# A sample function that takes 4 arguments
```

```
# and prints the,
```

```
def func1(a, b, c, d):
```

```
    print(a, b, c, d)
```

```
# Variable number of arguments
```

```
def func2(a, b, c, d, *args):
```

```
    print(a + b + c + d + sum(args))
```

```
# Variable number of keyword arguments
```

```
def func3(**kwargs):
```

```
    print(kwargs['a'] + kwargs['b'] + kwargs['c'] + kwargs['d'])
```

```
func1(*pos_args)
```

```
func2(*pos_args)
```

```
func3(**keyword_args)
```

```
# test = "PYTHON"
# unpacked = [*test]
# print(unpacked)
```

9.1.8. copy

The `copy` module provides two functions for copying objects: `copy()` and `deepcopy()`. The `copy()` function returns a shallow copy of an object. A shallow copy will only copy the top-level object elements and keep the references to the nested objects. A deep copy will copy all the elements of an object, including the nested objects.

```
# Example: Data Copying

import copy

# Create a deeply nested dictionary
numbers_dict = {
    "numbers": {
        "integers": [1, 2, 3, 4, 5],
        "floats": [1.1, 2.2, 3.3, 4.4, 5.5]
    }
}

# Create a shallow copy of the dictionary
shallow_copy = copy.copy(numbers_dict)

# Create a deep copy of the dictionary
deep_copy = copy.deepcopy(numbers_dict)

# Modify the original dictionary
numbers_dict["numbers"]["integers"].append(6)
numbers_dict["numbers"]["floats"].append(6.6)

# Print the original dictionary (modified)
print("Original", numbers_dict)

# Print the shallow copy (modified)
print("Shallow", shallow_copy)

# Print the deep copy (not modified)
print("Deep", deep_copy)
```

9.2. Binary Serialization

Serialization is the process of converting a Python object (lists, dict, tuples, etc.) into byte streams (***machine-readable format***). The byte streams can be then saved to disks or can be transferred over the network. The byte streams saved on the file contains the necessary information to

reconstruct the original python object. The process of converting byte streams back to python objects is called de-serialization.

WARNING

The serialization process is not secure. It is possible to create malicious pickle files that can execute arbitrary code when deserialized. Thus it is very important to ensure that serialized objects sent over the network are encrypted and authenticated.

9.2.1. pickle

Pickle is a module that provides functions for converting Python objects into byte streams and vice versa. It is used to serialize and deserialize Python objects. It is one of the most popular serialization modules in Python and is part of the standard library.

The main methods of the pickle module are:

- `dump()` - Serialize an object to a file
- `dumps()` - Serialize an object to a string
- `load()` - Deserialize an object from a file
- `loads()` - Deserialize an object from a string

The following example shows how to serialize and deserialize a Python object using the pickle module:

```
# Example: Serializing and Deserializing Objects with Pickle

import pickle
import sys

class Book(object):

    def __init__(self, title, author, price, year):
        self.title = title
        self.author = author
        self.price = price
        self.year = year

if __name__ == "__main__":

    # Create a book object
    book = Book(title='Python for Dummies', author='John Smith', price=25.0,
year=2014)

    # Print the book object
    print(book.__dict__)
```

```

# Serialize the book object
serialized_book = pickle.dumps(book)

# Print the size of the serialized object
print('Size of the serialized object: {}
bytes'.format(sys.getsizeof(serialized_book)))

# Print the serialized object
print(serialized_book)

# Deserialize the book object
deserialized_book = pickle.loads(serialized_book)

# Print the book object
print(deserialized_book.__dict__)

```

9.2.2. dill

Dill is a fork of the `pickle` module that provides additional features such as support for more data types and support for more Python versions. It is used to serialize and deserialize any Python objects.

Unfortunately, `dill` is not part of the standard library and needs to be installed separately. It can be installed using the following command:

```
pip install dill
```

Dill exhibits the same behavior as `pickle` when serializing and deserializing Python objects and uses the same syntax. Just replace `pickle` with `dill` in the code and it will work the same. The main methods of the `dill` module are:

- `dump()` - Serialize an object to a file
- `dumps()` - Serialize an object to a string
- `load()` - Deserialize an object from a file
- `loads()` - Deserialize an object from a string

```

# Example: Serializing and Deserializing Objects with Pickle

import dill
import sys

class Book(object):

    def __init__(self, title, author, price, year):
        self.title = title
        self.author = author

```



```

        self.price = price
        self.year = year

if __name__ == "__main__":

    # Create a book object
    book = Book(title='Python for Dummies', author='John Smith', price=25.0,
year=2014)

    # Print the book object
    print(book.__dict__)

    # Serialize the book object
    serialized_book = dill.dumps(book)

    # Print the size of the serialized object
    print('Size of the serialized object: {}
bytes'.format(sys.getsizeof(serialized_book)))

    # Print the serialized object
    print(serialized_book)

    # Deserialize the book object
    deserialized_book = dill.loads(serialized_book)

    # Print the book object
    print(deserialized_book.__dict__)

```

9.3. Text Serialization

Another form of serialization is text serialization. Text serialization is the process of storing data in a **human-readable format**. The following are some of the most common text serialization formats:

- [JSON](#)
- [HTML](#)
- [CSV](#)
- [XML](#)
- [YAML](#)
- [TOML](#)

9.4. Data Conversion

9.4.1. struct

The module `struct` provides functions for converting between strings and binary data. It is used to

define custom binary data formats. This module is particularly useful when you need to interact with binary data formats such as those found in network protocols, file formats, or low-level system interactions.

The main methods of the struct module are:

- `pack()` - Convert a Python object to a string
- `unpack()` - Convert a string to a Python object
- `calcsize()` - Calculate the size of a string based on a format string

Format	Data type	Python type	Size
<code>x</code>	pad byte	no value	1
<code>c</code>	char	string of length 1	1
<code>b</code>	signed char	integer	1
<code>B</code>	unsigned char	integer	1
<code>?</code>	Bool	bool	1
<code>h</code>	short	integer	2
<code>H</code>	unsigned short	integer	2
<code>i</code>	int	integer	4
<code>I</code>	unsigned int	integer	4
<code>l</code>	long	integer	4
<code>L</code>	unsigned long	integer	4
<code>q</code>	long long	integer	8
<code>Q</code>	unsigned long long	integer	8
<code>f</code>	float	float	4
<code>d</code>	double	float	8
<code>s</code>	char[]	string	1
<code>p</code>	char[]	string	1
<code>P</code>	void *	integer	4

Additionally `struct` module supports the following format alignment characters:

- `=` - native byte order
- `<` - little-endian
- `>` - big-endian
- `!` - network byte order

The following example shows how to convert a Python object to a string and vice versa using the `struct` module:

Example: Serializing and Deserializing Objects with Pickle

```
import struct

class Book(object):

    def __init__(self, title, author, price, year):
        self.title = title
        self.author = author
        self.price = price
        self.year = year

if __name__ == "__main__":

    # Create a book object
    book = Book(title='Python for Dummies', author='John Smith', price=25.0,
year=2014)

    # Define the byte stream format (32s = 32 characters, f = float, i = integer),
big-endian (>)
    serialized = struct.pack('>32s 32s f i',
                                book.title.encode('utf-8'),
                                book.author.encode('utf-8'),
                                book.price,
                                book.year
                                )

    # Print the size of the serialized object (32 + 32 + 4 + 4 = 72 bytes)
    print('Size of the serialized object: {} bytes'.format(struct.calcsize('32s 32s f
i'))))

    # Print the binary stream (72 bytes)
    print(serialized)

    # Deserialize the data
    title, author, price, year = struct.unpack('32s 32s f i', serialized)
    print(title.decode('utf-8').strip('\x00'))
    print(author.decode('utf-8').strip('\x00'))
    print(price)
    print(year)
```

And the following example shows how to convert a Python object to a string and vice versa using the struct module and buffers:

Example: Serializing and Deserializing Objects with Pickle

```
import struct
```

```

class Book(object):

    def __init__(self, title, author, price, year):
        self.title = title
        self.author = author
        self.price = price
        self.year = year

if __name__ == "__main__":

    # Create a book object
    book = Book(title='Python for Dummies', author='John Smith', price=25.0,
year=2014)

    # Define the byte stream format (32s = 32 characters, f = float, i = integer),
big-endian (>)
    format_string = '>32s 32s f i'

    # Create the buffer
    buffer_size = struct.calcsize(format_string)
    buffer = bytearray(buffer_size)

    # Pack the data into the buffer
    serialized = struct.pack(format_string,
                             book.title.encode('utf-8'),
                             book.author.encode('utf-8'),
                             book.price,
                             book.year
                             )

    # Print the size of the serialized object (32 + 32 + 4 + 4 = 72 bytes)
    print('Size of the serialized object: {} bytes'.format(struct.calcsize('32s 32s f
i'))))

    # Print the binary stream (72 bytes)
    print(serialized)

    # Deserialize the data
    title, author, price, year = struct.unpack_from(format_string, serialized)
    print(title.decode('utf-8').strip('\x00'))
    print(author.decode('utf-8').strip('\x00'))
    print(price)
    print(year)

```

9.4.2. base64

Base64 is a binary-to-text encoding scheme that represents binary data based on the ASCII string

format. It is commonly used in computer systems to encode binary data, such as images, audio files, and other non-text data, into a format that can be safely transmitted over text-based protocols, like email or HTTP.

Each symbol in the Base64 encoding is represented by 6 bits. The 6 bits are used to represent 64 different values, which is why it is called Base64. The 64 values are the 26 uppercase letters, the 26 lowercase letters, the 10 digits, and the symbols `+` and `/`.

The conversion from binary to Base64 is done by dividing the binary data into 6-bit chunks and converting each chunk to a Base64 symbol. If the binary data is not divisible by 6, then the remaining bits are padded with zeros.

```
# Example: Base64 Encoding and Decoding

import base64

# Convert the string to bytes
text_expected = 'Здравейте, хора!'
text_stream = text_expected.encode('utf-8')

# Encode the byte stream in Base64
encoded = base64.b64encode(text_stream)
print(encoded)

# Decode the byte stream from Base64
decoded = base64.b64decode(encoded)

# Print the decoded string
text_obtained = decoded.decode('utf-8')
print(text_obtained)

# Assert that the expected and obtained strings are equal
assert text_expected == text_obtained
```

9.4.3. binascii

The `binascii` module in Python provides a collection of functions for binary-to-text and text-to-binary conversions, as well as various other binary data manipulations. It is commonly used for working with binary data and encoding schemes like Base64, hexadecimal, and uuencoding.

All of these formats are used to represent binary data in a human-readable format and are part of the MIME (Multipurpose Internet Mail Extensions) standard. The MIME standard is used to specify the format of non-text data in email messages and other Internet protocols.

Format	Description
hex	Convert binary data to a hexadecimal string (2 characters per byte)

Format	Description
base64	Convert binary data to a Base64-encoded string (6-bit, printable ASCII characters)
uuencode	Convert binary data to a uuencoded string (7-bit, printable ASCII characters)
quoted-printable	Convert binary data to a quoted-printable string (7-bit, printable ASCII characters)

Example: binascii module

```
import binascii
```

```
def test_base64():
```

```
    print("Test Base64")
```

```
    # Convert the string to bytes
```

```
    text_out = 'Здравейте, хора!'
```

```
    text_stream = text_out.encode('utf-8')
```

```
    print(text_out)
```

```
    # Encode the byte stream in Base64
```

```
    encoded = binascii.b2a_base64(text_stream)
```

```
    print(encoded)
```

```
    # Decode the byte stream from Base64
```

```
    decoded = binascii.a2b_base64(encoded)
```

```
    print(decoded)
```

```
    # Print the decoded string
```

```
    text_in = decoded.decode('utf-8')
```

```
    print(text_in)
```

```
    # Assert that the expected and obtained strings are equal
```

```
    assert text_in == text_out
```

```
    print()
```

```
def test_hex():
```

```
    print("Test Hex")
```

```
    # Convert the string to bytes
```

```
    text_out = 'Здравейте, хора!'
```

```
    text_stream = text_out.encode('utf-8')
```

```

print(text_out)

# Encode the byte stream in hexadecimal
encoded = binascii.b2a_hex(text_stream)
print(encoded)

# Decode the byte stream from hexadecimal
decoded = binascii.a2b_hex(encoded)
print(decoded)

# Print the decoded string
text_in = decoded.decode('utf-8')
print(text_in)

# Assert that the expected and obtained strings are equal
assert text_in == text_out

print()

def test_uu():

    print("Test UUEncode")

    # Convert the string to bytes
    text_out = 'Здравейте, хора!'
    text_stream = text_out.encode('utf-8')
    print(text_out)

    # Encode the byte stream in uuencode
    encoded = binascii.b2a_uu(text_stream)
    print(encoded)

    # Decode the byte stream from uuencode
    decoded = binascii.a2b_uu(encoded)
    print(decoded)

    # Print the decoded string
    text_in = decoded.decode('utf-8')
    print(text_in)

    # Assert that the expected and obtained strings are equal
    assert text_in == text_out

    print()

if __name__ == "__main__":
    test_base64()
    test_hex()

```

9.4.4. codecs

The `codecs` module in Python provides a framework for encoding and decoding data using various character encodings, including ASCII, UTF-8, UTF-16, and more. It's a powerful module for handling different character encodings when reading from or writing to files, sockets, or other data sources. The module also allows you to define custom codecs if needed.

The `codecs` module provides the following functions for encoding and decoding data:

- `lookup()` - Lookup a codec by name
- `encode()` - Encode a string to bytes
- `decode()` - Decode bytes to a string
- `open()` - Open a file using a specific encoding
- `register()` - Register a custom codec
- `getencoder()` - Get an encoder function
- `getdecoder()` - Get a decoder function

The following example demonstrates the use of the `codecs` module to encode and decode data:

```
# Example: codecs module

import codecs
import pprint

def test_utf8():

    print("Test UTF-8")

    text = 'Здравейте, хора!'
    print(text)

    encoded = codecs.encode(text, 'utf-8')
    print(encoded)

    decoded = codecs.decode(encoded, 'utf-8')
    print(decoded)

    info = codecs.lookup('utf-8')
    print("lookup() -> ", info)

    print()

def test_utf16():
```



```

print("Test UTF-16")

text = 'Здравейте, хора!'
print(text)

encoded = codecs.encode(text, 'utf-16')
print(encoded)

decoded = codecs.decode(encoded, 'utf-16')
print(decoded)

info = codecs.lookup('utf-16')
print("lookup() -> ", info)

print()

def test_base64():

    print("Test Base64")

    text = 'Здравейте, хора!'
    print(text)

    encoded = codecs.encode(text.encode('utf-8'), 'base64')
    print(encoded)

    decoded = codecs.decode(encoded, 'base64')
    print(decoded.decode('utf-8'))

    info = codecs.lookup('base64')
    print("lookup() -> ", info)

    print()

if __name__ == "__main__":

    test_utf8()
    test_utf16()
    test_base64()

```

If a custom codec is needed, the `codecs` module provides the `register()` function to register a `Codec()` object. The `Codec()` object must implement the `encode()` and `decode()` methods and a search function that returns a `CodecInfo()` object or `None` if the encoding is not supported.

The following example demonstrates the use of the `codecs` module to register a custom codec:

```
# Example: Custom Codec
```

```

import codecs

class ROT13Codec(codecs.Codec):
    # ROT13 Cipher - see https://en.wikipedia.org/wiki/ROT13

    def encode(self, stream, errors='strict'):

        # Result is a list of characters
        encoded = []

        # Iterate over the input stream
        for char in stream:

            # Encode lower case letters
            if 'a' <= char <= 'z':
                offset = ord('a')
                encoded_char = chr(((ord(char) - offset + 13) % 26) + offset)

            # Encode upper case letters
            elif 'A' <= char <= 'Z':
                offset = ord('A')
                encoded_char = chr(((ord(char) - offset + 13) % 26) + offset)

            # Other characters are not encoded
            else:
                encoded_char = char

            # Append encoded character to the resulting list
            encoded.append(encoded_char)

        # Return the encoded bytes and the length of the input stream
        return ''.join(encoded), len(stream)

    def decode(self, stream, errors='strict'):
        # ROT13 is its own inverse
        return self.encode(stream, errors)

    def lookup(self, encoding):

        if encoding == 'rot13':
            codec = codecs.CodecInfo(
                name='rot13',
                encode=self.encode,
                decode=self.decode,
            )

        else:
            codec = None

```

```
return codec
```

```
# Register the codec with the codecs module before using it
codecs.register(ROT13Codec().lookup)

# Usage
encoded_text = codecs.encode("Hello, World!", encoding='rot13')
print(encoded_text) # Output: "Uryyb, Jbeyq!"

decoded_text = codecs.decode(encoded_text, encoding='rot13')
print(decoded_text) # Output: "Hello, World!"
```

9.5. Data Classes

9.5.1. array

An **array** is a data structure that stores a collection of elements of the same type. It is similar to a list, but it is more efficient for storing and performing mathematical operations on large sets of numerical data. It is often used as a lightweight alternative to a list.

```
# Example: Array of data

import array

def test_int():

    # Create an array of integers
    int_array = array.array('i', [1, 2, 3, 4, 5])

    # Convert the array to a bytes object (useful for binary data)
    bytes_data = int_array.tobytes()
    print(bytes_data)

    # Convert a bytes object back to an array
    new_array = array.array('i')
    new_array.frombytes(bytes_data)
    print(new_array)

    print()

def test_unicode():

    # Use unicode characters
    unicode_array = array.array('u', "Здравейте, хора!")

    # Convert the array to a bytes object (useful for binary data)
```

```

bytes_data = unicode_array.tobytes()
print(bytes_data)

# Convert a bytes object back to an array
new_array = array.array('u')
new_array.frombytes(bytes_data)
print(new_array)

print()

if __name__ == "__main__":
    test_int()
    test_unicode()

```

9.5.2. namedtuple

A **namedtuple** is a tuple that has a name for each field. It is similar to a tuple, but it is more readable and easier to use. It is often used as a lightweight alternative to a class.

```

# Example: namedtuple data structure

from collections import namedtuple

# Create a namedtuple (label, fields)
# - The label will be used in the representation of the namedtuple
# - The fields will be used to access the namedtuple attributes

Point = namedtuple('Point', ['x', 'y'])
p = Point(1, 2)

# Test the namedtuple representation
print(p)

# Test the namedtuple attributes
print(p.x)
print(p.y)

# Test the namedtuple index access
print(p[0])
print(p[1])

```

9.5.3. defaultdict

A **defaultdict** is a dictionary that has a default value for each key. It is used in cases where the default value is the same for all keys or when the default value is a function that returns the default value for a key.

You should consider using defaultdict in Python when you need to handle dictionary keys that may

not exist yet and want to provide default values for these keys.

```
# Example: defaultdict data structure

from collections import defaultdict

def test_default_factory(factory):

    d = defaultdict(factory)

    # Test the defaultdict
    try:
        print(d['key1'])
        print(d['key2'])
        print(d['key3'])

    except KeyError as e:
        print("KeyError: {}".format(e))
        assert True

    print()

if __name__ == "__main__":

    default_factories = [
        None,          # Behaves like a regular dictionary
        str,           # Returns an empty string if the key is not found
        int,           # Returns 0 if the key is not found
        float,         # Returns 0.0 if the key is not found
        list,          # Returns an empty list if the key is not found
        tuple,         # Returns an empty tuple if the key is not found
        dict,          # Returns an empty dictionary if the key is not found
        set,           # Returns an empty set if the key is not found
        lambda: 'value', # Returns a default value if the key is not found
    ]

    for default_factory in default_factories:
        test_default_factory(default_factory)
```

9.5.4. queue

A **queue** is a special data structure that stores a collection of elements. It is similar to a list, but it is more efficient for storing and retrieving elements. The **queue** can be implemented as a FIFO (first in first out) or LIFO (last in first out) data structure.

A real-world example of a FIFO queue is a queue of customers at a checkout counter. The customer that comes first is served first.

```

# Example: FIFO queue data structure

from six.moves import queue

def empty_queue(customers):

    # Test the queue
    while True:
        try:
            item = customers.get(block=False)
            print("Get item: {}".format(item))

        except queue.Empty as e:
            print("Empty: {}".format(e))
            break

    print()

def fill_queue(customers):

    items = ['Ivan', 'Dragan', 'Petkan', 'Stoyan']

    for item in items:
        customers.put(item)
        print("Add item: {}".format(item))

    print()

if __name__ == "__main__":

    # Create a FIFO queue
    d = queue.Queue()

    # Fill the queue
    fill_queue(d)

    # Empty the queue
    empty_queue(d)

```

An example of a LIFO queue is a stack of plates. The plate that is placed last is removed first.

```

# Example: FIFO queue data structure

from six.moves import queue

```

```

def empty_queue(customers):

    # Test the queue
    while True:
        try:
            item = customers.get(block=False)
            print("Get item: {}".format(item))

        except queue.Empty as e:
            print("Empty: {}".format(e))
            break

    print()

def fill_queue(customers):

    items = ['Ivan', 'Dragan', 'Petkan', 'Stoyan']

    for item in items:
        customers.put(item)
        print("Add item: {}".format(item))

    print()

if __name__ == "__main__":

    # Create a FIFO queue
    d = queue.LifoQueue()

    # Fill the queue
    fill_queue(d)

    # Empty the queue
    empty_queue(d)

```

Another type of queue is the priority queue. A priority queue is a queue that stores elements with a priority. The priority of an element is used to determine the order in which the elements are removed from the queue. The element with the highest priority is removed first.

```

# Example: FIFO queue data structure

from six.moves import queue

def empty_queue(customers):

    # Test the queue
    while True:

```

```

        try:
            priority, item = customers.get(block=False)
            print("Get {:8} : priority {:5}".format(item, priority))

        except queue.Empty as e:
            print("Empty: {}".format(e))
            break

    print()

def fill_queue(customers):

    priorities = [3, 1, 2, 4]
    items = ['Ivan', 'Dragan', 'Petkan', 'Stoyan']

    for priority, item in zip(priorities, items):
        priority = int(priority)
        customers.put((priority, item))
        print("Add {:8} : priority {:5}".format(item, priority))

    print()

if __name__ == "__main__":

    # Create a FIFO queue
    d = queue.PriorityQueue()

    # Fill the queue
    fill_queue(d)

    # Empty the queue
    empty_queue(d)

```

9.5.5. deque

The **deque** is a special data structure that stores a collection of elements and as the name suggests it is a double-ended queue. It is similar to a list, but it is more efficient for storing and retrieving elements.

As both ends can be used to store elements, the deque can be implemented as a FIFO (first in first out) or LIFO (last in first out) data structure.

CAUTION

Use this module mainly in single-threaded applications. For multi-threaded applications, use the queue module.

```
# Example: FIFO deque data structure
```



```

from collections import deque

def empty_deque(customers):

    # Test the deque
    while True:
        try:

            # Get the item from the right
            item = customers.pop()
            print("Get item: {}".format(item))

        except IndexError as e:
            print("Empty: {}".format(e))
            break

    print()

def fill_deque(customers):

    items = ['Ivan', 'Dragan', 'Petkan', 'Stoyan']

    for item in items:
        customers.appendleft(item)
        print("Add item: {}".format(item))

    print()

if __name__ == "__main__":

    # Create a deque
    d = deque()

    # Fill the deque
    fill_deque(d)

    # Empty the deque
    empty_deque(d)

```

```

# Example: LIFO deque data structure

from collections import deque

def empty_deque(plates):

    # Test the deque

```

```

while True:
    try:

        # Get the item from the right
        item = plates.pop()
        print("Get item: {}".format(item))

    except IndexError as e:
        print("Empty: {}".format(e))
        break

    print()

def fill_deque(plates):

    items = ['Soup plate', 'Salad plate', 'Dinner plate', 'Dessert plate']

    for item in items:
        plates.append(item)
        print("Add item: {}".format(item))

    print()

if __name__ == "__main__":

    # Create a deque
    d = deque()

    # Fill the deque
    fill_deque(d)

    # Empty the deque
    empty_deque(d)

```

9.5.6. `heapq`

The `heapq` is a Python module in the standard library that provides functions for implementing a binary heap queue (min-heap) algorithm. A binary heap is a specialized tree-based data structure that satisfies the heap property. In a min-heap, the parent nodes have values smaller than or equal to those of their children, making it efficient to retrieve and remove the smallest element from the heap.

CAUTION

Use this module mainly in single-threaded applications. For multi-threaded applications, use `PriorityQueue` from the `queue` module.

Some real-world examples of `heapq` are:

- Finding the top N elements in a collection

- Implementing priority queues
- Efficient sorting, where finding the smallest element is required
- Merging multiple sorted inputs
- Graph algorithms, such as Dijkstra's algorithm for finding the shortest path
- Priority based task scheduling

For additional information about `heapq`, see the following resources:

- [heapq](#)
- [Heap \(data structure\)](#)
- [Binary heap](#)
- [Heap sort](#)

```
# Example: heapq as a priority queue

import heapq

def empty_heapq(h):
    # Test the heapq
    while True:
        try:
            print(heapq.heappop(h))

        except IndexError as e:
            print("Empty: {}".format(e))
            break

    print()

def fill_heapq(h):

    # Add items to the heapq
    heapq.heappush(h, 4)
    heapq.heappush(h, 1)
    heapq.heappush(h, 7)

if __name__ == "__main__":

    # Create a heapq
    heap = []
    heapq.heapify(heap)

    # Fill the heapq
    fill_heapq(heap)
```

```
# Empty the heapq
empty_heapq(heap)
```

9.5.7. dataclasses

The `dataclasses` is a module introduced in Python 3.7 that provides a decorator and functions for automatically adding special methods to user-defined classes. It simplifies the creation of classes primarily used to store data, making your code more concise and readable.

Data classes are typically used when you want to create simple classes that mainly serve as data containers, and you want to avoid writing boilerplate code for common methods like `__init__()`, `__repr__()`, and `__eq__()`.

By default data classes are mutable, but you can make them immutable by adding the `frozen=True` keyword argument to the `@dataclass` decorator. This will make the class immutable and hashable, which means it can be used as a dictionary key or as an element in a set.

Data classes can also be inherited from other data classes, and you can add a post-init method that is called after the object has been initialized.

```
# Example: Data Classes (only Python 3.7+)

import dataclasses
from dataclasses import dataclass, field
from typing import List

@dataclass
class DataClass(object):
    name: str
    value: int

    def __post_init__(self):
        self.value = self.value * 2

    def __str__(self):
        return f"{self.name}: {self.value}"

@dataclass
class DataClassWithDefaults(DataClass):

    name: str = "MyData with defaults"
    value: int = 0
    data: List[int] = field(default_factory=list)

    def add_value(self, value):
        self.values.append(value)
```

```

def __str__(self):
    return f"{self.name}: {self.value}, {self.data}"

if __name__ == "__main__":

    my_data = DataClass(name="MyData", value=10)
    print(my_data)

    my_data = DataClassWithDefaults()
    print(my_data)

```

9.5.8. Counter

The **Counter** class is part of the Python **collections** module and is used to count the frequency of elements in an iterable. It's particularly useful when you need to tally the occurrences of items in a list, tuple, or any other iterable object. **Counter** returns a dictionary-like object where elements are keys, and their counts are the corresponding values.

The **Counter()** class can be used in a variety of situations, such as:

- Counting the number of occurrences of each word in a text file
- Counting the most common words in a text file
- Checking for duplicate elements

A real-world example of using the **Counter** class can be found in text analysis and natural language processing. Let's say you're analyzing a large body of text, such as a collection of articles, tweets, or customer reviews, and you want to extract insights about word frequency.

```

# Example: Counter class

from collections import Counter

# Create a Counter object
a = Counter('abcdeabcdabcaba')
b = Counter(reversed("abcdeabcdabcaba"))

# Print the Counter object
print(a)
print(b)

# Print the three most common element
print(a.most_common(3))
print(b.most_common(3))

# Add two Counter objects
c = a + b
print(c)

```

```
# Subtract two Counter objects
d = a - b
print(d)
```

9.5.9. OrderedDict

The `OrderedDict` is useful when you need to maintain the order of elements in a dictionary, which is valuable in scenarios like configuration files, where the order of settings matters, or when you want to implement data structures like ordered queues with key-value pairs. It ensures predictable and consistent ordering when you iterate over the dictionary, which is not guaranteed with the standard dict type.

A real-world example of using the `OrderedDict` class can be found in many frameworks and libraries that use it to store the order of configuration settings. Command line parsers also need to store the order of command line arguments, and they use `OrderedDict` to do so.

```
# Example: OrderedDict

from collections import OrderedDict

# Create an OrderedDict
od = OrderedDict()

# Add elements
od['a'] = 1
od['b'] = 2
od['c'] = 3
od['d'] = 4

# Print the OrderedDict
print(od)

# Move 'c' to the end
od.move_to_end('c')

# Print the OrderedDict
print(od)

# Move 'c' to the start
od.move_to_end('c', last=False)

# Print the OrderedDict
print(od)
```

9.5.10. ChainMap

`ChainMap` is a class in Python's collections module that provides a way to combine multiple

dictionaries or mappings into a single, unified view. It allows you to access the keys and values from all the underlying dictionaries as if they were merged into one. `ChainMap` is particularly useful when you want to create a chain of dictionaries and perform lookups or updates across them efficiently without having to merge the dictionaries physically.

A real-world example of using the `ChainMap` class can be found in web frameworks like Django and Flask. These frameworks use `ChainMap` to manage the configuration settings for an application. The settings are stored in multiple dictionaries, and `ChainMap` is used to combine them into a single view.

```
# Example: ChainMap

from collections import ChainMap

# Create two dictionaries
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}

# Create a ChainMap
chain = ChainMap(dict1, dict2)

# Print the ChainMap
print(chain)

# Print elements
print(list(chain.items()))

# Find value of a key from dict1
print(chain['a'])

# Find value of a key from dict2
print(chain['c'])
```

Chapter 10. Import System

The import system is a mechanism for loading modules and packages. It allows the reuse of code defined in other python files. Each python file defines a module with a unique name. When imported the module is executed and the functions and classes defined in the module are made available to the caller of the import statement.

The following are some of the most common import statements:

- Every python file is a **module**
- Each module lives in its own world called **namespace**
- Each folder containing python file with `init.py` is a **package**
- **Nested packages** are possible and must contain the `init.py` file
- Use `import` to import a module and have access through the module name (e.g. `module.function()`)
- Use `import as` to import a module and have access through the alias (e.g. `alias.function()`)
- Use `from` to import a specific function or class from a module to access it directly

```
# Example: Import Statements

# Importing the package
import product
print('version =', product.__version__)

# Importing a submodule
import product.api.submodule1
print('api.submodule1.my_id =', product.api.submodule1.my_id)

# Importing the entire module with an alias
import product.api.submodule1 as sm1
print('sm1.my_id =', sm1.my_id)

from product.api import submodule1
print('submodule1.my_id =', submodule1.my_id)

from product.api.submodule1 import my_id
print('my_id =', my_id)
```

10.1. Namespace

In Python, a namespace is a container that holds a collection of identifiers (such as variable names, function names, class names, etc.) and maps them to their corresponding objects (such as values, functions, or classes). Namespaces are used to manage and organize the names used in a Python program, preventing naming conflicts and ensuring that each name refers to the correct object.

What happens in a module, stays in a module

— David Beazly, Modules and Packages

```
# import the math module
import math

# import the random module
import random

# Demonstrate the use of namespaces
print(math.pi)
print(random.randint(1, 10))
```

In a Python program, there are four types of namespaces:

- Built-In (accessed using `vars(__builtins__)` or `dir(__builtins__)`)
- Class (accessed using `vars(<class_name>)` or `dir(<class_name>)`)
- Global (accessed using `globals()`)
- Local (accessed using `locals()`)

```
# Example: Namespace of the builtins module

import pprint

class MyClass(object):

    def __init__(self):
        self.a = 1
        self.b = 2

    def instance_method(self):
        pprint.pprint("INSTANCE GLOBALS: {}".format(globals().keys()))
        pprint.pprint("INSTANCE LOCALS: {}".format(locals().keys()))
        print()

    @classmethod
    def class_method(cls):
        pprint.pprint("CLASS GLOBALS: {}".format(globals().keys()))
        pprint.pprint("CLASS LOCALS: {}".format(locals().keys()))
        print()

    def func1(self):

        print("@FUNC1: Before the assignment of a")
        pprint.pprint("FUNC1 GLOBALS: {}".format(globals().keys()))
```

```

pprint.pprint("FUNC1 LOCALS: {}".format(locals().keys()))
print()

a = 1

print("@FUNC1: After the assignment of a")
pprint.pprint("FUNC1 GLOBALS: {}".format(globals().keys()))
pprint.pprint("FUNC1 LOCALS: {}".format(locals().keys()))
print()

def func2():

    print("@FUNC2: Before the assignment of b")
    pprint.pprint("FUNC2 GLOBALS: {}".format(globals().keys()))
    pprint.pprint("FUNC2 LOCALS: {}".format(locals().keys()))
    print()

    b = a + 1

    print("@FUNC2: After the assignment of b")
    pprint.pprint("FUNC2 GLOBALS: {}".format(globals().keys()))
    pprint.pprint("FUNC2 LOCALS: {}".format(locals().keys()))
    print()

func2()

print("@FUNC1: After the call to func2")
pprint.pprint("FUNC1 GLOBALS: {}".format(globals().keys()))
pprint.pprint("FUNC1 LOCALS: {}".format(locals().keys()))
print()

# Builtins level
print("==== Builtins namespace =====")
print()
pprint.pprint("BUILTINS VARS: {}".format(vars(__builtins__).keys()))
print()

# Module level
print("==== Module namespace =====")
print()
pprint.pprint("MODULE GLOBALS VARS: {}".format(globals().keys()))
pprint.pprint("MODULE LOCALS VARS: {}".format(locals().keys()))
print()

# Class level
print("==== Class namespace =====")
print()
pprint.pprint("CLASS VARS: {}".format(vars(MyClass).keys()))
MyClass.class_method()
print()

```

```

# Instance level
print("==== Instance namespace =====")
print()
my_instance = MyClass()
print(vars(my_instance))
my_instance.instance_method()
print()

# Function level
print("==== Function namespace =====")
print()
my_instance.func1()
print(vars(my_instance.func1))
print()

```

10.2. Module Search Path

The module search path is a list of directories that Python searches when importing a module. The path can be inspected or extended by using the `sys.path` list.

```

# Example: Import Search Path

import sys
import pprint

from api import submodule1
print('submodule1.my_id =', submodule1.my_id)

# Print the search path
paths = sys.path
for path in paths:
    print(path)

print()

# Add the current directory to the search path
sys.path.append('.')
paths = sys.path
for path in paths:
    print(path)

```

Typically, Python will start at the beginning of the list of locations and look for a given module in each location until the first match. The script directory is always the first location in the search path. This allows you to import modules from the same directory as the script.

CAUTION

If the user module has the same name as a standard library module, the user module will be loaded first. This can cause unexpected behavior and should be

avoided.

The search path is constructed in the following order:

- The script directory
- The directories in the `PYTHONPATH` environment variable
- The standard library directory using the `sys.prefix` and `sys.exec_prefix` variables (`./lib`)
- The site-packages directory (`./lib/site-packages`)

10.3. Packages

A package is a collection of modules that are grouped together in a directory. A package should have a `__init__.py` file in the directory. The `__init__.py` file is executed when the package is imported. It can be empty or can contain code that initializes the package.

```
package/
├── subpackage1/
│   ├── module1.py
│   └── __init__.py
├── subpackage2/
│   ├── module2.py
│   └── __init__.py
└── subpackage3/
    ├── module3.py
    └── __init__.py
```

The *init* file can be used to aggregate the contents of the package. For example, the following code imports the modules in the package and makes them available to the user using the package name. From the outside the package looks like a single module and thus encapsulates the implementation details of the package.

```
# Define the packet API
__version__ = '1.0.0'

# The dot(.) operator is used to import modules relative to the current package.
# A single dot (.) refers to the current package.
# Two dots (..) refer to the parent package.
# Three dots (...) refer to the grandparent package, and so on.

from .api.submodule1 import func1
from .core.submodule2 import func2
from .gui.submodule3 import func3

# The __all__ variable is used to define the public API of a module or a package.
```

```

__all__ = ['func1', 'func2', 'func3']

# Export decorator
def export(defn):

    # Add the object to the global namespace
    globals()[defn.__name__] = defn

    # Set the object to be exported
    __all__.append(defn.__name__)

    # Return the object
    return defn

# Example of the export decorator
@export
def func4():
    print('func4')

```

The `__init__.py` file is also used to define the API of the package. The API is the set of modules, classes, functions, and variables that are available to the user of the package. The API should only contain the objects that are intended to be used by the user of the package. All other objects should be hidden from the user.

This is achieved by using the `__all__` variable. It is a list of strings that contains the names of the modules, classes, functions, and variables that are available to the user of the package. The `__all__` variable is optional. If it is not defined, all objects in the package are available to the user.

Best practice for packages:

- Split large modules into smaller modules
- Use subpackages to group related modules
- Use one module import statement for several objects from a module
- Use only explicit relative imports for libraries and framework code
- Use absolute imports for application code

10.4. Relative Imports

Relative imports are a way to import modules in Python by specifying their relative location within the project directory structure. Relative imports are useful for keeping your code more modular and without hard-coding the root module name. This allows you to move the module to a different location without having to change the import statement. For example many frameworks use relative imports to import modules from the framework directory.

```

# This is an excerpt from the asyncio module in the Python standard library.

```

```

import sys

# This relies on each of the submodules having an __all__ variable.
from .base_events import *
from .coroutines import *
from .events import *
from .exceptions import *
from .futures import *
from .locks import *
from .protocols import *
from .runners import *
from .queues import *
from .streams import *
from .subprocess import *
from .tasks import *
from .threads import *
from .transports import *

__all__ = (base_events.__all__ +
           coroutines.__all__ +
           events.__all__ +
           exceptions.__all__ +
           futures.__all__ +
           locks.__all__ +
           protocols.__all__ +
           runners.__all__ +
           queues.__all__ +
           streams.__all__ +
           subprocess.__all__ +
           tasks.__all__ +
           threads.__all__ +
           transports.__all__)

```

10.5. Absolute Imports

Absolute imports are a way to import modules in Python by specifying their absolute location within the project directory structure. Absolute imports are useful for keeping your code more concise and readable and it is mostly suitable for application code or packages that are not expected to be moved around.

```

# Example: Absolute imports
import asyncio

# Enforce absolute imports
from __future__ import absolute_import

# Absolute imports
from product.api.submodule1 import func1

```

```
from product.core.submodule2 import func2
from product.gui.submodule3 import func3

# Call all functions
func1()
func2()
func3()
```

10.6. Main Modules

A main module is a module that is executed directly by the Python interpreter. The main module is the first module that is executed in a Python program. The attribute `__name__` from the `globals` dictionary is set to `main`.

```
# Example: Main program

import pprint

# The module attribute __name__ is set to '__main__' when the module is run as the
main program.
print(globals()['__name__'])
```

If a module is imported and executed, the attribute `__name__` is set to the name of the module. This allows you to distinguish between the main module and imported modules. This is useful when you want to execute code only when the module is executed directly by the Python interpreter.

```
# Example: Main program

def func1():
    return 1

def func2():
    return 2

def func3():
    return 3

def test_module():
    assert func1() == 1
    assert func2() == 2
    assert func3() == 3

# The following code prevents the module to be executed when imported as a
```

```
# module. It will be executed only when run as a script directly by the Python
# interpreter. If omitted the module will be executed on each import.
```

```
if __name__ == '__main__':
    test_module()
```

10.7. Main Packages

A main package is a package that has the file `main.py` in the package directory. This makes the package directory executable and marks explicitly the entry point of the package. This is very useful in case the package offers a command-line interface that can be implemented in this file.

```
import sys
import os.path

def main():
    progname = sys.argv[0]
    sys.argv[:] = sys.argv[1:]
    sys.path.insert(0, os.path.dirname(progname))

    print('sys.argv =', sys.argv)

if __name__ == "__main__":
    main()
```

10.8. Import Process

As everything else in Python, modules are objects. This means that they can be assigned to a variable, passed as an argument to a function, or returned from a function. This is useful when you want to dynamically import a module or when you want to import a module conditionally.

```
from types import ModuleType

# Create a new module object
test = ModuleType('test')
print("test.__dict__:", test.__dict__)

# Add some attributes to the module
test.__dict__['a'] = 1
test.__dict__['b'] = 2
test.__dict__['c'] = 3

# Print the module's namespace
print("test.__dict__:", test.__dict__)
```


A package in this context is just a module object whose `__path__` attribute is set to a list of directory names and the `__package__` attribute is set to the name of the package.

The import statement actually creates a module object and assigns it to a variable. The module object can then be used to access the functions and classes defined in the module.

The import process can be summarized in the following steps:

- The import statement is executed
- It searches for the module in the module search path
- If the module is found and it is not already loaded, then the module object is created
- The code in the module is compiled and then executed
- The module object is cached by storing it in the `sys.modules` dictionary
- The module object is assigned to the variable

```
# Example: Very naive implementation of the import statement
```

```
import marshal
import os
import sys
import types
```

```
def _import(module_name):
```

```
    # Check if the module is already imported
    if module_name in sys.modules:
        print('Module already imported')
        return sys.modules[module_name]
```

```
    # Read the source code from the file
    sourcepath = module_name + '.py'
    with open(sourcepath, 'r') as f:
        sourcecode = f.read()
```

```
    # Compile the source code to bytecode
    code = _compile(sourcecode, sourcepath, module_name)
```

```
    # Create a new module object
    module = types.ModuleType(module_name)
    module.__file__ = sourcepath
```

```
    # Cache the module object in sys.modules
    sys.modules[module_name] = module
```

```
    # Execute the bytecode in the module's namespace
    exec(code, module.__dict__)
```

```

# Return the module object
return sys.modules[module_name]

def _compile(sourcecode, sourcepath, module_name):
    """ Compile the source code of a module to bytecode. """

    MAGIC_NUMBER = (3439).to_bytes(2, 'little') + b'\r\n'

    # Compile the source code to bytecode
    code = compile(sourcecode, sourcepath, 'exec')

    # The following code is optional

    # Serialize the code object and write it to a .pyc file
    with open(module_name + '.pyc', 'wb') as f:
        mtime = os.path.getmtime(sourcepath)
        size = os.path.getsize(sourcepath)
        f.write(MAGIC_NUMBER)
        f.write(int(mtime).to_bytes(4, sys.byteorder))
        f.write(int(size).to_bytes(4, sys.byteorder))
        marshal.dump(code, f)

    return code

if __name__ == '__main__':
    s1 = _import('./product/api/submodule1')
    s2 = _import('./product/api/submodule1')
    print(s1.my_id, s2.my_id)

```

10.9. Importlib

The importlib module provides functions for programmatically importing modules. The importlib module also provides functions for creating custom importers. An importer is an object that is responsible for finding and loading modules.

10.10. Naming Conventions

The following rules apply for module names:

- Module names should be lower case with words separated by a single underscores
- Use leading underscore for private modules
- Don't use names that match common standard libraries
- Acronyms and abbreviations should not be used
- Non-ASCII characters should not be used

Chapter 11. Compatibility

This section covers the compatibility of Python with other languages and platforms. There are two scenarios where compatibility is important:

- Interoperability between Python and other languages
- Compatibility between different versions of Python

The following are some of the most common compatibility rules and tools:

- Writing Python code that is compatible with Python 2 and Python 3
- Using the `__future__` and `future`
- Using the `six`

11.1. Syntax

Python 2 and Python 3 have different syntaxes for some features. When writing code that should be compatible with both Python 2 and Python 3, you should follow the following rules:

- Use only **explicit relative imports** or **absolute imports**
- Use only the `print()` function
- Use `raise exception` instead of `raise exception as e`
- Use `except Exception as e` instead of `except Exception, e`
- Be careful with the division operator (/). In Python 2 it is an integer division operator, in Python 3 it is a float division operator

Some changes require more than just a simple syntax change. For example the following changes require additional tools to make the code compatible:

- **xrange() vs range()**: In Python 3 the `range()` function returns an iterator instead of a list. The same behavior can be achieved in Python 2 by using the `xrange()` function.
- **unicode vs str** : In Python 2, the `str` type represents a byte string, and the `unicode` type represents a Unicode string. In Python 3, the `str` type represents a Unicode string, and the `bytes` type represents a byte string.
- **raw_input() vs input()** : The `input` in Python 3 is equivalent to `raw_input` in Python 2
- **__metaclass__ vs metaclass=** : Python 3 uses the `metaclass` keyword argument instead of the `__metaclass__` attribute
- **map, filter, reduce, zip**: These built-in functions return iterators in Python 3 instead of lists

11.2. Using `__futures__`

The most common way to write code that is compatible with Python 2 and Python 3 is to use the `__future__` library. It provides backwards compatibility for features that have changed in Python 3.

The most commonly used features are:

- division (division is float division)
- print_function (ensure that print() is a function)
- unicode_literals (all string literals are unicode strings)

```
# Example: __future__ module

# Most commonly used imports from the __future__ module
from __future__ import print_function
from __future__ import unicode_literals
from __future__ import division

print "hello world" # Shall indicate an error if the future import is not present
```

11.3. Using **six**

The "six" library in Python is a compatibility library that helps you write Python 2 and Python 3 compatible code. It's mainly used when you need your code to work across both Python 2 and Python 3 versions, which is less common now because Python 2 reached its end of life on January 1, 2020, and Python 3 is the standard.

```
# Example: six module

# The most commonly used imports from the six module
from six.moves import input # Import the input function from the six.moves module
from six import with_metaclass # Import the with_metaclass function from the six
module

class Meta(type):
    def __new__(cls, name, bases, dct):
        print("Allocating memory for class", name)
        return super(Meta, cls).__new__(cls, name, bases, dct)

    def __init__(cls, name, bases, dct):
        print("Initializing class", name)
        super(Meta, cls).__init__(name, bases, dct)

class Base(with_metaclass(Meta)):
    def __init__(self):
        print("Initializing instance", self)
        super(Base, self).__init__()
```

```
class Derived(Base):
    def __init__(self):
        print("Initializing instance", self)
        super(Derived, self).__init__()

if __name__ == "__main__":
    # Use the six.moves.input function with Python 2 and 3
    input("Press Enter to continue...")

    # Use the six.with_metaclass function with Python 2 and 3
    print("Creating instance of Base")
    b = Base()
    print("Creating instance of Derived")
    d = Derived()
    print("Finished")
```

Chapter 12. Testing

Python has a rich ecosystem of testing frameworks and tools. The following are some of the most popular testing frameworks and tools:

- [unittest](#)
- [doctest](#)
- [unittest.mock](#)
- [pytest](#)
- [Coverage](#)
- [Hypothesis](#)
- [tox](#)

It is beyond the scope of this book to cover all of these frameworks and tools in detail. The reader is encouraged to explore these frameworks and tools and choose the one that best fits their needs. The following sections provide a brief overview of the test frameworks and tools mentioned above.

12.1. unittest

unittest is a testing framework that is part of the Python standard library. It provides a rich set of features for writing and running tests, including: fixtures, parameterized tests, test discovery, test collection, test execution, test reporting, and test debugging. See [unittest](#) for more information.

Key features of unittest:

- **Test Discovery:** unittest can automatically discover and run test cases within your project. It scans modules and packages for test cases, making it easy to organize and execute tests.
- **Test Fixtures:** unittest provides methods for setting up and tearing down test fixtures. You can use `setUp()` to prepare the environment before a test and `tearDown()` to clean up afterward. This ensures that tests are isolated and repeatable.
- **Test Case Classes:** Test cases are organized into classes derived from `unittest.TestCase`. Each method in the class whose name starts with "test" is treated as an individual test case.
- **Assertions:** unittest offers a variety of assertion methods (e.g., `assertEqual()`, `assertTrue()`, `assertRaises()`) to check expected outcomes in your tests. These assertions help you validate that your code is functioning correctly.
- **Test Discovery:** You can use test loaders to customize how tests are discovered and loaded. This is useful for selecting specific test cases or running tests with different patterns.
- **Test Result Reporting:** unittest provides detailed test result reporting, including information about passed, failed, and skipped tests. This helps you identify and address issues quickly.
- **Test Skipping:** You can skip specific tests using the `@unittest.skip` decorator or conditional skipping with `@unittest.skipIf` and `@unittest.skipUnless`. This is useful for excluding tests under certain conditions.
- **Parameterized Tests:** unittest allows you to create parameterized tests using test data provided

in iterable forms. This enables you to test a function or method with multiple input values easily.

- **Test Suites:** You can group related test cases into test suites using `unittest.TestSuite`. Test suites enable you to run a collection of tests as a single unit, making it easier to organize and execute tests.
- **Custom Test Loaders and Result Classes:** You can create custom test loaders to discover tests in unconventional ways and customize test result classes to modify result reporting.
- **Subtest Support:** Python's `unittest` supports subtests, allowing you to run multiple assertions within a single test function. This is useful for tracking multiple conditions in a single test case.
- **Skipping Test Classes:** You can skip entire test classes by decorating the class with `@unittest.skip` or skipping them conditionally with `@unittest.skipIf` and `@unittest.skipUnless`.
- **Test Timeouts:** You can set a timeout for individual test cases to ensure they don't run indefinitely. This is helpful for detecting test cases that get stuck.
- **Mocking:** `unittest.mock` is a testing framework that is part of the Python standard library. It provides a rich set of features for mocking objects and functions, including: mocking, patching, stubbing, spying, and faking. See [unittest.mock](#) for more information.

12.2. pytest

pytest is a popular testing framework that makes it easy to write simple and scalable tests. It provides a rich set of features for writing and running tests, including: fixtures, parameterized tests, test discovery, test collection, test execution, test reporting, and test debugging.

Key features of pytest:

- **Automated Test Discovery:** Pytest automatically discovers and runs test functions, classes, and modules. You don't need to manually specify each test to run, which simplifies test management.
- **Fixture Support:** Pytest provides a robust fixture mechanism, allowing you to set up and tear down resources needed for your tests. Fixtures enhance code reuse and ensure that setup and teardown logic is clean and maintainable.
- **Parameterized Testing:** You can use the `@pytest.mark.parametrize` decorator to run the same test function with multiple sets of input data, simplifying the testing of various scenarios.
- **Powerful Assertions:** Pytest uses Python's `assert` statement for making assertions, and it provides detailed failure information, making it easy to diagnose issues in failing tests.
- **Introspection and Debugging:** Pytest offers extensive introspection capabilities. You can use the `--pdb` option to drop into a debugger when a test fails, making debugging straightforward.
- **Plugins and Extensibility:** Pytest has a rich ecosystem of plugins that extend its functionality. You can write your custom plugins or use existing ones to tailor pytest to your specific testing needs.
- **Test Markers and Custom Labels:** You can use custom markers to label and categorize tests. This allows you to run specific groups of tests based on their labels, improving test organization.
- **Parallel Test Execution:** Pytest supports parallel test execution, taking advantage of multiple

cores for faster test runs. This is particularly useful for large test suites.

- **Integration:** Pytest can be used alongside other testing frameworks, allowing you to gradually transition to pytest without rewriting all your tests.
- **Rich Reporting:** Pytest generates informative test reports in various formats, including console, JUnit XML, and HTML. These reports provide a summary of test results and help identify issues quickly.
- **Flexible Test Discovery:** Pytest allows you to customize test discovery by specifying naming conventions, directories, and filtering options. You can also exclude certain tests or modules from execution.
- **Fixture Scopes:** Fixtures can have different scopes (function, class, module, session), allowing you to control how long a resource is used and shared among tests.
- **Parametrized Fixtures:** Fixtures themselves can be parametrized, enabling dynamic fixture generation based on test inputs.
- **Pythonic Test Organization:** You can organize your tests in a way that follows Python's module and package structure.
- **Continual Development:** Pytest is actively maintained and developed, ensuring that it remains a relevant and up-to-date testing framework.

12.3. Coverage

Coverage is a tool for measuring code coverage of Python programs. Code coverage is a metric that helps you understand how much of your code is executed by your tests. It indicates which parts of your code are tested and which parts are not. The goal of coverage analysis is to identify areas of your code that may require additional testing. See [Coverage](#) for more information.

- **Code Coverage Metrics:** The coverage tool calculates various code coverage metrics, including statement coverage (the percentage of executable statements executed), branch coverage (the percentage of decision branches taken), and more. These metrics provide insights into the thoroughness of your testing.
- **Integration:** coverage seamlessly integrates with popular Python testing frameworks like pytest, unittest, and nose. It can be used with these frameworks to measure code coverage during test execution.
- **Tracking Executed Code:** coverage works by tracking which lines of code are executed during test runs. It instruments your Python code to collect this information.
- **HTML and Console Reports:** coverage generates detailed coverage reports in both HTML and console formats. These reports show which lines of code were covered by tests and which were not. HTML reports are particularly useful for visualizing coverage data.
- **Branch Coverage:** In addition to statement coverage, coverage can also track branch coverage. Branch coverage measures whether both true and false branches of conditional statements have been executed.
- **Selective Coverage:** You can use coverage to measure coverage for specific parts of your code by including or excluding modules, packages, or specific files from analysis. This allows you to focus on critical areas of your codebase.

- **Continuous Integration (CI) Integration:** Many CI/CD (Continuous Integration/Continuous Deployment) systems integrate with coverage to provide coverage reports as part of the build process. This helps ensure that code coverage is monitored automatically in the development workflow.
- **Cross-Platform:** coverage is cross-platform and works on various operating systems, making it suitable for different development environments.
- **Community Support:** coverage has an active community, and it's widely used in the Python ecosystem. This means you can find extensive documentation, tutorials, and community support if you encounter issues.

12.4. Hypothesis

Hypothesis is a library for property-based testing. It can be used to generate test data and assertions. It can be used with pytest, unittest, and doctest. Property-based testing is an approach where you define general properties or rules that your code should satisfy, and the testing framework generates a wide range of test cases automatically to check if those properties hold true. See [Hypothesis](#) for more information.

Key features of Hypothesis:

- **Generative:** Hypothesis generates input data for your tests. It explores a range of input values to discover edge cases and unusual scenarios that you might not have considered
- **Properties:** Instead of writing specific test cases with known inputs and expected outputs, you define properties that your code should adhere to. Hypothesis then tests these properties with a wide variety of inputs
- **Strategies:** Hypothesis provides a wide range of testing strategies for various types of data, including integers, floats, text, lists, and more. You can also create custom strategies to suit your specific needs.
- **Integration:** Hypothesis can be integrated with popular testing frameworks like pytest, unittest, and nose. This allows you to seamlessly incorporate property-based testing into your existing testing workflow.
- **Root Cause:** When Hypothesis finds a failing test case, it attempts to find the simplest possible input that reproduces the failure. This helps you identify the root cause of issues quickly.

12.5. tox

The Tox library is a popular tool in the Python ecosystem for managing and automating the testing of Python packages across multiple environments. It helps ensure that your Python package works correctly across various Python versions, interpreters, and configurations.

Key features of Tox:

- **Cross-Environment Testing:** Tox allows you to define multiple testing environments, each with a specific Python version, interpreter, and configuration. This enables you to test your code across a wide range of scenarios.

- **Isolated Virtual Environments:** Tox creates isolated virtual environments for each testing environment you specify. This ensures that your tests run in clean, reproducible environments, avoiding conflicts with system-wide Python installations.
- **Automated Testing:** Tox automates the entire testing process. You can run all tests for your project, including unit tests, integration tests, and more, with a single command: `tox`.
- **Dependency Management:** Tox can handle the installation of project dependencies and testing dependencies. This ensures that the correct dependencies are installed in each virtual environment.
- **Parallel Test Execution:** Tox can execute tests in parallel, taking advantage of multicore processors to speed up testing, making it suitable for large projects.
- **Integration with Testing Frameworks:** Tox integrates seamlessly with popular testing frameworks like `pytest`, `nose`, `unittest`, and more. You can specify your preferred testing framework in the configuration.
- **Continuous Integration (CI) Integration:** Tox is commonly used in CI/CD (Continuous Integration/Continuous Deployment) pipelines. It ensures that your code passes tests in various environments, helping you catch issues early.
- **Custom Environments:** Besides standard Python versions and interpreters, Tox allows you to define custom testing environments, making it versatile for different project requirements.
- **Plugin System:** Tox features a plugin system that allows you to extend its functionality. You can create custom plugins or use existing ones to tailor Tox to your specific needs.
- **Clean and Isolated Execution:** Tox ensures that tests do not interfere with each other or with your development environment. This isolation helps maintain consistent testing results.
- **Documentation and Reporting:** Tox provides documentation and reporting features that help you understand the test results across different environments and configurations.

References

Chapter 1. Bibliography

- Clean Code: A Handbook of Agile Software Craftsmanship
- Fundamentals of Software Architecture: An Engineering Approach
- Dive into Design Patterns
- The Complete Python Manual: Learn how to master Python and expand your programming skills
- Python Pocket Reference
- Mastering Python
- Fluent Python: Clear, Concise, and Effective Programming
- Python Cookbook: Recipes for Mastering Python 3
- Python Tricks: A Buffet of Awesome Python Features
- Advanced Python Programming: Build high performance, concurrent, and multi-threaded apps with Python using proven design patterns
- Learn Python 3 the Hard Way: A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code
- Advanced Python Programming: Build high performance, concurrent, and multi-threaded apps with Python using proven design patterns
- Effective Python: 59 Specific Ways to Write Better Python
- Python Tricks: A Buffet of Awesome Python Features
- Effective Computation in Physics: Field Guide to Research with Python
- The big book of small Python projects
- Competitive Programming in Python
- Python Penetration Testing Essentials
- Learning Penetration Testing with Python
- Python Data Analytics
- Numerical Python: A Practical Techniques Approach for Industry

Chapter 2. Links

2.1. Beginner

- <https://www.python.org/>
- <https://www.python.org/dev/peps/pep-0008/>
- <https://www.python.org/dev/peps/pep-0020/>
- <https://www.python.org/dev/peps/pep-0257/>
- <https://www.w3schools.com/python/>
- <https://realpython.com/>
- <https://www.geeksforgeeks.org/python-programming-language/>
- <http://p-nand-q.com/python/lambda.html>
- <https://www.educative.io/edpresso/what-is-mro-in-python>
- <http://www.srikanthtechnologies.com/blog/python/mro.aspx>
- <https://www.toptal.com/python/python-class-attributes-an-overly-thorough-guide>
- <https://www.residentmar.io/2019/07/07/python-mixins.html>
- <https://www.pythontutorial.net/python-oop/python-mixin>
- <https://tinyurl.com/pylinters>
- <https://codilime.com/blog/python-code-quality-linters/>
- <https://dsstream.com/improve-your-python-code-quality/>

2.2. Intermediate

- <https://python101.pythonlibrary.org/index.html>
- <https://book.pythontips.com/en/latest/index.html>
- <https://realpython.com/python-iterators-iterables/>
- <https://www.pythontutorial.net/advanced-python/python-iterator-vs-iterable/>
- https://commons.wikimedia.org/wiki/File:W3sDesign_Dependency_Injection_Design_Pattern_UML.jpg
- <https://maximilienandile.github.io/2019/10/20/Dependency-injection-what-is-it-how-to-do-it-in-Java-and-why-to-use-it/>
- <https://www.youtube.com/watch?v=IKD2-MAkXyQ>
- <https://towardsdatascience.com/comprehensive-guide-to-python-logging-in-the-real-world-part-1-8762d5aa76da>
- <https://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.html#:~:text=Exceptions%20provide%20the%20means%20to,lead%20to%20confusing%20spaghetti%20code>

- <https://www.javatpoint.com/python-exception-handling>
- <https://guicommits.com/how-to-structure-exception-in-python-like-a-pro/>
- <https://docs.python.org/2.7/tutorial/errors.html#tut-userexceptions>
- <https://docs.python.org/3.10/tutorial/errors.html#tut-userexceptions>
- <https://realpython.com/pytest-python-testing/>
- <https://pytest-with-eric.com/comparisons/pytest-vs-unittest/>
- https://python-future.org/compatible_idioms.html
- <https://realpython.com/python-import/>
- <https://peps.python.org/pep-0366/>
- <https://www.stakater.com/post/a-practical-introduction-to-ci-cd-ct-in-devopscontinuous-integration-ci-continuous-deployment>
- <https://kruschecompany.com/v-model-software-development-methodology/>
- <https://www.softsuave.com/blog/spiral-model-in-sdlc/>
- <https://www.youtube.com/watch?v=mp22SDTnsQQ>

2.3. Expert

- <https://python-3-patterns-idioms-test.readthedocs.io/en/latest/index.html>
- <https://www.dabeaz.com/tutorials.html>
- https://python-future.org/compatible_idioms.html
- <https://github.com/jsbueno/metapython/blob/main/static.py>
- <https://stackoverflow.com/questions/12356713/aspect-oriented-programming-aop-in-python>
- <https://softwareengineering.stackexchange.com/questions/99433/aop-concepts-explained-for-the-dummy>
- <https://peps.python.org/pep-0420/>
- <https://packaging.python.org/en/latest/guides/packaging-namespaces-packages/>
- <https://realpython.com/python-walrus-operator/>
- https://www.python-course.eu/python3_generators.php
- <https://www.youtube.com/watch?v=EnSu9hHGq5o>
- <https://www.python.org/dev/peps/pep-0255>
- https://www.pythonlikeyoumeanit.com/Module2_EssentialsOfPython/Generators_and_Comprehensions.html
- <https://www.geeksforgeeks.org/coroutine-in-python/>
- https://www.python-course.eu/python3_generators.php
- <https://www.youtube.com/watch?v=EnSu9hHGq5o>

2.4. Libraries

- <https://pymotw.com/2/contents.html>
- <https://pymotw.com/3/index.html>