Python

PCPP1

June 18, 2024
Antonia Frey

Contents

1 Fundamentals

1.1 Python

Python is a high-level, interpreted programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991. Python's design philosophy emphasizes code readability and productivity, making it an ideal language for both beginners and experienced programmers alike.

1.1.1 Versions

Python 2 and Python 3 represent two major versions of the Python programming language. Python 2, released in 2000, served as the primary version for many years until its end of life in 2020. Python 3, introduced in 2008, addressed design flaws and inconsistencies present in Python 2. Python 3 is actively maintained and recommended for all new projects, while Python 2 is no longer officially supported.

1.1.2 Hello World

This Python code will print ”Hello, World!” to the console when executed. It's a simple yet classic introductory example often used to demonstrate the basic syntax of a programming language.

```python
# Hello World Program
print("Hello, World!")
```

- **# Hello World Program**: This line is a comment. It's meant to provide a brief description or title for the program. It does not affect the execution of the code but helps other developers understand the purpose of the program.
- **print(”Hello, World!”):** This line is the actual Python code. It prints the string ”Hello, World!” to the console. The `print()` function is used to display output.

1.1.3 Official Python Documentation

The official Python documentation is hosted online at docs.python.org. It serves as the primary source of documentation for the Python programming language and its standard library.

1.1.4 Applications

Python is a versatile programming language widely used across various domains due to its simplicity, readability, and extensive libraries.

- **Web Development**: Widely used for web development, with frameworks like Django and Flask for building web applications, APIs, and dynamic websites.
- **Data Science and Machine Learning**: Popular choice for data analysis, machine learning, and artificial intelligence projects, thanks to libraries like NumPy, pandas, scikit-learn, TensorFlow, and PyTorch.
- **Automation and Scripting**: Commonly used for automation tasks, scripting, and system administration, thanks to its ease of use and cross-platform compatibility.
- **Scientific Computing**: Used in scientific computing and computational modeling, with libraries like SciPy and Matplotlib for numerical calculations, data visualization, and plotting.

1.2 Data Types

There are several built-in data types that are fundamental to the language's core functionality. Here's a brief overview of the main Python data types:

1.2.1 Numeric Types

```
my_int: int = 10
my_float: float = 3.14
my_complex: complex = 2 + 3j
```

- **Integer**: Integer numbers, representing whole numbers
- **Float**: Floating-point numbers
- **Complex**: Complex numbers with a real and imaginary part

1.2.2 Strings

```
first_string: str = "Hello, World!"
second_string: str = 'Hello, World!'
```

- **String**: Sequences of Unicode characters, immutable, enclosed in single or double quotes

1.2.3 Collections

A collection is a single variable used to store multiple values.

```
my_list: list[int] = [1, 2, 3, 4, 5]
my_tuple: tuple[int] = (1, 2, 3)
my_set: set[int] = {1, 2, 3}
my_frozenset: frozenset[int] = frozenset({1, 2, 3})
```

- **List = []**: Ordered collection of items, mutable, duplicates are ok
- **Tuple = ()**: Ordered collection of items, immutable, duplicates are ok
- **Set =** {}: Unordered collection of unique items, mutable (add/remove items), no duplicates
- **Frozenset = frozenset(**{}**)**: Immutable version of a set

```
my_dict: dict[str, int or str] = {
    "name": "John",
    "age": 30,
    "city": "New York"
}
```

- **Dictionary**: Collection of key-value pairs, mutable and unordered collection

1.2.4 Boolean Type

```
my_boolean: bool = True  # Boolean value
```

- **Boolean**: Represents truth values `True` or `False`.

1.2.5 None Type

```
my_none = None
```

- **NoneType**: Represents the absence of a value or null value.

These are the primary built-in data types. Understanding these data types and their properties is essential for effective programming. Additionally, Python's dynamic typing system allows variables to change types dynamically, enhancing flexibility and ease of use.

1.2.6 Comparison of List, Tuple, and Set

Property	**List**	**Tuple**	**Set**
Creation	`my_list = [1,2,3]`	`my_tuple = (1,2,3)`	`my_set = {1,2,3}`
Ordering	Ordered (Maintains insertion order)	Ordered (Maintains insertion order)	Unordered (No guaranteed order)
Duplicates	Allows	Allows	No duplicates (Collection of unique elements)
Mutability	Mutable	Immutable	Mutable
Access Time	O(1) (Constant time)	O(1) (Faster than list)	O(1) (Faster than list or tuple)

1.2.7 Dictionaries

A dictionary is a built-in data type that allows you to store a collection of key-value pairs. It's a highly flexible data structure that provides efficient lookups, insertions, and deletions.

- **Key**: Each key in a dictionary must be unique and immutable (such as strings, numbers, or tuples). Keys are used to access the corresponding values.
- **Value**: The associated value for each key in the dictionary. Values can be of any data type, including lists, tuples, dictionaries, or even other dictionaries.

A value in a dictionary can be checked using the `in` operator. However, by default, the `in` operator checks for keys in the dictionary, not values. To check if a value exists in a dictionary, the `values()` method can be utilized to obtain a view of the dictionary's values.

```
phonebook = {"Alice": "123456", "Bob": "789012", "Charlie": "345678"}

if "Bob" in phonebook:
    print("Bob's number is:", phonebook["Bob"])
else:
    print("Bob is not in the phonebook.")


if "345678" in phonebook.values():
    print("The number 345678 exists in the phonebook.")
```

1.2.8 Type Conversion Methods

Type conversion methods are functions or operations used to convert data from one type to another. These methods allow you to change the data type of a variable or value.

- `int()`: Converts a value to an integer.
- `float()`: Converts a value to a floating-point number.
- `str()`: Converts a value to a string.
- `list()`: Converts an iterable to a list.
- `tuple()`: Converts an iterable to a tuple.
- `set()`: Converts an iterable to a set.
- `bool()`: Converts a value to a boolean.

These type conversion methods are essential for manipulating and transforming data between different types as needed in various programming scenarios.

1.2.9 List Comprehension

List comprehension is a concise way to create lists. It allows to generate lists based on existing iterables such as lists, strings, or range objects, with a compact and readable syntax.

```python
squares = [x**2 for x in range(10)]
print(squares)
# Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = [x for x in numbers if x % 2 == 0]
print(even_numbers)
# Output: [2, 4, 6, 8, 10]
```

```python
# Define the dimensions of the matrix
rows = 3
columns = 3

# Create a 2D array (matrix) using list comprehension
matrix = [[0 for _ in range(columns)] for _ in range(rows)]
```

1.3 Modules, Packages, and Namespaces

- **Modules**
  Individual files containing Python code, such as functions, classes, and variables. They allow you to organize your code into separate files for better maintainability and reusability. You can import modules in other scripts to use the code they contain.
- **Packages**
  Directories containing one or more modules and an `__init__.py` file. They provide a way to organize modules into a hierarchical structure. Packages allow you to manage larger projects more effectively by grouping related functionality together. You can import packages in your scripts to access the modules and their contents.
- **Namespace**
  Namespaces are closely related to modules and packages. Each module and package has its own namespace, which serves as a container for the names defined within it. This ensures that names defined in one module or package do not clash with names in another module or package. When you import the `math` module as `m`, you're creating a namespace `m` which contains all the names defined in the `math` module.

```python
# Importing my_module from my_package
from my_package import my_module

# Importing math module
import math

# Using constants and functions from math module
print("The value of pi is:", math.pi)
print("The square root of 16 is:", math.sqrt(16))

# Using a function from my_module
my_module.say_hello()
```

1.3.1 Order of imports

PEP 8 suggests a specific order to import different types of modules:

1. Standard library imports
2. Related third-party imports
3. Local application or library-specific imports

1.3.2 The `__name__` Variable

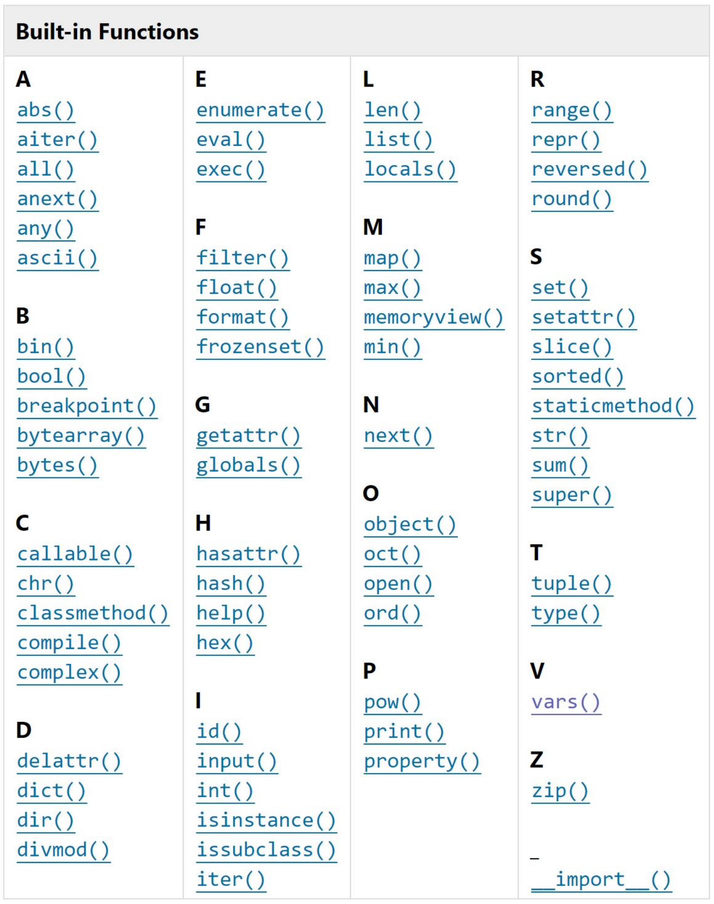
When Python runs a script, it sets the special variable `__name__` to "__main__" for the script that is being executed directly. If the script is imported as a module into another script, then `__name__` is set to the name of the module.

```python
if __name__ == "__main__":
    # This code block will only run if the script is executed directly
    # It won't run if the script is imported as a module into another script
```

1.4 Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed and described here in alphabetical order.

Built-in functions are functions that are available in the Python interpreter without the need to import any external modules. They are part of the core Python language and provide basic functionality that is commonly used in programming tasks. Built-in functions span a wide range of operations, from basic arithmetic and type conversion to more specialized tasks like input/output operations and data manipulation.



Built-in Functions

A
abs()
aiter()
all()
anext()
any()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

Figure 1: Built-in Functions

The `len()` function in Python is a built-in function that returns the length of an object. It's a versatile and commonly used tool that provides a convenient way to determine the number of elements in various data structures, such as strings, lists, tuples, dictionaries, and more.

```python
# Example use of len() with a list
my_list = [1, 2, 3, 4, 5]
list_length = len(my_list)
print("Length of the list:", list_length)

# Example use of len() with a string
my_string = "Hello, World!"
print("Number of charactersin the string:", len(my_string))
```

1.4.1 Map, Filter and Reduce

The `map()`, `filter()`, and `reduce()` functions are powerful tools in functional programming, allowing for concise and efficient data transformations and computations over iterables. They help streamline code and make it more expressive by abstracting away common patterns of iteration and computation.

- **`map()`**
  - The `map()` function applies a specified function to each item in an iterable (such as a list) and returns a new iterator containing the results.
  - Syntax: `map(function, iterable)`
  - Example:

    ```
    numbers = [1, 2, 3, 4, 5]
    doubled = map(lambda x: x * 2, numbers)
    print(list(doubled))  # Output: [2, 4, 6, 8, 10]
    ```

  - In this example, the `lambda x:  x * 2` function is applied to each item in the `numbers` list, doubling each value.

- **`filter()`**
  - The `filter()` function constructs an iterator from elements of an iterable (such as a list) for which a specified function returns `True`.
  - Syntax: `filter(function, iterable)`
  - Example:

    ```
    numbers = [1, 2, 3, 4, 5]
    evens = filter(lambda x: x % 2 == 0, numbers)
    print(list(evens))  # Output: [2, 4]
    ```

  - In this example, the `lambda x:  x % 2 == 0` function filters out only the even numbers from the `numbers` list.

- **`reduce()`**
  - The `reduce()` function is used to apply a rolling computation to sequential pairs of values in an iterable, producing a single accumulated result.
  - Syntax: `reduce(function, iterable[, initializer])`
  - Example:

    ```
    from functools import reduce

    numbers = [1, 2, 3, 4, 5]
    total = reduce(lambda x, y: x + y, numbers)
    print(total)  # Output: 15
    ```

  - In this example, the `lambda x, y:  x + y` function is applied cumulatively to the items of the `numbers` list, resulting in the sum of all elements.
  - In Python 3, the `reduce()` function is available in the `functools` module, so you do need to import it if you want to use `reduce()`. However, in Python 2, `reduce()` was a built-in function and didn't require an import.

1.4.2 Iterator Functions

Iterators are objects that implement the iterator protocol, which consists of two methods: `__iter__()` and `__next__()`. Iterators are used to iterate over a sequence of elements, such as lists, tuples, dictionaries, and custom objects.

- `__iter__()`: This method returns the iterator object itself. It's called when you create an iterator using the `iter()` function or when an iterator is requested implicitly, such as in a `for` loop.
- `__next__()`: This method returns the next item from the iterator. It's called repeatedly to retrieve successive items from the iterator. When there are no more items to return, it raises a `StopIteration` exception.

```python
class MyIterator:
    def __init__(self, limit):
        self.limit = limit
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < self.limit:
            value = self.current
            self.current += 1
            return value
        else:
            raise StopIteration()


# Using the custom iterator
iterator = MyIterator(5)
for item in iterator:
    print(item)
```

This example demonstrates how to create a simple custom iterator in Python that follows the iterator protocol. It generates values lazily, allowing for efficient memory usage, especially when dealing with large datasets or infinite sequences. The `__iter__()` method is called when an object is used in an iterable context, such as in a for loop, to obtain an iterator for the object. Iterators support the `next()` function, which is used to retrieve the next item from the iterator.

Several Python built-in functions return iterators or objects that can be iterated over. Some of the most commonly used ones include:

- **`iter()`**: This built-in function returns an iterator object for the given iterable. If the object is already an iterator, it returns itself. Otherwise, it calls the `__iter__()` method of the object to obtain an iterator.
- **`enumerate()`**: Returns an iterator that yields tuples containing a count (starting from zero by default) and the values obtained from iterating over the iterable.
- **`map()`**: Applies a given function to each item of an iterable (like a list or tuple) and returns an iterator that yields the results.
- **`filter()`**: Constructs an iterator from those elements of the iterable for which the function returns `True`.
- **File objects**: When reading from a file in Python, file objects return iterators over lines of the file. For example, when using a `for` loop to iterate over lines in a file, each iteration returns the next line from the file.
- **`range()`**: In Python 3, `range()` returns a range object that behaves like an iterator. It generates numbers within a specified range on demand rather than storing them all in memory.

The `iter()` function is used to obtain iterators from different types of iterables: a list, a custom callable, and a file object. Once you have an iterator, you can iterate over it using a `for` loop, use it with other functions that accept iterators, or manually call `next()` on it to retrieve the next element.

```python
my_list = [1, 2, 3, 4, 5]
my_iterator = iter(my_list)
# Print the first element of the list
print(next(my_iterator))

# Iterate over the remaining elements using a for loop
for i in my_iterator:
    print(i)
```

The `enumerate()` function is used to iterate over a sequence (such as a list, tuple, or string) while also tracking the index of each item. It returns an iterator that yields tuples containing both the index and the value of each item in the sequence.

```python
my_list = ['apple', 'banana', 'cherry']

for index, value in enumerate(my_list):
    print(f"Index: {index}, Value: {value}")
```

A range object is not technically an iterator, but it behaves similarly to one. The `range` object itself does **not** have a `next()` method.

```python
# Create a range object with range(start, stop, step)
my_range = range(10, 0, -2)

# Print each number generated by the range object using next()
for i in my_range:
    print(i) # 10, 8, 6, 4, 2
```

1.4.3 Generator Functions

Generators are a *special type of iterator* created using generator functions or generator expressions. Iterators are a general concept in Python for objects that can be iterated over, while generators are a specific implementation of iterators that provide a concise and efficient way to generate values lazily using generator functions or expressions. Generators are a powerful tool for working with iterators, especially when dealing with large datasets or when generating sequences dynamically.

Generator functions use the `yield` keyword to produce a series of values lazily, allowing for efficient memory usage and enabling the processing of large datasets or infinite sequences. Generators functions pause and resume their execution, allowing them to produce a sequence of values over time, rather than computing them all at once and storing them in memory.

When you use `yield` in a function, it turns that function into a generator. When the function is called, it doesn't execute immediately; instead, it returns a generator object. When you iterate over this generator object using a loop or other iteration constructs, the function starts executing, and when it encounters a `yield` statement, it temporarily suspends its execution and returns the value specified by yield. The next time you iterate over the generator, it resumes execution from where it left off.

```python
def my_generator():
    yield 1
    yield 2
    yield 3

# Using the generator
gen = my_generator()
print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
# print(next(gen))  # Would raise StopIteration error
```

```python
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Create a Fibonacci generator
fib_gen = fibonacci()

# Print the first 10 Fibonacci numbers
print("First 10 Fibonacci numbers:")
for _ in range(10):
    print(next(fib_gen))
```

The key here is that the tuple texttt(b, a + b) is created before any assignment occurs, so a is assigned the value of b and b is assigned the sum of the old values of a and b. This allows for a simultaneous update of both variables without needing a temporary variable.

1.4.4 Lambda Functions

Lambda functions, also known as **anonymous functions**, are small, inline functions that are defined using the `lambda` keyword. Unlike regular functions defined using the `def` keyword, lambda functions are single-expression functions that can be created quickly without needing a formal `def` statement.

```
Adding two numbers
add = lambda x, y: x + y
print(add(3, 5))  # Output: 8

# Squaring a number
square = lambda x: x ** 2
print(square(4))  # Output: 16

# Checking if a number is even
is_even = lambda x: x % 2 == 0
print(is_even(5))  # Output: False
print(is_even(6))  # Output: True
```

To create an iterator that yields only the even numbers from a given list, the `filter()` function is utilized along with a lambda function. The `filter()` function takes two arguments: the first argument is the lambda function that defines the condition for filtering, and the second argument is the iterable from which elements are filtered. In this case, the lambda function checks if a number `x` is even by performing the modulo operation `% 2` and comparing the result to `0`. The `filter()` function then returns an iterator that yields only those elements from the input iterable for which the lambda function returns `True`. Finally, the filtered iterator is iterated over to print the even numbers.

```
# Define a list of numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Use filter() with a lambda function
even_numbers_iterator = filter(lambda x: x % 2 == 0, numbers)

# Iterate over the filtered iterator and print the even numbers
print("Even numbers:")
for num in even_numbers_iterator:
    print(num)
```

2 Object-Oriented Programming (OOP)

2.1 Terms and Programming Concepts

- **Class:** A class is a blueprint for creating objects. It defines the attributes (variables) and methods (functions) that all instances of the class will have.
- **Instance:** An instance is a specific realization of a class. It is an individual object created from a class, with its own set of attributes and methods.
- **Attribute:** An attribute is a piece of data associated with a particular object. It can be thought of as a variable that belongs to an object.
- **Method:** A method is a function defined within a class. It operates on the data within the class and can modify the state of the object or perform some action.
- **Object:** In Python, everything is an object. An object is a collection of data (variables) and methods (functions) that act on the data.
- **Type:** The type of an object determines what class it belongs to. It defines the attributes and methods that the object will have.

```python
# Define a Dog class
class Dog:
    # Constructor method to initialize object attributes
    def __init__(self, name, breed, age):
        self.name = name
        self.breed = breed
        self.age = age

    # Method to get description of the dog
    def get_description(self):
        return f"{self.name} is a {self.age}-year-old {self.breed}."


# Create an instance of the Dog class
my_dog = Dog('Buddy', 'Golden Retriever', 3)

# Accessing age attribute directly
print(my_dog.age)  # Output: 3

# Call the get_description method to print dog's description
print(my_dog.get_description())

# Print the type of the object and the name of the class
print(type(my_dog))          # Output: <class '__main__.Dog'>
print(type(my_dog).__name__)  # Output: Dog
```

2.1.1 The `__init__()` method

The `__init__` method is a special method used for initializing objects of a class. It is also known as a constructor method. The `__init__` method is called automatically whenever a new instance of the class is created.

2.1.2 Instance and Class Variables

- **Instance Variables**
  Variables that are unique to each instance of a class. They are defined within the constructor method (`__init__`) using the self keyword. Each instance of the class has its own copy of instance variables. Instance variables hold data that is specific to each object or instance of the class.
- **Class Variables**
  Variables that are shared among all instances of a class. They are defined outside of any method, typically at the beginning of the class definition. Class variables are accessed using the class name itself, rather than through instances of the class. They are used to store data that is common to all instances of the class.

```python
class Car:
    # Class variables
    wheels = 4
    total_cars = 0  # Counter for total number of cars

    def __init__(self, make, model):
        # Instance variables
        self.make = make
        self.model = model
        # Increment the total number of cars when a new instance is created
        Car.total_cars += 1


# Creating instances
car1 = Car('Toyota', 'Camry')
car2 = Car('Honda', 'Accord')
car3 = Car('Ford', 'Focus')

# Accessing class variable total_cars
print("Total number of cars:", Car.total_cars)  # Output: 3

# Accessing class variable using instance name (not recommended)
print(car1.total_cars) # Output: 3
```

Although accessing class variables using an instance name works in Python, it's clearer and more conventional to access them using the class name directly. This helps to distinguish between instance variables and class variables clearly.

2.1.3 The `__dict__` Attribute

The `__dict__` is a special attribute that is present in every object and holds the object's attributes. It is a dictionary that maps attribute names (as keys) to their corresponding values. You can use `__dict__` to access, manipulate, or inspect the attributes of an object dynamically.

```python
class MyClass:
    class_variable = 0

    def __init__(self, x, y):
        self.x = x
        self.y = y


obj1 = MyClass(10, 20)

# Printing instance variables of obj1
print(obj1.__dict__)  # Output: {'x': 10, 'y': 20}

# Printing class variables and methods of MyClass
print(MyClass.__dict__)  # {'__module__': '__main__', 'class_variable': 0, ...
```

2.1.4 The `vars()` Function

The `vars()` function is closely related to the `__dict__` attribute. It returns the `__dict__` attribute of an object if it exists, allowing you to access the object's attributes in the form of a dictionary. This function provides a convenient way to inspect the attributes of an object dynamically, similar to directly accessing the `__dict__` attribute.

```python
class MyClass:
    class_variable = 0

    def __init__(self, x, y):
        self.x = x
        self.y = y


obj1 = MyClass(10, 20)

# Printing instance variables of obj1
print(vars(obj1))

# Printing class variables and methods of MyClass
print(vars(MyClass))
```

```
{'x': 10, 'y': 20}
{
    '__module__': '__main__',
    'class_variable': 0,
    '__init__': <function MyClass.__init__ at 0x00000239ACED7600>,
    '__dict__': <attribute '__dict__' of 'MyClass' objects>, ...
}
```

2.2 Core Syntax Operations

2.2.1 Magic Methods

Magic methods, also known as dunder methods (short for ”double underscore”), are special methods that have double underscore prefixes and suffixes, like `__add__`. These methods enable customization of how objects behave in certain contexts, such as arithmetic operations, comparison operations, and object creation.

Comparison methods

Operator	Magic method	Implementation meaning or purpose
`==`	`__eq__(self, other)`	Equality operator
`!=`	`__ne__(self, other)`	Inequality operator
`<`	`__lt__(self, other)`	Less-than operator
`>`	`__gt__(self, other)`	Greater-than operator
`<=`	`__le__(self, other)`	Less-than-or-equal-to operator
`>=`	`__ge__(self, other`	Greater-than-or-equal-to operator

Unary operators and functions

Operator	Magic method	Implementation meaning or purpose
`+`	`__pos__(self)`	Unary positive, like a = +b
`-`	`__neg__(self)`	Unary negative, like $a = -b$
`abs()`	`__abs__(self)`	Behavior for `abs()` function
`round(a, b)`	`__round__(self, b)`	Behavior for `round()` function

Common, binary operators and functions

Operator	Magic method	Implementation meaning or purpose
`+`	`__add__(self, other)`	Addition operator
`-`	`__sub__(self, other)`	Subtraction operator
`*`	`__mul__(self, other)`	Multiplication operator
`//`	`__floordiv__(self, other)`	Integer division operator
`/`	`__truediv__(self, other)`	Division operator
`%`	`__mod__(self, other)`	Modulo operator
`**`	`__pow__(self, other)`	Exponential (power) operator

Augmented operators and functions

Operator	Magic method	Implementation meaning or purpose
`+=`	`__iadd__(self, other)`	Addition and assignment operator
`-=`	`__isub__(self, other)`	Subtraction and assignment operator
`*=`	`__imul__(self, other)`	Multiplication and assignment operator
`//=`	`__ifloordiv__(self, other)`	Integer division and assignment operator
`/=`	`__idiv__(self, other)`	Division and assignment operator
`%=`	`__imod__(self, other)`	Modulo and assignment operator
`**=`	`__ipow__(self, other)`	Exponential (power) and assignment operator

Type conversion methods

Function	Magic method	Implementation meaning or purpose
`int()`	`__int__(self)`	Conversion to integer type
`float()`	`__float__(self)`	Conversion to float type
`oct()`	`__oct__(self)`	Conversion to string, containing an octal representation
`hex()`	`__hex__(self)`	Conversion to string, containing a hex representation

Object introspection

Function	Magic method	Implementation meaning or purpose
`str()`	`__str__(self)`	Handling str() function calls
repr()	`__repr__(self)`	Handling repr() function calls
`format()`	`__format__(self, formatstr)`	When string formatting is applied to an object
`hash()`	`__hash__(self)`	Handling hash() function calls
`dir()`	`__dir__(self)`	Handling dir() function calls
`bool()`	`__nonzero__(self)`	Handling bool() calls (`__bool__()` in Python 3)

Object retrospection

Function	Magic method	Implementation
`isinstance(obj, class)`	`__instancecheck__(self, obj)`	Handling isinstance() calls
`issubclass(subclass, class)`	`__subclasscheck__(self, subclass)`	Handling issubclass() calls

Object attribute access

Expression	Magic method	Implementation
`obj.attr`	`__getattr__(self, attr)`	Access to a non-existing attribute
`obj.attr`	`__getattribute__(self, attr)`	Access to an existing attribute
`obj.attr = val`	`__setattr__(self, attr, val)`	Setting an attribute value
`del object.attr`	`__delattr__(self, attr)`	Deleting an attribute

Methods allowing access to containers

Expression	Magic method	Implementation
`len(container)`	`__len__(self)`	Returns number of elements of the container
container[key]	`__getitem__(self, key)`	Fetching an element identified by the key
`container[key] = value`	`__setitem__(self, key, value)`	Setting a value to an element identified by the key
`del container[key]`	`__delitem__(self, key)`	Deleting an element identified by the key
`for el in container`	`__iter__(self)`	Returns an iterator for the container
`item in container`	`__contains__(self, item)`	Returns true if container has the item

By implementing these methods in a class, you can customize how instances of the class behave in various contexts, making your code more expressive and idiomatic.

2.2.2 Comparison Methods

This code demonstrates the use of the `__eq__` method in classes, along with the behavior of the equality operator (==).

```
class Point():
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return (self.x, self.y) == (other.x, other.y)

point1=Point(1,2)
point2=Point(1,2)

print(point1.__eq__(point2))  # Output: True
print(point1 == point2)  # Output: True
```

2.2.3 Accessing Containers

The `__getitem__` method is a special method that allows objects to support indexing and slicing operations. It is called when you use square brackets (`[]`) to access elements from a container object, such as lists, dictionaries, or custom classes.

```
class MyContainer:
    def __init__(self, data):
        self.data = data

    def __getitem__(self, index):
        return self.data[index]

container = MyContainer([1, 2, 3, 4, 5])
print(container[2])  # Output: 3
```

2.2.4 Numeric Methods

The `__abs__()` method is a special method that allows objects to define custom behavior for the built-in `abs()` function when applied to instances of the class.

```
class MyNumber:
    def __init__(self, value):
        self.value = value

    def __abs__(self):
        return abs(self.value)

x = MyNumber(-5)
print(abs(x))  # Output: 5
```

2.2.5 Customizing Object Behavior

The methods `__call__`, `__repr__`, and `__str__` are special methods that allow to customize the behavior of objects.

- **The __call__ method:** allows the instance of `MyClass` to behave like a function. It is called when an instance of the class is used as if it were a function, for example, `obj()`.
- **The __repr__ method:** provides an unambiguous representation of the object, primarily meant for developers. It should ideally be valid Python code to recreate the object. It is used for debugging, logging, or creating new instances of the object.
- **The __str__ method:** provides a more user-friendly string representation of the object. It is called when the built-in function `str()` is used on the object, or when the object is passed to the `print()` function.

```python
class MyClass:
    def __init__(self, x):
        self.x = x

    def __call__(self, y):
        # Allows instances of a class to be invoked like functions
        return self.x + y

    def __repr__(self):
        # Returns a string representation for developers
        return f'MyClass({self.x})'

    def __str__(self):
        # Returns a user-friendly string representation of the object
        return f'MyClass object with x = {self.x}'

# Creating an instance of MyClass
obj = MyClass(5)

# Using __call__ method
print(obj(3))

# Using __repr__ method
print(repr(obj))
print([obj]) # Calls __repr__ for each object in the list
logging.error(repr(obj)) # Using repr() for debugging/logging
#print(`obj`)  # Equivalent to repr(obj), in Python 2.x

# Using __str__ method, calls obj.__repr__() if __str__ is not defined
print(obj)
print(str(obj))
print(f'{obj}')
print("%s" % obj)
print("{}".format(obj))
print(MyClass(2))
```

2.2.6 Object Attribute Access

The __getattr__, __setattr__, and __delattr__ methods are special methods that allow you to customize attribute access, assignment, and deletion behavior in objects.

- **__getattr__**: This method is called when an attribute lookup fails. It allows you to define custom behavior for accessing attributes that don't exist in an object's namespace.
- **__setattr__**: This method is called when an attribute assignment is made. It allows you to customize behavior when setting attributes.
- **__delattr__**: This method is called when an attribute deletion is requested. It allows you to customize behavior when deleting attributes.

```python
class DynamicAttributes:
    def __init__(self):
        self.data = {}

    def __getattr__(self, name):
        if name in self.data:
            return self.data[name]
        else:
            raise AttributeError(f"Object has no attribute '{name}'")

    def __setattr__(self, name, value):
        print(f"Setting attribute '{name}' to '{value}'")
        self.data[name] = value

    def __delattr__(self, name):
        if name in self.data:
            del self.data[name]
            print(f"Deleted attribute '{name}'")
        else:
            raise AttributeError(f"Object has no attribute '{name}'")


# Creating an instance of DynamicAttributes
obj = DynamicAttributes()

# Setting attribute
obj.x = 10  # Output: Setting attribute 'x' to '10'

# Accessing attribute
print(obj.x)  # Output: 10

# Deleting attribute
del obj.x  # Output: Deleted attribute 'x'

# Trying to access deleted attribute
print(obj.x)  # Raises AttributeError
```

2.3 Inheritance, Polymorphism, and Composition

2.3.1 Superclasses and Subclasses

- **Superclasses:** Provide a blueprint or template for the subclasses to follow.
- **Subclasses:** Inherit properties (attributes) and behaviors (methods) from their superclasses.

```python
class Animal:
    def speak(self):
        print("This animal makes a sound.")

class Dog(Animal):
    def speak(self):
        print("Woof!")

class Cat(Animal):
    def speak(self):
        print("Meow!")


# Create instances of Dog and Cat
dog = Dog()
cat = Cat()
```

The code defines three classes: `Animal`, `Dog`, and `Cat`. The `Animal` class has a method named `speak` that prints ”This animal makes a sound.” Both the `Dog` and `Cat` classes are subclasses of `Animal`, inheriting the `speak` method. The `Dog` class overrides the `speak` method with its own implementation to print ”Woof!”, and similarly, the `Cat` class overrides the `speak` method with its own implementation to print ”Meow!”.

2.3.2 Reflection: `isinstance()` and `issubclass()`

Reflection refers to the ability of a program to examine and modify its own structure, behavior, and metadata at runtime. This includes the capability to inspect classes, functions, modules, and objects to retrieve information about their attributes, methods, and other properties

- **Introspection**: Examining the type and structure of objects at runtime.
- **Dynamic instantiation**: Creating instances of classes dynamically, based on runtime information.
- **Attribute and method manipulation**: Accessing and modifying attributes and methods of objects dynamically.
- **Dynamic module loading**: Loading modules dynamically during program execution.
- **Metaprogramming**: Writing code that generates or modifies other code.

The `isinstance()` and `issubclass()` function are two built-in functions in Python used for type checking and inheritance checking respectively.

- **`isinstance()`:** This function is used to check if an object belongs to a particular class or any subclass of that class. It takes two arguments: the object and the class (or a tuple of classes). It returns True if the object is an instance of the specified class or any of its subclasses, otherwise, it returns False.

  ```
  print(isinstance(dog, Animal))  # Output: True
  print(isinstance(dog, Dog))     # Output: True
  print(isinstance(cat, Dog))     # Output: False
  ```

- **`issubclass()`:** This function is used to check if a class is a subclass of another class. It takes two arguments: the subclass and the superclass. It returns True if the first class is indeed a subclass of the second class, otherwise, it returns False. Reflection enables dynamic manipulation of code and data structures, allowing for tasks such as:

  ```
  print(issubclass(Dog, Animal))  # Output: True
  print(issubclass(Cat, Animal))  # Output: True
  print(issubclass(Cat, Cat))     # Output: True
  ```

  Cat is a subclass of Animal, and `issubclass(Cat, Cat)` correctly outputs `True` because a class is considered a subclass of itself. This behavior is expected in Python.

2.3.3 Introspection

Introspection refers to the ability of a program to examine the type, properties, and methods of objects at runtime. It allows you to dynamically inspect and manipulate objects, classes, functions, and modules within your code. Python's introspection capabilities are facilitated by various built-in functions and modules that provide information about objects. Some of the commonly used functions and modules for introspection include:

- **type():** Returns the type of an object.
- **dir():** Returns a list of attributes and methods of an object.
- **getattr():** Retrieves the value of an attribute of an object dynamically, based on its name.
- **hasattr():** Checks whether an object has a specific attribute.
- **callable():** Checks whether an object is callable (i.e., whether it can be called as a function).
- **help():** Provides interactive help on objects, modules, functions, classes, and methods.

```
class Cat:
    def __init__(self, name, color):
        self.name = name
        self.color = color

my_cat = Cat("Whiskers", "gray")

# Example using getattr and hasattr
print(getattr(my_cat, "name"))  # Output: Whiskers
print(hasattr(my_cat, "name"))   # Output: True
print(hasattr(my_cat, "age"))    # Output: False
```

2.3.4 Inheritance vs. Composition

In object-oriented programming, inheritance and composition are two fundamental concepts used for structuring and organizing code.

- **Inheritance**: A new class (subclass or derived class) is created by inheriting properties and behaviors from an existing class (superclass or base class). It promotes code reusability by allowing subclasses to inherit attributes and methods from a superclass. Subclasses can then add their own attributes and methods or override the ones inherited from the superclass.

```python
class Vehicle:
    def drive(self):
        return "Vehicle is being driven"

class Car(Vehicle):
    def drive(self):
        return "Car is being driven"
```

- **Composition**: A class contains objects of other classes as members. Instead of inheriting their behavior, the containing class delegates tasks to contained objects. It allows for building complex objects by combining simpler ones. Each object maintains its own behavior, and the containing object orchestrates their interactions.

```python
class Engine:
    def start(self):
        return "Engine started"

class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        return self.engine.start()
```

2.3.5 Real-life problems using ”is a” and ”has a” relations

- **Inheritance: ”is-a” Relationship**
  Subclasses are specialized versions of their superclass. Subclasses inherit common properties and behaviors from their superclass. For example, a Car is a type of Vehicle.

- **Composition: ”has-a” Relationship**
  Classes contain instances of other classes as components. This allows them to utilize the functionalities of those components. For example, a Car has an Engine, Wheel, and Transmission.

2.3.6 Class Hierarchies

Class hierarchies are structures that organize classes into a tree-like hierarchy, where classes can inherit attributes and methods from their parent classes. This enables code reuse, modularity, and the implementation of polymorphism.

2.3.7 Single vs. Multiple Inheritance

Inheritance allows a class to inherit attributes and methods from another class. There are two types of inheritance: single inheritance and multiple inheritance.

- **Single Inheritance:** Class inherits from only one base class.

```python
class Animal:
    def sleep(self):
        return "Animal sleeps"

class Dog(Animal):
    def bark(self):
        return "Dog barks"


dog = Dog()

# Dog inherits from Animal
# Dog class has access to the methods of Animal class
print(dog.sleep())  # Output: "Animal sleeps"
print(dog.bark())   # Output: "Dog barks"
```

- **Multiple Inheritance:** Class inherits from multiple base classes.

```python
class Animal:
    def sleep(self):
        return "Animal sleeps"

class Pet:
    def play(self):
        return "Pet plays"

class Dog(Animal, Pet):
    def bark(self):
        return "Dog barks"


dog = Dog()

# Dog inherits from both Animal and Pet
# Dog class has access to the methods of Animal and Pet classes
print(dog.sleep())  # Output: "Animal sleeps"
print(dog.play())   # Output: "Pet plays"
print(dog.bark())   # Output: "Dog barks"
```

2.3.8 Polymorphism

Polymorphism refers to the ability of different objects to respond to the same method or function call in different ways. It allows objects of different classes to be treated as objects of a common superclass, enabling code reuse and flexibility in design. There are two main types of polymorphism:

- **Method Overriding:** This occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method in the subclass overrides the method with the same name and signature in the superclass.
- **Operator Overloading:** This allows operators to have different implementations depending on the types of operands involved. Python provides special methods, often referred to as ”magic methods” or ”dunder methods”, to enable operator overloading.

2.3.9 Duck Typing

Duck typing is a concept, where the type or class of an object is determined not by its inheritance hierarchy or explicit declaration, but by its behavior. In other words, an object's suitability is determined by whether it behaves like a duck, if it walks like a duck and quacks like a duck, then it must be a duck. This means that if an object supports the necessary methods or attributes required by a particular operation, it can be used in that context regardless of its actual class or inheritance.

```python
class Duck:
    def quack(self):
        print("Quack!")

class Robot:
    def quack(self):
        print("I'm quacking like a duck!")

def make_it_quack(obj):
    obj.quack()

# Instances of Duck and Robot
duck = Duck()
robot = Robot()

# Duck typing allows any object with a quack() method
make_it_quack(duck)    # Output: "Quack!"
make_it_quack(robot)  # Output: "I'm quacking like a duck!"
```

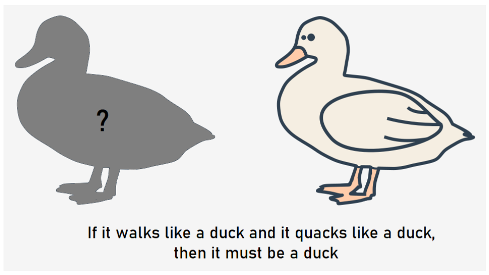


Figure 2: Duck typing

2.3.10 Diamond Problem: Method Resolution Order (MRO)

The Diamond problem does not exist in Python because it gives preference to the class that is inherited first. The Method Resolution Order (MRO) defines the order in which Python looks for methods and attributes in a class hierarchy. When a method is called on an object, Python searches for it starting from the class of the object and then follows the MRO to find the method in the inheritance hierarchy.

```python
class A:
    def hello(self):
        print('Hello from class A')

class B(A):
    def hello(self):
        print('Hello from class B')

class C(A):
    def hello(self):
        print('Hello from class C')

class D(B, C):
    pass

D().hello()  # Output: Hello from class B
```

`D().hello()` calls the `hello()` method of class D. Since class D doesn't have its own `hello()` method, Python looks for it in the classes D inherits from, following the MRO. Therefore, the output of `D().hello()` is ”Hello from class B”.

```python
# Output the MRO for class D
print(D.__mro__)
```

```
(__main__.D, __main__.B, __main__.C, __main__.A, object)
```

Python will look for the `hello` method in class D first, then in class B, then in class C, then in class A, and finally in the base class `object`. Given the provided class hierarchy, the MRO for class D is as follows: $D \rightarrow B \rightarrow C \rightarrow A \rightarrow$ `object`

2.3.11 The Inheritance Tangle

The diamond inheritance problem is a common challenge encountered in object-oriented programming, particularly in languages like Python that support multiple inheritance. This issue arises when a class inherits from two or more classes, each of which in turn inherits from a common base class. As a result, the inheritance hierarchy forms a diamond shape, with multiple paths leading to the same base class.

```python
class A:
    def info(self):
        print('Class A')

class B(A):
    def info(self):
        print('Class B')

class MyClass(A, B):
    pass

MyClass().info()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[24], line 9
      6     def info(self):
      7         print('Class B')
----> 9 class MyClass(A, B):
     10     pass
     12 MyClass().info()

TypeError: Cannot create a consistent method resolution
order (MRO) for bases A, B
```

The error encountered stems from the diamond inheritance problem in Python. When a class like `MyClass(A, B)` is defined, inheriting from both `A` and `B`, Python faces difficulty in establishing a consistent method resolution order (MRO) due to conflicting inheritance paths. In this scenario, both `A` and `B` serve as base classes for `MyClass`, sharing a common ancestor (`object` by default in Python 3). However, Python's MRO algorithm struggles to determine a linear order for method search amidst these conflicting paths.

To resolve this error, it's necessary to redesign the class hierarchy to avoid the diamond inheritance problem. A common approach involves ensuring that each class inherits from its base classes via a single, unambiguous path.

2.3.12 The `super()` Function

`super()` is a built-in function that allows you to call methods and access attributes from the parent class within a subclass. It provides a way to invoke methods defined in the superclass (parent class) of an object, enabling better code reuse and maintaining a clean inheritance structure.

The `super()` function is typically used inside methods of a subclass to call the corresponding method of the superclass. This is particularly useful when the subclass overrides a method defined in the superclass but still wants to execute the overridden method's behavior. By using `super()`, you can avoid explicitly naming the parent class and make your code more flexible to changes in the inheritance hierarchy.

```python
class Parent:
  def __init__(self, msg):
    print("Calling Parent class constructor")
    self.message = msg

  def printmessage(self):
    print(self.message)

class Child(Parent):
  def __init__(self, msg):
    print("Calling Child class constructor")
    super().__init__(msg)


obj = Child("Hello, and welcome!")
obj.printmessage()
```

```
Calling Child class constructor
Calling Parent class constructor
Hello, and welcome!
```

2.3.13 Subclassing Built-in Classes

Inheriting properties from built-in classes allows for flexible customization of classes to suit specific needs without having to start from scratch. This is often done to customize behavior, add functionality, or override existing methods. However, it's essential to be cautious when subclassing built-ins, as it may lead to unexpected behavior if not done carefully.

```python
class CustomList(list):
    def __init__(self, *args):
        super().__init__(*args)

    def append_twice(self, item):
        self.append(item)
        self.append(item)

    def sum(self):
        return sum(self)

    def average(self):
        if len(self) == 0:
            return 0
        return sum(self) / len(self)


# Creating an instance of CustomList
my_list = CustomList([1, 2, 3, 4, 5])

# Inheriting built-in method 'append'
my_list.append(6)

# Adding custom method 'append_twice'
my_list.append_twice(7)

print("Modified List:", my_list) # [1, 2, 3, 4, 5, 6, 7, 7]
print(my_list.sum()) # 35
print(my_list.average()) # 4.375
```

Inheritance allows to acquire properties and methods from built-in classes such as `list`, `dict`, or `str`. Here's an example demonstrating inheritance from the `list` class to create a custom class called `CustomList`, along with additional functionality.

2.4 Function Argument Syntax

2.4.1 Function Parameter Handling

Function parameter handling refers to the way functions accept and process input arguments. It involves defining parameters in function definitions and handling these parameters within the function body.

- **Positional Parameters**

  Positional parameters are defined by their position in the function call. Their values are assigned based on their order in the function definition.

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

# Positional parameters
result = greet("Alice", "Hello")
print(result)  # Output: Hello, Alice!
```

- **Keyword Parameters**

  Keyword parameters are explicitly named in the function call, allowing arguments to be passed in any order.

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

# Keyword parameters
result = greet(greeting="Hi", name="Bob")
print(result)  # Output: Hi, Bob!
```

- **Default Parameters**

  Default parameters have predefined values and are used when the caller does not provide a value for them.

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

# Default parameter
result = greet("Charlie")
print(result)  # Output: Hello, Charlie!
```

- **Arbitrary Number of Arguments**

  Functions can accept an arbitrary number of positional or keyword arguments using the `*args` and `**kwargs` syntax. You aren't restricted to using the name `*args`; you can choose any name you prefer, like `*names` or any other identifier.

```
def greet(*names):
    for name in names:
        print(f"Hello, {name}!")

# Arbitrary number of positional arguments
greet("Alice", "Bob", "Charlie")
```

2.4.2 Special Identifiers: *args and **kwargs

The identifiers *args and **kwargs are special syntax used in function definitions to pass a variable number of arguments to a function. They are often referred to as ”argument unpacking” and ”keyword argument unpacking,” respectively.

- ***args**
  Allows a function to accept any number of positional arguments. When *args is used in a function definition, it collects all the positional arguments that are passed to the function into a tuple.

```
def my_function(*args):
    for arg in args:
        print(arg)

my_function(1, 2, 3)
# Output:
# 1
# 2
# 3
```

- ****kwargs**
  Allows a function to accept any number of keyword arguments. When **kwargs is used in a function definition, it collects all the keyword arguments (arguments passed with their respective keys) into a dictionary.

```
def my_function(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

my_function(name="Alice", age=30, city="New York")
# name: Alice
# age: 30
# city: New York
```

2.4.3 Forwarding Arguments

When forwarding arguments received by a function defined with *args and **kwargs to another function, the following approach should be used:

```
def combiner(a, b, *args, **kwargs):
    print("a:", a, "Type:", type(a)) # a: 10 Type: <class 'int'>
    print("b:", b, "Type:", type(b)) # b: 20 Type: <class 'str'>
    super_combiner(*args, **kwargs)

def super_combiner(*super_args, **super_kwargs):
    print(super_args) # (40, 60, 30)
    print(super_kwargs) # {'arg1': 50, 'arg2': '100'}

combiner(10, '20', 40, 60, 30, arg1=50, arg2='100')
```

2.4.4 Order of Function Parameters

Function parameters are typically defined in a specific order:

1. Positional parameters
2. Parameter with default values
3. `*args`
4. `**kwargs`

```
def say_greeting(name, greeting="Hello", *args, **kwargs):
    print(greeting + ", " + name)
    print(args)
    for key, value in kwargs.items():
        print(f"{key}: {value}")

say_greeting("John")
say_greeting("John", "Welcome", 123, 456, one=1, two=2, three=3)
```

A function can accept either all required arguments or a mix of required and optional keyword arguments, but they must follow a specific order. First come all the required arguments, followed by the optional keyword arguments. If a function is defined where a non-default argument follows a default argument, a SyntaxError will occur.

```
def say_greeting_bad(greeting="Hello", name):
    print("Oops, this won't work!") # SyntaxError
```

```
def say_greeting(greeting="Hello", name="World"):
    return greeting + " " + name

say_greeting()                                  # "Hello World"
say_greeting("Hi", "Joe")                       # "Hi Joe"
say_greeting(greeting="Welcome", name="Tom")  # "Welcome Tom"
say_greeting("Hi")                              # "Hi World"
say_greeting(name="Alice")                      # "Hello Alice"
```

2.4.5 Closures

A closure is a function that remembers the values from its enclosing lexical scope even when the scope is no longer available. It means that the function definition encloses (or captures) variables from the surrounding scope, allowing the function to access and manipulate those variables even after the scope in which they were defined has finished executing.

```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

closure = outer_function(10)
result = closure(5)  # Result will be 15
```

2.5 Decorators

Decorators are a way to modify the behavior of function or class behavior by wrapping them within another function or class. Typically implemented as a higher-order function, decorators return a function that augments the original functionality, often applied using wrapper syntax. This mechanism allows for the addition of functionalities such as logging, caching, or access control without directly altering the original function or class definition.

```python
def my_decorator(func):
    print(f'Running function... "{func.__name__}"')
    return func

@my_decorator
def hello():
    print('Hello, world!')

hello()
# Running function... "hello"
# Hello, world!
```

2.5.1 Decorating Functions with Functions

The `log_function_call` decorator logs the function name and its arguments whenever the decorated function is called. Adjustments to the logging level and format can be made according to specific requirements.

```python
import logging
import functools

logging.basicConfig(level=logging.INFO)

def log_function_call(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with args: {args}")
        logging.info(f"Calling {func.__name__} with args: {args}")
        return func(*args, **kwargs)
    return wrapper

@log_function_call
def add(a, b):
    print(f"{a}+{b}={a+b}")
    return a + b

@log_function_call
def greet(name):
    print(f"Hello, {name}!")
    return f"Hello, {name}!"

add(2, 3)
greet("Alice")
```

```
Calling add with args: (2, 3)
2+3=5
Calling greet with args: ('Alice',)
Hello, Alice!
```

Both add and greet functions are decorated with log_function_call. Therefore, each time these functions are called, the log_function_call decorator logs the function name and its arguments. This is why the lines indicating the function calls with their respective arguments are also displayed in the output.

2.5.2 Decorating Fuctions with Classes

This code defines a class named MyDecorator that serves as a decorator. When a function is decorated with @MyDecorator, an instance of MyDecorator is created with the decorated function passed to its constructor.

```python
class MyDecorator:
    def __init__(self, function):
        self.function = function

    def __call__(self, *args, **kwargs):
        print('Running __call__ method...')
        self.function(*args, **kwargs)
        print('The decorator is still working...')


@MyDecorator
def hello(*args, **kwargs):
    print('Hello, world!')
    print(f'args: {args}')
    print(f'kwargs: {kwargs}')


hello('script', 'py', lang='Python', version=(3, 10, 0))
```

```
Running __call__ method...
Hello, world!
args: ('script', 'py')
kwargs: {'lang': 'Python', 'version': (3, 10, 0)}
The decorator is still working...
```

The __call__ method of MyDecorator is invoked when the decorated function is called. Inside __call__, it first prints a message indicating that the __call__ method is running. Then, it calls the original function with the provided arguments and keyword arguments (*args and **kwargs). After the original function has been executed, it prints another message indicating that the decorator is still working.

2.5.3 Decorator Stacking

Decorator stacking, also known as decorator chaining, refers to the practice of applying multiple decorators to a single function or method. This allows you to compose functionality in a flexible and modular way, by adding or modifying behavior in a sequential manner. Decorators can be stacked by applying them one after another using the "@" symbol. When a function or method is decorated with multiple decorators, they are applied in the order they appear, from top to bottom.

```python
def decorator1(func):
    def wrapper():
        return func() + "World!"
    return wrapper

def decorator2(func):
    def wrapper():
        return "Hello" + func()
    return wrapper

@decorator1
@decorator2
def greet():
    return ", "
print(greet())
```

```
Hello, World!
```

In this example, decorator1 adds ", World!" to the result of the decorated function, and decorator2 adds "Hello" to the result. When the greet function is called, it passes through both decorators, resulting in the final output "Hello, World!".

2.5.4 Decorator Arguments

Decorator arguments facilitate customization of decorators by enabling the passing of additional parameters to them. This enhances the flexibility and reusability of decorators, making them adaptable to various use cases.

In this example, the repeat decorator factory takes an argument n and returns the decorator function, which in turn takes the function to be decorated (func). Inside the wrapper function, the original function func is called n times.

```python
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            return [func(*args, **kwargs) for _ in range(n)]
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    return f"Hello, {name}!"

print(greet("World"))
```

```
['Hello, World!', 'Hello, World!', 'Hello, World!']
```

This code demonstrates the use of a decorator function, `warehouse_decorator`, to modify the behavior of other functions.

```python
def warehouse_decorator(material):
    def wrapper(func):
        def internal_wrapper(*args):
            print(f"Wrapping items from {func.__name__} with {material}")
            func(*args)
            print()
        return internal_wrapper
    return wrapper

@warehouse_decorator('paper')
def pack_books(*args):
    print("We'll pack books:", args)

@warehouse_decorator('foil')
def pack_toys(*args):
    print("We'll pack toys:", args)

@warehouse_decorator('cardboard')
def pack_fruits(*args):
    print("We'll pack fruits:", args)


pack_books('Alice in Wonderland', 'Winnie the Pooh')
pack_toys('doll', 'car')
pack_fruits('plum', 'pear')
```

```
Wrapping items from pack_books with paper
We'll pack books: ('Alice in Wonderland', 'Winnie the Pooh')

Wrapping items from pack_toys with foil
We'll pack toys: ('doll', 'car')

Wrapping items from pack_fruits with cardboard
We'll pack fruits: ('plum', 'pear')
```

Decorators can accept arguments by enclosing them within an additional function. This additional function acts as a closure, capturing the arguments provided to the decorator.

2.5.5 Decorator Pattern

The Decorator pattern is a structural design pattern that allows behavior to be added to individual objects dynamically without affecting the behavior of other objects from the same class. It's used to extend or modify the behavior of objects at runtime without altering their existing code.

In the Decorator pattern:

- **Component**: This is the interface or abstract class defining the base functionality. It's the object that will be decorated.
- **Concrete Component**: This is the class implementing the base component interface. It provides the basic functionality that can be extended or modified.
- **Decorator**: This is an abstract class or interface that wraps around the base component. It has the same interface as the base component, allowing it to be used interchangeably. The decorator may contain a reference to the base component and adds additional functionality by delegating calls to the base component.
- **Concrete Decorator**: These are the concrete subclasses of the decorator that add specific behavior or modify existing behavior. Each concrete decorator typically extends the functionality of the base component in a specific way.

The Decorator pattern allows for flexible and modular design, promoting code reusability and maintainability. It enables you to add or modify functionality in a flexible and dynamic manner, without needing to modify the underlying classes.

```python
# Base Component
class Component:
    def operation(self):
        pass

# Concrete Component
class ConcreteComponent(Component):
    def operation(self):
        return "ConcreteComponent"

# Decorator
class Decorator(Component):
    def __init__(self, component):
        self._component = component

    def operation(self):
        return self._component.operation()

# Concrete Decorator
class ConcreteDecoratorA(Decorator):
    def operation(self):
        return f"ConcreteDecoratorA({self._component.operation()})"

class ConcreteDecoratorB(Decorator):
    def operation(self):
        return f"ConcreteDecoratorB({self._component.operation()})"
```

```
# Usage
component = ConcreteComponent()
decorated_component = ConcreteDecoratorA(ConcreteDecoratorB(component))
print(decorated_component.operation())
```

```
ConcreteDecoratorA(ConcreteDecoratorB(ConcreteComponent))
```

In this example:

- **Component** is the base interface.
- **ConcreteComponent** is the concrete class implementing the base interface.
- **Decorator** is the abstract decorator class.
- **ConcreteDecoratorA** and **ConcreteDecoratorB** are concrete decorator classes.
- We create a decorated component by stacking decorators **ConcreteDecoratorA** and **ConcreteDecoratorB** on top of **ConcreteComponent**.

2.6 Static and Class Methods

2.6.1 Terms and Concept

Static and class methods are two types of methods that can be defined within a class.

- **Class methods**
  Defined using the `@classmethod` decorator. Accepts the class (`cls`) as the first argument. Can access and modify class-level data (class variables). Often used for operations that involve the class itself, such as creating alternate constructors or manipulating class-level attributes. Can be called on both the class and instances of the class.
- **Static methods**
  Defined using the `@staticmethod` decorator. Does not accept the class or instance as the first argument. Independent of the class or instance state. Primarily used for utility functions that are related to the class but don't require access to class or instance variables. Can be called on both the class and instances of the class, but usually called using the class name for clarity.

```python
class MyClass:
    class_variable = "Class variable"

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

    @classmethod
    def class_method(cls):
        return cls.class_variable

    @staticmethod
    def static_method():
        return "This is a static method"


# Using Class Method
print(MyClass.class_method())  # Output: Class variable

# Using Static Method
print(MyClass.static_method())  # Output: This is a static method

# Accessing Class Method from Instance
obj = MyClass("Instance variable")
print(obj.class_method())  # Output: Class variable

# Accessing Static Method from Instance
print(obj.static_method())  # Output: This is a static method
```

2.6.2 The `@classmethod` and `@staticmethod` Decorators

Decorators allow modifying or extending the behavior of functions or methods without changing their actual implementation. Here's why `@classmethod` and `@staticmethod` are considered decorators:

`@classmethod`

- Decorator that modifies the behavior of a method defined within a class to become a class method.
- Takes the method it decorates and binds it to the class itself rather than to the instance.
- When the method is called, the class is implicitly passed as the first argument (`cls` conventionally) rather than the instance (`self`).

`@staticmethod`

- Decorator that defines a static method within a class.
- It does not take the class or instance as the first argument.
- Static methods are independent of the class or instance state and behave like regular functions within the class namespace.

Decorators are indicated by the `@` symbol followed by the decorator name (e.g., `@decorator_name`). When you apply a decorator to a function or method, you're essentially telling Python to pass that function or method to the decorator, which then modifies its behavior accordingly.

Aspect	Class Method	Static Method
Decorator	`@classmethod`	`@staticmethod`
First Argument	`cls`	None
Access Class Data	Yes	No
Access Instance	No	No
Purpose	Operations involving the class itself	Utility functions related to the class

2.6.3 The `cls` Parameter

When defining a class method using the @classmethod decorator, the first parameter conventionally named cls is used to refer to the class itself. This parameter allows the method to access class-level variables and perform operations related to the class itself.

- **Name:** `cls` is a conventional name used for the first parameter of a class method. However, you can use any valid variable name in its place. Using `cls` is a widely accepted convention and helps in making the code more readable to other developers.
- **Reference to the class:** Inside a class method, `cls` is a reference to the class itself, not an instance of the class. This allows the class method to access and modify class-level variables, call other class methods, or perform any operation related to the class itself.
- **Implicit passing:** When calling a class method, you don't need to explicitly pass the class as an argument. Python automatically passes the class as the first argument to the method when it's called.
- **Alternative to `self`:** While `self` is used to refer to an instance of the class in instance methods, `cls` is used to refer to the class itself in class methods.

2.7 Abstract Classes and Methods

Abstract classes and methods are used to define a blueprint for other classes. They are not meant to be instantiated directly but rather serve as a template for subclasses. Abstract classes contain one or more abstract methods, which are methods without implementations. Subclasses must implement these abstract methods to provide their own functionality.

- **Abstract Classes**
  - An abstract class is a class that contains one or more abstract methods.
  - Abstract classes cannot be instantiated directly; they are meant to be subclassed.
  - Abstract classes can have both regular methods with implementations and abstract methods without implementations.
- **Abstract Methods**
  - An abstract method is a method declared within an abstract class that does not contain any implementation.
  - Abstract methods are meant to be overridden by subclasses.
  - Subclasses of an abstract class must implement all abstract methods defined in the abstract class; otherwise, they will also be considered abstract and cannot be instantiated.

2.7.1 Overriding Abstract Methods

Overriding abstract methods is a specific case of method overriding that occurs in the context of abstract base classes (ABCs) in object-oriented programming. Abstract methods are methods declared in an abstract class that don't have any implementation in the abstract class itself; instead, they are meant to be implemented by subclasses.

```python
from abc import ABC, abstractmethod

class Figure(ABC):
    @abstractmethod
    def area(self):
        pass

class Square(Figure):
    def __init__(self,a):
        self.a = a

    def area(self):
        return self.a * self.a

s = Square(5)
s.area()
```

Attempting to instantiate an abstract class like Figure directly will result in a TypeError because abstract classes cannot be instantiated. They are meant to be subclassed and provide a structure for derived classes to implement specific functionality. Python raises a TypeError with the message "Can't instantiate abstract class Figure with abstract method area" because the Figure class is abstract and the area method is not implemented.

2.7.2 Multiple Inheritance and Multiple Child Classes

```python
from abc import ABC, abstractmethod

# Abstract class for animals
class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

# Abstract class for flying creatures
class Flyable(ABC):
    @abstractmethod
    def fly(self):
        pass

# Child class 1 inheriting from Animal
class Dog(Animal):
    def speak(self):
        return "Woof!"

# Child class 2 inheriting from Animal and Flyable
class Bird(Animal, Flyable):
    def speak(self):
        return "Chirp!"

    def fly(self):
        return "Flying high!"

# Child class 3 inheriting from Animal and Flyable
class Bat(Animal, Flyable):
    def speak(self):
        return "Screech!"

    def fly(self):
        return "Gliding gracefully!"

# Instantiate objects of child classes
dog = Dog()
bird = Bird()
bat = Bat()

# Call methods of child classes
print(dog.speak())  # Output: Woof!
print(bird.speak()) # Output: Chirp!
print(bird.fly())   # Output: Flying high!
print(bat.speak())  # Output: Screech!
print(bat.fly())    # Output: Gliding gracefully!
```

2.8 Attribute Encapsulation

Attribute encapsulation describes the idea of bundling attributes and methods that work on those attributes within a class. It is used to hide the attributes inside a class like in a capsule, preventing unauthorized parties' direct access to them. Publicly accessible methods are provided in the class to access the values, and other objects call those methods to retrieve and modify the values within the object. This can be a way to enforce a certain amount of privacy for the attributes.

Attribute encapsulation can be achieved using properties, which are a type of special methods called "decorators" that allow defining getter, setter, and deleter methods for class attributes. By using properties, you can control access to attributes and enforce constraints on their values.

2.8.1 Getter, Setter and Deleter methods

Here's a Player class implementation that includes attributes for name, age, level, score, and a protected attribute `weight`. The `weight` attribute has getter, setter, and deleter methods to encapsulate its access.

```python
class Player:
    def __init__(self, name, age, level, score, weight):
        self.name = name
        self.age = age
        self.level = level
        self.score = score
        self._weight = weight  # Note: Weight is a protected attribute

    @property
    def weight(self):
        return self._weight

    @weight.setter
    def weight(self, value):
        if value < 0:
            raise ValueError("Weight cannot be negative")
        self._weight = value

    @weight.deleter
    def weight(self):
        print("Weight is being reset.")
        del self._weight


# Example usage
player1 = Player("Alice", 25, 5, 1000, 70)
print("Player 1 Weight:", player1.weight)  # Output: 70

player1.weight = 75
print("Player 1 Weight:", player1.weight)  # Output: 75

del player1.weight
# print("Player 1 Weight:", player1.weight)  # AttributeError
```

2.8.2 Name Mangling

Using a single underscore prefix (e.g., `_weight`) is conventionally used to indicate that an attribute is intended to be private, but it's not enforced by the language itself. Double underscores (e.g., `__weight`) invoke name mangling, which changes the name of the attribute to include the class name, thus making it harder to access from outside the class.

```python
class Player:
    def __init__(self, name, age, level, score, weight):
        self.name = name
        self.age = age
        self.level = level
        self.score = score
        self.__weight = weight


bob = Player("Bob", 30, 7, 1200, 75)

# Accessing __weight directly using name mangling pattern
print(bob._Player__weight)
```

Name mangling is a technique used to avoid unintentional conflicts in the namespace. It involves prefixing the name of an identifier with double underscores but not ending with more than one trailing underscore.

2.8.3 Public vs. Protected vs. Private

Public attributes/methods are openly accessible, protected attributes/methods are intended for internal use with a gentle reminder, and private attributes/methods are meant to be truly private with limited direct access.

- **Public Attributes/Methods:** These are accessible from outside the class.
- **Protected Attributes/Methods:** By convention, prefixed with a single underscore.
- **Private Attributes/Methods:** These are prefixed with a double underscore.

```python
class Cat:
    def __init__(self, name, age):
        self._name = name  # protected attribute
        self.__age = age   # private attribute

    def _meow(self):  # protected method
        print(f"{self._name} says: Meow!")

    def get_age(self):
        return self.__age

    def set_age(self, new_age):
        if new_age >= 0:
            self.__age = new_age
        else:
            print("Age cannot be negative!")
```

2.9 Shallow and Deep Copy Operations

2.9.1 The `copy()` and `deepcopy()` Methods

Copying objects can be achieved through shallow or deep copying mechanisms. Shallow copying creates a new object with references to the original elements, allowing changes to be reflected in both copies. Deep copying, however, generates a completely independent copy of the original object, ensuring changes made to one do not affect the other.

- **Shallow Copy**
  - A shallow copy creates a new object, but instead of copying the elements recursively, it copies only the references to the elements. In other words, it creates a new container object (like a list or dictionary) but inserts references to the original elements.
  - Changes made to the original elements will be reflected in the shallow copy, and vice versa. However, changes made to the container itself (e.g., adding or removing elements) will not be reflected in the other copy.
  - Shallow copies are created using the `copy()` method or the `copy.copy()` function.
- **Deep Copy**
  - A deep copy, on the other hand, creates a new object and recursively copies all the elements within the object and any nested objects it contains. It means it creates a completely independent copy of the original object.
  - Changes made to the original object or its elements will not affect the deep copy, and vice versa. The deep copy is completely isolated from the original object.
  - Deep copies are created using the `copy.deepcopy()` function.

```python
import copy

# Original list with nested list
original_list = [[1, 2, 3], [4, 5, 6]]

# Shallow copy
shallow_copied_list = copy.copy(original_list)

# Deep copy
deep_copied_list = copy.deepcopy(original_list)

# Modifying elements in the original list
original_list[0][0] = 100

print("Original List:", original_list)
print("Shallow Copied List:", shallow_copied_list)
print("Deep Copied List:", deep_copied_list)
```

```
Original List: [[100, 2, 3], [4, 5, 6]]
Shallow Copied List: [[100, 2, 3], [4, 5, 6]]
Deep Copied List: [[1, 2, 3], [4, 5, 6]]
```

2.9.2 Object: Label vs. Identity vs. Value

- **Label:** Refers to the name given to an object. When you create an object and assign it a name using a variable, you're essentially giving it a label.
- **Identity:** Refers to the unique identifier assigned to an object by Python. It is the memory address where the object is stored.
- **Value:** Refers to the actual data stored within an object. It represents the content or state of the object.

2.9.3 The `id()` Function and the `is` Operator

The `id()` function returns a unique identifier (an integer) for the specified object. This identifier represents the memory address where the object is stored in memory. The `id()` function is used to obtain the identity of an object.

- **`id()` Function:**
  - The `id()` function returns a unique identifier (an integer) for the specified object. This identifier represents the memory address where the object is stored in memory.
  - The `id()` function is used to obtain the identity of an object.
  - **Syntax:** `id(object)`
- **`is` Operator:**
  - The `is` operator is used to compare the identities of two objects. It returns `True` if the two objects have the same identity (i.e., they are stored at the same memory address), and `False` otherwise.
  - The `is` operator is used to check if two variables reference the same object in memory.
  - **Syntax:** `object1 is object2`

```python
# Define two variables with the same value
x = 10
y = 10

# Print the identity and value of each variable
print("Variable x - Identity:", id(x))
print("Variable y - Identity:", id(y))
print("Variable x is y:", x is y)
```

```
Variable x - Identity: 140735477355592
Variable y - Identity: 140735477355592
Variable x is y: True
```

2.10 Metaprogramming

2.10.1 The type() Function

The `type()` function is a built-in function that serves multiple purposes. Primarily, it can be used to obtain the type of an object or dynamically create classes. When `type()` is called with three arguments, it creates a new class. The arguments are as follows:

- **name**: A string specifying the name of the class.
- **bases**: A tuple specifying the base classes (optional, defaults to an empty tuple).
- **dict**: A dictionary containing the attributes and methods of the class.

```
MyClass = type('MyClass', (), {'x': 5})

obj = MyClass()
print(obj.x)   # Output: 5
```

2.10.2 Metaclasses

Metaclasses are a feature that enable customization of class creation. Classes themselves are objects, and the class of a class is referred to as its metaclass. Metaclasses allow for control over the process of creating classes, providing a means to modify or extend this process according to specific requirements.

- **Meaning**: Metaclasses, as the name suggests, are classes for classes. Metaclasses provide a way to define how classes behave, just as classes define how instances behave.
- **Purpose**: The primary purpose of metaclasses is to customize the creation of classes. This customization can involve modifying attributes and methods, controlling inheritance, enforcing coding conventions, or even altering the behavior of instances of those classes.
- **Usage**: Used in advanced scenarios where fine-grained control over class creation is required.

```
class MyMeta(type):
    def __new__(cls, name, bases, dct):
        # Customize the class creation process
        dct['extra_attribute'] = 42
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=MyMeta):
    pass

# MyClass will have the init attribute added by MyMeta
print(MyClass.extra_attribute)  # Output: 42
```

In Python, `__new__()` is the first method to be called in an object's instantiation process, even before `__init__()`. It's responsible for creating and returning the new object.

The following code demonstrates an alternative to creating a metaclass with the class attribute `extra_attribute` that achieves the same behavior.

```python
class MyMeta(type):
    def __new__ (mcs, name, bases, dictionary):
        obj = super().__new__(mcs, name, bases, dictionary)
        obj.extra_attribute = 42
        return obj

class MyClass(metaclass=MyMeta):
    pass

# MyClass will have the init attribute added by MyMeta
print(MyClass.extra_attribute)   # Output: 42
```

2.10.3 Special Attributes

Classes and instances have special attributes that provide information about them. Here are some of these special attributes:

- __**name**__: Prints the name of the class
- __**class**__: Prints the class of the instance
- __**bases**__: Prints the base classes of the class, showing the inheritance
- __**dict**__: Prints the instance attributes as a dictionary

```python
class Animal:
    def __init__(self, breed):
        self.breed = breed

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(breed)
        self.name = name

# Creating an instance of Dog
bello = Dog("bello", "Golden Retriever")

# Printing class name
print(bello.__class__.__name__)    # Output: Dog
print(Dog.__name__)                # Output: Dog
print(type(bello).__name__)        # Output: Dog

# Printing other special attributes
print("Class:", bello.__class__)   # Output: <class '__main__.Dog'>

print("Base classes:", bello.__class__.__bases__)
# Output: (<class '__main__.Animal'>,)

print("Instance attributes:", bello.__dict__)
# Output: {'breed': 'Golden Retriever', 'name': 'bello'}
```

2.10.4 Operating with Metaclasses

Operating with metaclasses involves customizing the creation and behavior of classes.

```python
class DogMeta(type):
    def __new__(cls, name, bases, dct):
        dct['sound'] = "Woof"

        def what_species(cls):
            print(f"This is a {cls.species}")

        dct['what_species'] = classmethod(what_species)

        return super().__new__(cls, name, bases, dct)

    def num_legs(cls):
        return 4


class Dog(metaclass=DogMeta):
    species = "Canis lupus familiaris"

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

    def bark(self):
        print(f"{self.name} says {self.sound}!")

    @staticmethod
    def num_teeth():
        return 42


# Creating instances of Dog
dog1 = Dog("Buddy")
dog2 = Dog("Max")

# Accessing class variables and calling class methods
print("Dog species:", Dog.species)  # Dog species: Canis lupus familiaris
print("Number of legs:", Dog.num_legs())  # Number of legs: 4
Dog.what_species()  # This is a Canis lupus familiaris
print("Number of teeth:", Dog.num_teeth())  # Number of teeth: 42

# Accessing instance variables and calling instance methods
dog1.bark()  # Buddy says Woof!
dog2.bark()  # Max says Woof!
```

This code defines a metaclass called `DogMeta` and a class called `Dog` that uses this metaclass.

- **DogMeta:** Metaclass controlling creation and behavior of the `Dog` class.
- **Dog:** Represents a dog, defined with `DogMeta`. Contains attributes and methods such as species, bark, and num_teeth.

2.10.5 SingletonMeta

A singleton refers to a design pattern where a class is designed to have only one instance, no matter how many times it is instantiated. This pattern ensures that there's only one object created for a given class, and all further references to that class refer to the same instance.

```python
class SingletonMeta(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]


# Example class using the SingletonMeta metaclass
class DatabaseConnector(metaclass=SingletonMeta):
    def __init__(self, host, port):
        self.host = host
        self.port = port
        # Additional initialization code for database connection


# Usage example
db1 = DatabaseConnector("localhost", 3306)
db2 = DatabaseConnector("example.com", 5432)

# Both instances are the same
print(db1 is db2)  # True
print(db1.host, db1.port)  # example.com 5432
```

In this example, we'll create a metaclass called `SingletonMeta` that ensures only one instance of the class `DatabaseConnector` is created throughout the program.

2.11 (De)serialization of Objects

2.11.1 Object Persistence, Serialization, and Deserialization

Object persistence, serialization, and deserialization are fundamental concepts used for data management and transfer.

- **Object Persistence**
  - **Meaning**: Object persistence refers to the ability of an object to outlive the execution of a program by storing its state.
  - **Purpose**: It enables data to be stored and retrieved beyond the runtime of a program.
  - **Usage**: Commonly used in scenarios like web applications, databases, or file systems where data needs to persist across multiple program executions.
- **Serialization**
  - **Meaning**: Serialization is the process of converting an object into a format that can be stored, transferred, or reconstructed later.
  - **Purpose**: It facilitates the transfer and storage of objects in a standardized format.
  - **Usage**: Commonly used in network communication, data storage, and inter-process communication scenarios.
- **Deserialization**
  - **Meaning**: Deserialization is the reverse process of serialization, involving reconstructing an object from its serialized form.
  - **Purpose**: It allows the recovery of serialized data and reconstruction of objects for use within a program.
  - **Usage**: Typically used in conjunction with serialization to restore objects from serialized data stored in files, databases, or received over a network.

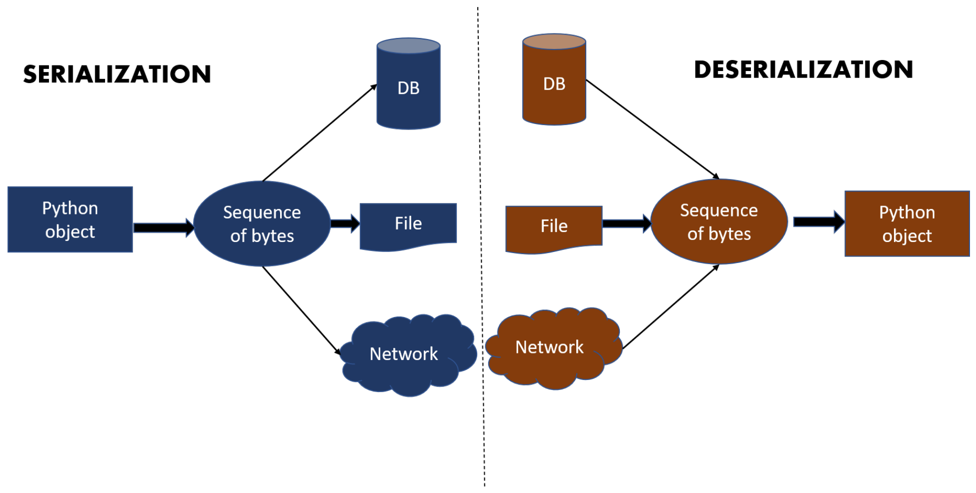


Figure 3: (De)serialization

2.11.2 The `pickle` Module

The `pickle` module provides functions for serializing and deserializing Python objects. It allows objects to be converted into a byte stream, which can then be saved to a file or transmitted over a network. The primary functions provided by the `pickle` module are:

- **pickle.dump()**
  This function is used to serialize an object into a binary file. It takes two arguments: the object to be serialized and a file object (opened in binary write mode 'wb') to which the serialized data will be written. The function then writes the serialized data to the specified file, converting the object into a byte stream that can be saved to disk. Pickle is a module that provides serialization and deserialization functionalities.
- **pickle.load()**
  This function is used to deserialize an object from a binary file previously created using `pickle.dump()`. It takes a file object (opened in binary read mode 'rb') containing the serialized data as input and returns the deserialized object. The function reads the serialized data from the file and reconstructs the original object, effectively reversing the serialization process.
- **pickle.dumps()**
  This function is used to serialize an object into a byte stream representation, but it does **NOT write the data to a file**. Instead, it returns a byte string containing the serialized data.
- **pickle.loads()**
  This function is used to deserialize an object from a byte stream previously created using `pickle.dumps()`. It takes a byte string containing the serialized data as input and returns the deserialized object.

```python
import pickle

class Fruit:
    def __init__(self, name: str, calories: float):
        self.name = name
        self.calories = calories

    def describe(self):
        print(self.name, self.calories, sep=': ')


# Create an instance of the Fruit class
banana: Fruit = Fruit('Banana', 100)

# Serialize the banana object and write it to the 'data.pickle' file
with open('data.pickle', 'wb') as file:
    pickle.dump(banana, file)

# Deserialize the object from the 'data.pickle' file
with open('data.pickle', 'rb') as file:
    fruit: Fruit = pickle.load(file)

fruit.describe() # Output: Banana: 100
```

The `pickle` module supports the serialization of various data types, including:

- **Basic Data Types**
  Python's basic data types such as integers, floats, strings, and booleans can be easily serialized using the `pickle` module.
- **Lists, Tuples, and Dictionaries**
  Lists, tuples, and dictionaries, which are fundamental data structures, can also be serialized using `pickle`. This allows complex data structures to be stored and retrieved as a single object.
- **Custom Objects**
  Custom objects created by the user can also be serialized using `pickle`, provided that they are properly defined and their attributes are serializable. This allows entire object graphs to be stored and reconstructed later.
- **Functions and Classes**
  Functions and classes can be pickled as well, although there are limitations on what can be serialized. For instance, functions and classes defined at the top level of a module can typically be pickled, but dynamically created functions or classes may not be picklable.

2.11.3 The shelve Module

The `shelve` module provides a simple interface for managing persistent storage of Python objects. It allows you to store and retrieve objects from a file using key-value pairs, similar to a dictionary. It is particularly useful for applications that need to store data between program executions without requiring the complexity of a relational database.

```python
import shelve

with shelve.open('TestDB') as db:
    db['one'] = 1
    db['two'] = 2
    db['three'] = 3

with shelve.open('TestDB') as db:
    print(type(db))
    print(dict(db))
    print(db['three'])
```

```
<class 'shelve.DbfilenameShelf'>
{'one': 1, 'two': 2, 'three': 3}
3
```

Note: The `shelve` module uses the `pickle` module internally for serializing and deserializing Python objects to and from the shelf file.

2.11.4 Storing Dictionaries with shelve

This code defines a `Fruit` class with attributes for name and calories. It creates a dictionary of fruit objects and stores it persistently using the `shelve` module.

```python
import shelve

class Fruit:
    def __init__(self, name, calories):
        self.name = name
        self.calories = calories


data: dict = {
    'apple': Fruit('Apple', 10),
    'banana': Fruit('Banana', 100),
    'orange': Fruit('Orange', 50)
}

# Store fruits data
with shelve.open('FruitsDB') as db:
    db.update(data)
```

2.11.5 Retrieve Data from the shelf

This code retrieves data from a persistent storage called a ”shelf”, specifically from a shelf file named `FruitsDB`.

```python
import shelve

with shelve.open('FruitsDB') as db:
    # Retrieve the apple object
    apple: Fruit = db['apple']

    # Retrieve the banana object using the .get() method
    banana = db.get('banana')
    lemon = db.get('lemon')  # Returns None if the key does not exist

    # If key doesn't exist, it will add it with a default value of -1
    pomegranate = db.setdefault('pomegranate', -1)


# Prints the calories count of apple
print("Calories in an apple:", apple.calories)  # Output: 10
```

2.12 Exceptions

2.12.1 Errors

Errors can be raised explicitly using the `raise` statement. This feature allows developers to generate exceptions when certain conditions are met, granting greater control over error handling within their code.

```
raise SomeException("Error message")
```

2.12.2 BaseException

BaseException is the base class for all built-in exceptions. It serves as the superclass for all exceptions defined, including both system-defined exceptions (like ZeroDivisionError, TypeError, ValueError, etc.) and user-defined exceptions.

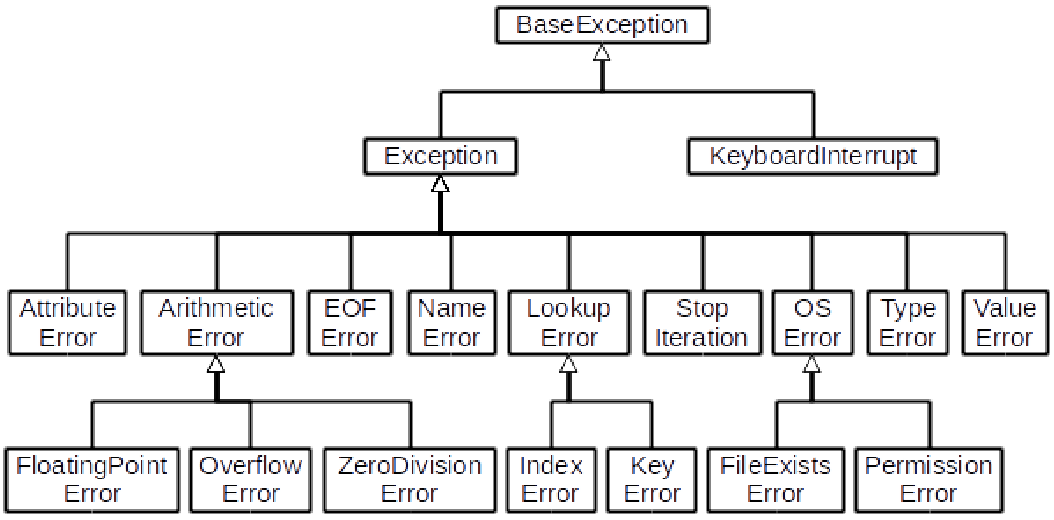


Figure 4: BaseException

2.12.3 Exceptions as Objects

In many programming languages, including Python, exceptions are represented as objects. An exception object encapsulates information about an exceptional event that has occurred during the execution of a program. This object typically contains details such as the type of exception, a message describing the error, and sometimes additional data relevant to the specific exception.

2.12.4 Exceptions

Exceptions are used to handle errors and unexpected situations that may occur during the execution of a program. Using `try` and `except` blocks allows to handle errors, preventing the program from crashing and providing useful feedback to the user. It's a fundamental tool for writing robust and reliable code.

```
try:
    x = int(input("Please enter a number: "))
    result = 10 / x
    print("Result:", result)

except ZeroDivisionError:
    print("Error: Cannot divide by zero!")

except ValueError:
    print("Error: Please enter a valid number!")
```

- **`try`:** The `try` block contains the code that you want to execute. It is the part of the code where you anticipate an error might occur. If an exception occurs within this block, Python will stop executing the code in the `try` block and move to the `except` block.
- **`except`:** The `except` block is used to handle exceptions raised in the preceding `try` block. You can specify the type of exception you expect to occur. If the specified exception occurs in the `try` block, Python will execute the code in the `except` block.

2.12.5 Named Attributes of Exception Objects

Exception objects often have named attributes that provide information about the nature of the exception. These attributes can vary depending on the programming language and the specific exception type, but common attributes include:

- **Type:** Indicates the type of exception that occurred. This could be a built-in exception type provided by the language or a custom exception type defined by the programmer.
- **Message:** A human-readable description of the error that occurred. This message often provides details about what went wrong and can be helpful for debugging.
- **Stack Trace:** A traceback or stack trace provides information about the sequence of function calls that led to the exceptional condition. It includes the file names, line numbers, and function names of the code where the exception occurred, helping developers pinpoint the source of the error.
- **Error Code:** Some exceptions may include an error code that provides additional context or categorization for the error.
- **Additional Data:** Depending on the exception type and the needs of the programmer, exception objects may contain additional data relevant to the exceptional condition. This could include input values, state information, or any other details that might be useful for handling the exception.

2.12.6 Custom Exceptions

Custom exceptions can be created to handle specific cases that are not covered by built-in exceptions.

```python
# Custom exception class by inheriting from the built-in Exception class
class CustomException(Exception):
    def __init__(self, message="An error occurred"):
        self.message = message
        super().__init__(self.message)

# Example function that raises the custom exception
def example_function(number):
    if number < 0:
        # Raise the custom exception if the input is negative
        raise CustomException("Number cannot be negative")

try:
    # Call function with a negative number to trigger the custom exception
    example_function(-5)
except CustomException as e:
    # Handle the custom exception
    print("Custom Exception Caught:", e.message)
```

2.12.7 Chained Exceptions

Chained exceptions, also known as nested exceptions or exception chaining, refer to the practice of capturing and preserving information about one exception while throwing another. This mechanism is particularly useful in scenarios where an error occurs within the context of handling another error.

Many modern programming languages and frameworks support chaining exceptions directly in their exception handling mechanisms. For example you can chain exceptions using the `raise` statement with the `from` keyword:

```python
try:
    # Code that may raise Exception A
    raise Exception("Exception A occurred")
except Exception as e:
    try:
        # Code that may raise Exception B
        raise ValueError("Exception B occurred") from e
    except ValueError as ve:
        # Handle or re-raise Exception B
        print("Caught ValueError:", ve)
```

In the above example, Exception B is raised with Exception A chained to it using the `from` keyword. This preserves the traceback information of Exception A while raising Exception B.

Chained exceptions are beneficial for understanding the flow of errors in a program and can greatly assist in debugging complex systems. However, it's essential to use them judiciously and avoid excessive nesting, which can make the code harder to follow.

2.12.8 The `__cause__` and `__context__` Attributes

The `__cause__` and `__context__` are attributes of the exception object (`ve` in this case) that provide information about the relationships between exceptions.

- **`__cause__`**: This attribute refers to the exception that caused the current exception to occur. It is used to indicate that the current exception was raised in direct response to another exception. In the given code, `ve.__cause__` prints the original exception (`Exception A`) that caused the `ValueError` exception (`Exception B`) to be raised.
- **`__context__`**: This attribute refers to the exception that was being handled when the current exception was raised. It provides context for the current exception. In the given code, `ve.__context__` would print `None` because there's no explicitly defined context for the `ValueError` exception. If there were an outer `except` block handling another exception, the exception object from that outer block would be accessible via `__context__`.

```python
try:
    # Code that may raise Exception A
    raise Exception("Exception A occurred")
except Exception as e:
    try:
        # Code that may raise Exception B
        raise ValueError("Exception B occurred") from e
    except ValueError as ve:
        # Handle or re-raise Exception B
        print("Caught ValueError:", ve)
        print(ve.__cause__)
        print(ve.__context__)
```

```
Caught ValueError: Exception B occurred
Exception A occurred
Exception A occurred
```

2.12.9 Implicitly and Explicitly Chained Exceptions

Chained exceptions, both implicit and explicit, provide a way to maintain a history of exceptions that have occurred during program execution. This feature helps developers to debug and understand the flow of exceptions in their code more effectively.

- **Implicitly Chained Exceptions**
  Implicitly chained exceptions occur when an exception is raised within the except block without explicitly specifying the original exception as the cause. In this case, Python automatically chains the new exception to the original one.
- **Explicitly Chained Exceptions**
  You can explicitly raise a new exception while capturing the original exception to provide more context. This is done using the `raise ... from ...` syntax.

2.12.10 The `__traceback__` Attribute

The `__traceback__` attribute of an exception object contains the traceback information associated with that exception. The traceback provides a detailed record of the execution path that led to the exception, including the sequence of function calls and the line numbers where the exception occurred.

```python
import traceback

def function_with_error():
    # Some code that may raise an exception
    raise ValueError("An error occurred in function_with_error")

try:
    function_with_error()
except ValueError as e:
    # Accessing the __traceback__ attribute
    tb = e.__traceback__
    # Printing the traceback using the traceback module
    print(traceback.format_tb(tb))
```

The traceback module provides functions to work with traceback objects.

- `traceback.print_tb(tb)`: Prints the traceback information to the standard output.
- `traceback.format_tb(tb)`: Returns a formatted string containing traceback information.
- `traceback.extract_tb(tb)`: Returns a list of tuples representing traceback information. Each tuple contains filename, line number, function name, and source line.

2.12.11 Operating with different kinds of Exceptions

The code efficiently handles potential errors by incorporating separate blocks for different types of exceptions, ensuring readability through clear separation and descriptive comments. Additionally, it adheres to Python's PEP 8 style guide for code layout, enhancing overall clarity and maintainability.

```python
try:
    # Your code that may raise exceptions
    x = int(input("Enter a number: "))
    y = 10 / x
    print("Result:", y)

except ValueError:
    # Handle ValueError (invalid input)
    print("Please enter a valid integer.")

except ZeroDivisionError:
    # Handle ZeroDivisionError (division by zero)
    print("Cannot divide by zero.")

except Exception as e:
    # Catch any other unexpected exceptions
    print("An error occurred:", e)
```

3 Coding Conventions, Best Practices, and Standarization

3.1 PEPs

Python Enhancement Proposals (PEPs) is an online document, which is a collection of guidelines, best practices, descriptions of (new) features and implementations, as well as processes, mechanisms and important information surrounding Python. There are many PEPs, hundreds of them.

- **PEP 0 - Index of Python Enhancement Proposals (PEPs)**
- **PEP 1 - PEP Purpose and Guidelines: Information about the purpose of PEPs, their types, and introduces general guidelines**
- **PEP 8 - Style Guide for Python Code: Conventions and best practices for Python coding**
- **PEP 20 - The Zen of Python: List of principles for Python's design**
- **PEP 257 - Docstring Conventions: Provides guidelines for conventions and semantics associated with Python docstrings**
- **PEP 484 - Type Hints: Introduces a syntax for specifying type annotations in Python code**

3.2 PEP 1 – PEP Purpose and Guidelines

3.2.1 What is a PEP?

PEP stands for Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. The PEP should provide a concise technical specification of the feature and a rationale for the feature. We intend PEPs to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

Because the PEPs are maintained as text files in a versioned repository, their revision history is the historical record of the feature proposal. This historical record is available by the normal git commands for retrieving older revisions, and can also be browsed on GitHub.

3.2.2 PEP Audience

The typical primary audience for PEPs are the core developers of the CPython reference interpreter and their elected Steering Council, as well as developers of other implementations of the Python language specification.

However, other parts of the Python community may also choose to use the process (particularly for Informational PEPs) to document expected API conventions and to manage complex design coordination problems that require collaboration across multiple projects.

3.2.3 PEP Types

There are three kinds of PEP:

- **Standards Track PEP:** Describes a new feature or implementation for Python. It may also describe an interoperability standard that will be supported outside the standard library for current Python versions before a subsequent PEP adds standard library support in a future version.
- **Informational PEP:** Describes a Python design issue, or provides general guidelines or information to the Python community, but does not propose a new feature. Informational PEPs do not necessarily represent a Python community consensus or recommendation, so users and implementers are free to ignore Informational PEPs or follow their advice.
- **Process PEP:** Describes a process surrounding Python, or proposes a change to (or an event in) a process. Process PEPs are like Standards Track PEPs but apply to areas other than the Python language itself. They may propose an implementation, but not to Python's codebase; they often require community consensus; unlike Informational PEPs, they are more than recommendations, and users are typically not free to ignore them. Examples include procedures, guidelines, changes to the decision-making process, and changes to the tools or environment used in Python development. Any meta-PEP is also considered a Process PEP.

3.3 PEP 8 – Style Guide for Python Code

PEP 8 is a guide for writing Python code that enhances its readability and maintainability. It provides guidelines for formatting, naming conventions, code structure, and documentation.

Code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 says, “Readability counts”.

3.3.1 Key Points from PEP 8

- **Indentation:** Use 4 spaces per indentation level. Spaces are the preferred indentation method. Tabs should be used solely to remain consistent with code that is already indented with tabs. Python disallows mixing tabs and spaces for indentation.
- **Line Length:** Limit lines to 79 characters to enhance readability. For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters.
- **Blank Lines:** Use blank lines to separate functions, classes, and sections of code logically.
- **Imports:** Import statements should be on separate lines and grouped in the following order: standard library imports, related third-party imports, and local application/library specific imports.
- **Whitespace:** Use whitespace to improve readability, but avoid excessive use.
- **Naming Conventions:** Follow naming conventions for variables, functions, and classes to maintain consistency.
- **Comments:** Write comments to explain non-obvious code, but avoid redundant comments.
- **Docstrings:** Use docstrings to document modules, functions, classes, and methods.
- **Coding Recommendations:** Follow best practices for coding, such as using exception handling properly and preferring exceptions to returning `None`.
- **Version Bookkeeping:** Use `__version__` for version information in modules.
- **Line Break Before or After Binary Operator:** It is permissible to break before or after a binary operator, as long as the convention is consistent locally. For new code it is suggested, where a line break occurs before binary operators for improved readability.
- **String Quotes:** Single-quoted strings and double-quoted strings are the same. This PEP does not make a recommendation for this. Pick a rule and stick to it. When a string contains single or double quote characters, however, use the other one to avoid backslashes in the string. It improves readability.
- **Line Continuation:** If implicit line continuation is not possible, such as when breaking a long string or expression outside of parentheses, you can use a backslash at the end of the line to indicate that it continues on the next line.

3.3.2 Naming Conventions

Following naming conventions is essential for writing clean, readable, and maintainable code. Following naming conventions helps make your Python code more consistent and understandable for yourself and others who may read or work with your code.

- **Variables and function names**
  - Use **snake_case**, lowercase letters with words separated by underscores for variable names and function names.
  - Example: `my_variable`, `calculate_average()`
- **Constants**
  - Use **SCREAMING_SNAKE_CASE**, uppercase letters with words separated by underscores for constant names.
  - Example: `MAX_VALUE`, `PI`
- **Class names**
  - Use **CamelCase** for class names, starting with an uppercase letter.
  - Example: `MyClass`, `Person`
- **Package and module names**
  - Use lowercase letters for package and module names, avoiding underscores if possible.
  - Example: `mypackage`, `my_module`, `utilities`, `data_processing`

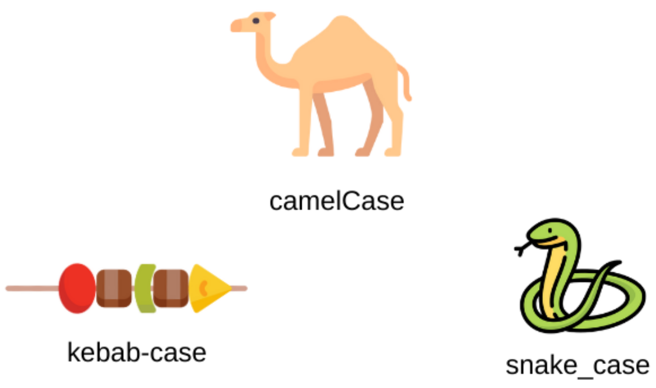


Figure 5: Cases

3.3.3 Comments

Block Comments

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a comment marker # and a single space, unless it is indented text inside the comment. Paragraphs inside a block comment are separated by a line containing a single comment marker.

```python
# This is a block comment
# This function calculates the area of a circle.
# It takes the radius of the circle as input and returns the calculated area.
def calculate_area(radius):
    pi = 3.14159
    area = pi * radius ** 2
    return area
```

Inline comments

- Use inline comments sparingly.
- An inline comment is a comment on the same line as a statement.
- Should be separated by at least two spaces from the statement.
- They should start with a # and a single space.
- Inline comments are unnecessary and in fact distracting if they state the obvious.

3.3.4 Compliant Checker

A PEP8 compliant checker is a tool used to ensure that Python code conforms to the guidelines outlined in PEP8, which is the Python Enhancement Proposal that provides conventions for writing Python code in a consistent and readable manner. PEP8 covers aspects such as indentation, line length, import formatting, naming conventions, and more.

There are several tools available for checking PEP8 compliance of Python code, including:

- **pycodestyle**: Formerly known as pep8, this is a tool to check Python code against the PEP8 style guide. It can be run from the command line.
- **flake8**: Combines several tools in one package, including pycodestyle, pyflakes (for static code analysis), and McCabe (for complexity checking). It provides a comprehensive tool for checking both PEP8 compliance and potential errors in your code.
- **Black**: While not strictly a PEP8 checker, Black is a code formatter that automatically reformats Python code to comply with PEP8 standards. It enforces a consistent style by rewriting code to adhere to its strict rules.
- **PyLint**: Another popular tool for analyzing Python code for errors, it also provides checks for PEP8 compliance among other things.

3.4 PEP 20 – The Zen of Python

PEP 20 is titled ”The Zen of Python” and was authored by Tim Peters. It serves as a set of guiding principles or philosophies for the design of the Python programming language. It encapsulates the core values and ideals that Python's creators and community strive to uphold in their programming practices. The aphorisms, or principles, outlined in PEP 20 are:

- 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!

3.4.1 Easter Egg: `import this`

The term ”Easter Egg” refers to a hidden feature or message that developers embed within the code for amusement or as a tribute. The term is derived from the tradition of hiding Easter eggs for children to find during Easter egg hunts.

The code snippet ”`import this`” is a well-known Easter Egg. When executed in a Python interpreter, it displays ”The Zen of Python,” which is a collection of aphorisms that capture the guiding principles of Python's design philosophy. These aphorisms were written by Tim Peters, and they provide insights into the mindset and values of Python developers.

3.5 PEP 257 – Docstring Conventions

PEP 257 provides conventions for writing docstrings. Docstrings are documentation strings that appear as the first statement in a module, function, class, or method definition. The key points from PEP 257 are:

- Docstrings should be enclosed in triple quotes, e.g. `"""This is a docstring."""`.
- For a multi-line docstring, the opening and closing quotes should be on separate lines.
- The first line of a docstring should be a summary of the object's purpose, starting with a capital letter and ending with a period.
- If the docstring has multiple paragraphs, subsequent paragraphs should be separated by a blank line.
- Docstrings should be written in complete sentences and follow proper grammar and punctuation rules.
- Docstrings should include information about the function or method's parameters, return values, exceptions raised, and any side effects.
- For module-level docstrings, they should include an overview of the module's contents and usage examples.
- For class docstrings, they should describe the class's purpose, behavior, and usage, including information about constructor arguments and instance variables.

3.5.1 One-line Docstrings

One-liners are for really obvious cases. They should really fit on one line.

```python
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
    ...
```

- Triple quotes are used even though the string fits on one line. Makes it easy to expand it.
- The closing quotes are on the same line as the opening quotes. Looks better for one-liners.
- There's no blank line either before or after the docstring.
- The docstring is a phrase ending in a period. It prescribes the function or method's effect as a command ("Do this", "Return that"), not as a description; e.g. don't write "Returns the pathname ...".
- The one-line docstring should NOT be a "signature" reiterating the function/method parameters (which can be obtained by introspection).

3.5.2 Multi-line Docstrings

Multi-line docstrings consist of a summary line just like a one-line docstring, followed by a blank line, followed by a more elaborate description. The summary line may be used by automatic indexing tools; it is important that it fits on one line and is separated from the rest of the docstring by a blank line. The summary line may be on the same line as the opening quotes or on the next line. The entire docstring is indented the same as the quotes at its first line (see example below).

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
    ...
```

3.5.3 Linters and Fixers

Linters and fixers are tools used in software development to analyze source code for potential errors, bugs, stylistic inconsistencies, and adherence to coding standards. They automate the process of code review by identifying issues and providing feedback to developers, helping ensure code quality, consistency, and maintainability.

- **Linter**
  - Static code analysis tool used to flag programming errors, bugs, stylistic errors, and suspicious constructs in the source code without actually executing it.
  - Maintain a consistent coding style within a project and can catch common programming mistakes early in the development process.
  - Commonly used in software development to improve code quality, readability, and maintainability.
- **Fixer**
  - Tool that automatically corrects or adjusts code based on the suggestions and warnings provided by a linter.
  - Automatically apply changes to the code, such as formatting corrections, removing unused imports, or fixing syntax errors.
  - Streamline the development process by automating repetitive tasks and ensuring that the codebase adheres to the defined coding standards.

3.6 PEP 484 – Type Hints

In Python, ”types” refer to the classification of data into different categories, such as integers, strings, lists, etc. Python is known for its dynamic typing system, which means that the type of a variable is inferred at runtime based on the value it holds. This is in contrast to statically typed languages like Java or C++, where you must declare the type of a variable explicitly.

3.6.1 Dynamic Typing

Python’s dynamic typing offers flexibility and simplicity, allowing developers to write code more quickly and concisely. However, it also means that type errors may only be discovered at runtime, leading to potential bugs that can be harder to catch during development.

To address this issue, Python introduced optional static typing through PEP 483 and PEP 484. These proposals introduced type hints, a syntax for annotating variables, function parameters, and return values with their expected types.

```python
# Import the typing module for type hints
from typing import List

# Type hints for variables
age: int = 30
names: List[str] = ["Alice", "Bob", "Charlie"]

# Functions with type hints for parameters and return value
def greet(name: str) -> str:
    return f"Hello, {name}"

def upper_everything(elements: list[str]) -> list[str]:
    return [el.upper() for el in elements]
```

3.6.2 Type Checker: Mypy

Mypy is an optional static type checker for Python that aims to combine the benefits of dynamic (or ”duck”) typing and static typing. Mypy combines the expressive power and convenience of Python with a powerful type system and compile-time type checking. Mypy type checks standard Python programs; run them using any Python VM with basically no runtime overhead.

Using mypy alongside your Python development workflow can help catch type-related errors early, improve code maintainability, and enhance collaboration, especially in larger projects or teams where type clarity is beneficial.

```
# Install mypy
pip install mypy

# Run mypy on a single Python script or module
mypy your_script.py

# Run mypy on an entire directory (recursively)
mypy your_directory/
```

4 GUI Programming

4.1 Graphical User Interface (GUI)

A GUI, pronounced as ”gooey”, stands for Graphical User Interface. The primary rationale behind GUIs is to enhance user experience by providing visual representations of program functions and data. GUIs enable users to interact with software applications through graphical elements like windows, buttons, menus, and icons, rather than relying solely on text-based commands.

GUI programming involves creating visually appealing interfaces for software applications using graphical elements like windows, buttons, and menus. Understanding basic GUI concepts and terminology is essential for developing user-friendly and intuitive applications.

4.1.1 Visual Programming

Visual programming involves creating applications using graphical elements rather than writing code manually. Examples of visual programming environments include Scratch, Blockly, and LabVIEW. These environments typically provide drag-and-drop interfaces for building applications, making it easier for users to create software without extensive programming knowledge.

4.1.2 Widgets Toolkits

Widget toolkits, also known as **GUI toolkits** or **frameworks**, provide libraries of reusable graphical components (”widgets”) for building GUI applications. These toolkits offer APIs (Application Programming Interfaces) for creating and managing GUI elements, handling user interactions, and rendering graphical content. Examples of popular widget toolkits include Tkinter (for Python), Swing (for Java), WinForms (for .NET), and Qt (for C++ and Python).

Widgets, also known as controls, are graphical elements used in GUIs to interact with users or display information. Some basic terms associated with widgets include:

- **Windows:** Rectangular areas on the screen that contain application content and controls.
- **Title and Title Bars:** The title of a window and the bar across the top of a window that contains the title and control buttons (minimize, maximize, close).
- **Buttons:** Controls that trigger actions when clicked, such as submitting a form or closing a window.
- **Icons:** Small graphical representations of files, folders, or applications.
- **Labels:** Text elements used to display static text or descriptions.

4.1.3 Classical vs. Event-Driven Programming

In classical programming, applications follow a sequential flow where each instruction is executed in order. In contrast, event-driven programming relies on events triggered by user actions or system events. Applications respond to these events by executing specific event-handling code. GUI programming is primarily event-driven, where user interactions such as clicking a button or typing in a text box trigger events that drive application behavior.

Events are actions or occurrences that happen during program execution, such as clicking a button, pressing a key, or moving the mouse. GUI applications respond to these events by executing event-handling code associated with each event type. Examples of events include mouse clicks, key presses, window resizing, and timer expiration.

4.2 Tkinter

4.2.1 Simple Tkinter Application

Tkinter is a Python library for creating GUI applications. It is the standard GUI toolkit for Python and is included with most Python installations, making it readily available for developers. Tkinter is based on the Tk GUI toolkit originally developed for the Tcl programming language.

```python
import tkinter as tk

# Function to close the window
def close_window():
    root.destroy()

# Create the main Tkinter window
root = tk.Tk()
root.title("Simple Tkinter Application")

# Create a label widget
label = tk.Label(root, text="Hello, Tkinter!")
label.pack(pady=10)

# Create a button widget
close_button = tk.Button(root, text="Close", command=close_window)
close_button.pack()

# Start the Tkinter event loop
root.mainloop()
```

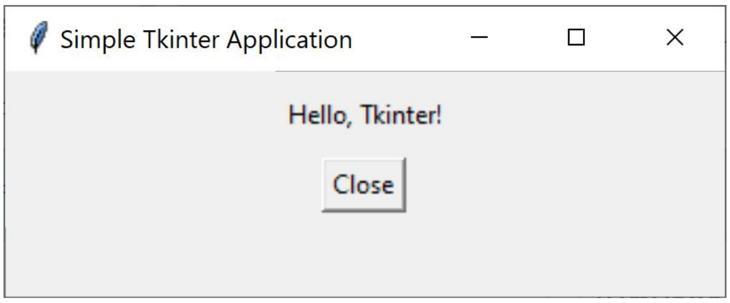


Figure 6: Simple Tkinter Application

4.2.2 The `mainloop()`

In Tkinter, `mainloop()` is a method that runs an event loop, which listens for events such as user input (like mouse clicks and key presses) and system events. It's a fundamental part of any Tkinter application as it keeps the application running and responsive to user interactions.

4.2.3 Tkinter Widgets

A widget is a graphical element or control used to display information or receive input in a GUI.

- **Frame**: A container widget used to organize other widgets.
- **Label**: Displays text or images.
- **Button**: Generates an event when clicked.
- **Checkbutton**: Represents an on/off choice.
- **Radiobutton**: Allows the user to select one option from several.
- **Entry**: Accepts single-line text input from the user.
- **Combobox**: A combination of an Entry widget and a drop-down list.
- **Listbox**: Displays a list of items for selection.
- **Scrollbar**: Allows scrolling through content in widgets like Listbox or Canvas.
- **Sizegrip**: Provides a resize handle for resizable windows.
- **Text**: Multiline text editor.
- **Progressbar**: Displays the progress of a task.
- **Scale**: Represents a range of values as a slider.
- **Spinbox**: Input field with up/down buttons for numeric values.
- **Separator**: Horizontal or vertical line used to separate widgets.
- **Labelframe**: A frame with a title.
- **Panedwindow**: A container widget that holds panes (subwindows) split by a separator.
- **Notebook**: A tabbed notebook-like widget.
- **Canvas**: A drawing area for creating graphics and plots.
- **Treeview**: Displays hierarchical data in a tree-like structure.

4.2.4 Widget Constructors

A widget constructor is a method or function used to create an instance of a GUI component, typically known as a widget.

In frameworks like Tkinter (Python), JavaFX (Java), or PyQt (Python), GUI components like buttons, labels, text fields, etc., are created by calling specific constructor functions or methods provided by the framework. These constructor functions usually take parameters that define the initial properties or attributes of the widget, such as its size, position, text, color, etc. For example, in Tkinter, you might use the `Button` constructor to create a button widget.

```python
import tkinter as tk

root = tk.Tk()

# Create a Button using the Button constructor
button = tk.Button(root, text="Click me", bg="red")

button.pack()

root.mainloop()
```

4.2.5 Widget Properties and Methods

Widget properties and methods provide essential functionalities for customizing and interacting with GUI elements in Tkinter.

- **Common Properties**
  - `text:` Sets or retrieves the text displayed by the widget.
  - `bg:` Sets or retrieves the background color of the widget.
  - `fg:` Sets or retrieves the foreground (text) color of the widget.
  - `font:` Sets or retrieves the font used for the text in the widget.
  - `width:` Sets or retrieves the width of the widget.
  - `height:` Sets or retrieves the height of the widget.
  - `state:` Sets or retrieves the state of the widget (normal, disabled, etc.).
  - `variable:` Sets or retrieves the associated Tkinter variable.

- **Common Methods**
  - `config(**options):` Configures one or more widget options.
  - `grid(**options):` Places the widget in the parent widget using a grid layout.
  - `pack(**options):` Packs the widget into the parent widget.
  - `place(**options):` Places the widget in the parent widget using absolute positioning.
  - `focus():` Sets the keyboard focus to the widget.
  - `bind(event, handler):` Binds an event to a handler function.
  - `destroy():` Destroys the widget and removes it from the display.

4.2.6 Positioning Widgets: `pack()` and `place()` and `grid()`

In Tkinter, you can use various geometry managers to position widgets within a window. The most commonly used managers are `pack()`, `place()`, and `grid()`. If you want to position widgets within the interior of a window, you typically use the grid or place geometry managers.

- **Pack Manager**
  - Simple layout manager that arranges widgets in a block, either horizontally or vertically
  - Widgets are packed into their parent container one after the other, vertically by default
  - Options like side, fill, and expand control how widgets are packed and distributed within the available space
  - Easy to use and suitable for most simple layouts
- **Place Manager**
  - Precise control over the position and size of widgets by specifying exact coordinates
  - Positioning with absolute positioning rather than being packed in relation to each other
  - More precise control but more complex to manage, especially with resizable windows
  - Often used when you need pixel-perfect positioning or when arranging widgets relative to specific locations or other widgets
- **Grid Manager**
  - Divides the window into a grid of rows and columns.
  - You can place widgets at specific row and column coordinates.
  - Offers more control over widget positioning compared to the pack manager.
  - Ideal for complex layouts.

4.2.7 Pack Manager Example

This example demonstrates how to use the pack geometry manager in Tkinter to layout widgets within the application window. Widgets are packed against one side of the container, and in this example, we pack labels and buttons with padding between them.

```python
import tkinter as tk

# Create the main application window
root = tk.Tk()
root.title("Pack Manager Example")

# Pack Manager Example
label1 = tk.Label(root, text="Label 1", bg="lightblue")
label1.pack(padx=10, pady=10)

button1 = tk.Button(root, text="Button 1", bg="lightcoral")
button1.pack(padx=10, pady=10)

# Start the Tkinter event loop
root.mainloop()
```

4.2.8 Place Manager Example

This example illustrates the usage of the place geometry manager, which allows precise positioning of widgets using absolute coordinates. Widgets are placed at specific `x` and `y` coordinates within the window. This manager provides fine-grained control over widget placement.

```
import tkinter as tk

# Create the main application window
root = tk.Tk()
root.title("Place Manager Example")

# Place Manager Example
label1 = tk.Label(root, text="Label 1", bg="lightblue")
label1.place(x=10, y=10)

button1 = tk.Button(root, text="Button 1", bg="lightcoral")
button1.place(x=10, y=40)

# Start the Tkinter event loop
root.mainloop()
```

4.2.9 Grid Manager Example

This example demonstrates the grid geometry manager, which divides the window into a grid of rows and columns. Widgets are then placed at specific row and column coordinates within this grid. This manager provides more control over widget positioning compared to `pack()`.

```
import tkinter as tk

# Create the main application window
root = tk.Tk()
root.title("Grid Manager Example")

# Create widgets and place them using the grid manager
label1 = tk.Label(root, text="Label 1", bg="lightblue")
label1.grid(row=0, column=0, padx=10, pady=10)

label2 = tk.Label(root, text="Label 2", bg="lightgreen")
label2.grid(row=1, column=0, padx=10, pady=10)

button1 = tk.Button(root, text="Button 1", bg="lightcoral")
button1.grid(row=0, column=1, padx=10, pady=10)

button2 = tk.Button(root, text="Button 2", bg="lightyellow")
button2.grid(row=1, column=1, padx=10, pady=10)

# Start the Tkinter event loop
root.mainloop()
```

4.2.10 Screen Coordinates

Screen coordinates refer to the position of elements relative to the display screen. They are typically expressed as a pair of integers representing the horizontal (x) and vertical (y) distances from a reference point, often the top-left corner of the screen.

```python
import tkinter as tk

root = tk.Tk()

# Get the screen width and height
screen_width = root.winfo_screenwidth()
screen_height = root.winfo_screenheight()

print("Screen width:", screen_width)
print("Screen height:", screen_height)

root.mainloop()
```

4.2.11 Positioning and Sizing Windows

Positioning and sizing windows in GUIs involves specifying their location and dimensions on the screen. This is typically achieved using screen coordinates for positioning and width and height values for sizing. To center a Tkinter window on the screen, you can calculate the coordinates for the top-left corner of the window based on the screen dimensions and the size of your window.

```python
import tkinter as tk

def center_window(window, width, height):
    screen_width = window.winfo_screenwidth()
    screen_height = window.winfo_screenheight()

    x = (screen_width - width) // 2
    y = (screen_height - height) // 2

    window.geometry(f"{width}x{height}+{x}+{y}")

root = tk.Tk()

# Set the width and height of the window
window_width = 400
window_height = 300

# Center the window on the screen
center_window(root, window_width, window_height)

root.mainloop()
```

Positioning elements using screen width involves calculating coordinates relative to the screen's width rather than fixed pixel values. By utilizing screen width-based positioning, developers can create responsive and adaptable layouts that adjust dynamically based on the screen's dimensions.

4.2.12 Event Controllers

Event controllers, also known as **event handlers** or **event listeners**, are components of software systems responsible for managing and responding to various user actions or system events. In GUIs, event controllers play a crucial role in capturing user interactions with the interface elements and triggering actions or responses.

- **Callback functions**
  - Callbacks are passed as an argument to another function or method. The purpose of a callback function is to be executed at a later time, often in response to a specific event or condition.
  - In GUIs callback functions are commonly used to handle events triggered by user interactions with GUI elements such as buttons, menus, or input fields. These functions are passed as arguments to event-binding methods like `command` for buttons or `bind` for other widgets.
  - When the associated event occurs (e.g., a button click), the callback function is invoked or called and executes the specified code or behavior.

- **Closing a window with `destroy()`**
  - The `destroy()` method is used to destroy or close a window or widget. When called on a Tkinter widget, such as a window (`Tk`) or a frame (`Frame`), it removes the widget and all its child widgets from the screen and releases associated system resources.
  - Typically, `destroy()` is used in response to a specific condition or event, such as when the user closes a window or when a certain action is completed.
  - When a window is destroyed, the main event loop (`mainloop()`) exits, and the program terminates.

```python
import tkinter as tk

# Event controller for click button
def click_button():
    print("Button clicked!")  # Callback function for click button

# Event controller for destroy button
def close_window():
    root.destroy()  # Callback function for destroy button

root = tk.Tk()

# Create a button to trigger the close_window callback
click_button = tk.Button(root, text="Click Button", command=click_button)
click_button.pack(padx=20, pady=20)

# Create a button to trigger the close_window callback
destroy_button = tk.Button(root, text="Destroy Button", command=close_window)
destroy_button.pack(padx=20, pady=20)

root.mainloop()
```

4.2.13 Binding Events with bind()

The `bind()` method is used to associate an event with a function. This method allows you to specify what should happen when a particular event occurs on a widget, such as a mouse click, a key press, or a mouse movement.

Mouse button events are represented by strings in a specific format.

- `<Button-1>` represents the left mouse button being clicked.
- `<Button-2>` represents the middle mouse button (if present) being clicked.
- `<Button-3>` represents the right mouse button being clicked.

```
import tkinter as tk

def on_button_click(event):
    print("Button Clicked")

def on_button_release(event):
    print("Button Released")

def on_mouse_enter(event):
    print("Mouse Entered")

def on_mouse_leave(event):
    print("Mouse Left")

root = tk.Tk()
root.title("Identifying and Deriving Events")

# Create a button widget
button = tk.Button(root, text="Click Me")
button.pack(pady=20)

# Bind functions to identify button events
button.bind("<Button-1>", on_button_click)
button.bind("<ButtonRelease-1>", on_button_release)

# Bind functions to identify mouse events
button.bind("<Enter>", on_mouse_enter)
button.bind("<Leave>", on_mouse_leave)

root.mainloop()
```

4.2.14 Dialogs

Dialog boxes are GUI components used to interact with the user and request information or confirmation within an application. They typically appear as pop-up windows and can serve various purposes such as displaying messages, prompting for input, or confirming actions. In Tkinter, different types of dialog boxes can be created using built-in functions from the `tkinter.messagebox` module.

- **Message Box**
  Displays a message and an OK button.

  ```
  import tkinter.messagebox as mb
  mb.showinfo("Title", "Message")
  ```

- **Warning Box**
  Displays a warning message and an OK button.

  ```
  import tkinter.messagebox as mb
  mb.showwarning("Title", "Warning Message")
  ```

- **Error Box**
  Displays an error message and an OK button.

  ```
  import tkinter.messagebox as mb
  mb.showerror("Title", "Error Message")
  ```

- **Question Box**
  Displays a question and provides options for the user to respond with ”Yes”, ”No”, ”Cancel”, or other custom buttons.

  ```
  import tkinter.messagebox as mb
  response = mb.askquestion("Title", "Question")
  ```

- **OK/Cancel Box**
  Asks the user to confirm or cancel an action.

  ```
  import tkinter.messagebox as mb
  response = mb.askokcancel("Title", "Question")
  ```

- **Yes/No Box**
  Asks the user to respond with ”Yes” or ”No”.

  ```
  import tkinter.messagebox as mb
  response = mb.askyesno("Title", "Question")
  ```

- **Retry/Cancel Box**
  Asks the user to retry or cancel an operation.

  ```
  import tkinter.messagebox as mb
  response = mb.askretrycancel("Title", "Question")
  ```

4.2.15 Validity of User Input

Validating user input and handling errors are essential aspects of creating robust and user-friendly applications.

```python
import tkinter as tk
from tkinter import messagebox

def validate_number():
    try:
        value = float(entry.get())
        if value < 0 or value > 100:
            raise ValueError("Value must be between 0 and 100")
        messagebox.showinfo("Success", "Input is valid")
        entry.delete(0, tk.END)  # Clear input if valid
    except ValueError as e:
        messagebox.showerror("Error", str(e))
        entry.delete(0, tk.END)

root = tk.Tk()
root.title("Input Validation Example")

label = tk.Label(root, text="Enter a number between 0 and 100:")
label.pack(pady=10)

entry = tk.Entry(root)
entry.pack(pady=5)

validate_button = tk.Button(root, text="Validate", command=validate_number)
validate_button.pack(pady=5)

root.mainloop()
```

4.2.16 Canvas

A canvas is a widget that provides a drawing surface for creating graphics, diagrams, and other visual elements. It allows you to draw lines, shapes, text, images, and more using various methods and options.

```
import tkinter as tk

def draw_rectangle():
        # Draw a blue rectangle
    canvas.create_rectangle(50, 50, 150, 150, fill="blue")

root = tk.Tk()
root.title("Canvas Example")

# Create a canvas widget
canvas = tk.Canvas(root, width=200, height=200, bg="white")
canvas.pack()

# Draw a rectangle on the canvas when a button is clicked
draw_button = tk.Button(root, text="Draw Rectangle", command=draw_rectangle)
draw_button.pack(pady=10)

root.mainloop()
```

The canvas widget in Tkinter provides several methods for drawing and manipulating graphics on the canvas. Here are some commonly used canvas methods:

- `create_line(x1, y1, x2, y2, ...)`: Draws a straight line segment between the points `(x1, y1)` and `(x2, y2)`, and so on.
- `create_rectangle(x1, y1, x2, y2, ...)`: Draws a rectangle with the top-left corner at `(x1, y1)` and the bottom-right corner at `(x2, y2)`.
- `create_oval(x1, y1, x2, y2, ...)`: Draws an oval or circle inscribed within the rectangle defined by the top-left corner at `(x1, y1)` and the bottom-right corner at `(x2, y2)`.
- `create_polygon(x1, y1, x2, y2, ..., options)`: Draws a polygon with vertices specified by alternating x and y coordinates.
- `create_text(x, y, ...)`: Draws text at the specified coordinates `(x, y)`.
- `create_image(x, y, ...)`: Displays an image at the specified coordinates `(x, y)`.
- `move(item, dx, dy)`: Moves the specified item (e.g., line, shape, text) by the specified delta values `(dx, dy)`.
- `delete(item)`: Deletes the specified item from the canvas.
- `itemconfig(item, ...)`: Modifies the configuration options of the specified item.
- `bind(sequence, callback)`: Binds an event (e.g., mouse click, key press) to a callback function for the canvas.

4.2.17 Coloring Widgets

Coloring widgets in Tkinter enables customization of various elements such as labels, buttons, frames, etc., to align with the desired design of a graphical user interface (GUI). Both the background color (bg) and the foreground color (fg) of widgets can be specified.

```python
import tkinter as tk

root = tk.Tk()
root.title("Color Example")
root.geometry("250x100")

# Set background color for the entire window
root.configure(bg="lightcoral")

# Label with default colors
label_default = tk.Label(root, text="Label", bg="darkblue", fg="green")
label_default.pack(padx=10, pady=10)

# Label with hexadecimal color
label_hex = tk.Label(root, text="Hex Label", bg="#00008B", fg="#90EE90")
label_hex.pack(padx=10, pady=10)

root.mainloop()
```

- **Default Color Names**
  - Tkinter provides a set of predefined color names, allowing you to easily select colors without specifying specific color codes.
  - For example, ”red”, ”blue”, ”lightblue”, etc., are recognized color names in Tkinter.
  - These names correspond to common colors and simplify the process.

- **Hexadecimal Color Codes**
  - Hexadecimal color codes represent colors using a combination of six hexadecimal digits.
  - Each pair of digits corresponds to the intensity of the red, green, and blue components.
  - For example, #FF0000 represents pure red, #00FF00 represents pure green, and #0000FF represents pure blue.
  - Hexadecimal color codes provide a wide range of color options.

- **RGB Values**
  - RGB (Red, Green, Blue) values specify colors based on the intensity of red, green, and blue light components.
  - Each component’s intensity is represented by a value between 0 and 255.
  - For example, (255, 0, 0) represents pure red, (0, 255, 0) represents pure green, and (0, 0, 255) represents pure blue.
  - Tkinter does **NOT** directly support specifying colors using RGB values. RGB values must be converted to hexadecimal format for use in Tkinter.

4.2.18 Variables

In Tkinter, variables are often used to link GUI elements (like widgets) with application data. The common variable types used are `StringVar`, `IntVar`, `DoubleVar`, and `BooleanVar`, each corresponding to a specific data type.

Observable variables, also known as observable objects or observable properties, are data structures that notify their observers when their internal state changes. These variables represent the state of GUI elements (such as widgets or components), and they allow other parts of the application to react to changes in this state.

In scenarios where you need to decouple components of an application observable variables can be useful. For example, in a GUI application, observable variables can represent the state of various GUI elements (like checkboxes, text fields, etc.), and different parts of the application can react to changes in these elements without directly coupling to each other.

```python
from tkinter import Tk, Button, StringVar

root = Tk()
root.geometry('200x200')

str_var = StringVar()
str_var.set("Hello, Tkinter!")

def update_text():
    str_var.set("Button clicked!")

button = Button(root, textvariable=str_var, command=update_text)
button.pack()

root.mainloop()
```

In Tkinter, `textvariable` is a configuration option available for specific widgets, including `Button`, `Label`, `Entry`, `Checkbutton`, and `Radiobutton`. It enables you to connect a Tkinter variable (such as `StringVar`, `IntVar`, etc.) with the widget's content. Consequently, when the linked variable's value changes, the widget's content automatically updates accordingly, and vice versa.

The text on the button changes to ”Button clicked!”. Using the `textvariable` option in Tkinter allows you to dynamically update the text of a widget by associating it with a tkinter variable such as `StringVar()`. This can be particularly useful when you want to update the text displayed on a button dynamically.

```python
import tkinter as tk

class ObservableVariable:
    def __init__(self):
        self._value = ""
        self._observers = []

    def get(self):
        return self._value

    def set(self, value):
        self._value = value
        self.notify_observers()

    def add_observer(self, observer):
        self._observers.append(observer)

    def remove_observer(self, observer):
        self._observers.remove(observer)

    def notify_observers(self):
        for observer in self._observers:
            observer(self._value)

root = tk.Tk()
root.title("Observable Function in Tkinter")

observable_var = ObservableVariable()

# Function to be called whenever the input changes
def input_changed(new_value):
    print("Input changed:", new_value)

# Entry widget
entry = tk.Entry(root)
entry.pack(pady=10, padx=10)

# Bind the input_changed function to the observable variable
observable_var.add_observer(input_changed)

# Function to update the observable variable when the entry content changes
def update_variable(*args):
    new_value = entry.get()
    observable_var.set(new_value)

# Bind the update_variable function
entry.bind("<KeyRelease>", update_variable)

root.mainloop()
```

4.2.19 Radiobuttons

Radio buttons are a type of GUI element that allows users to select one option from a set of mutually exclusive options. They are typically used in forms, settings, and preference panels where users need to make a single selection from a list of choices.

```python
import tkinter as tk

root = tk.Tk()
root.title("Radio Button Example")
root.geometry("300x150")
root.config(pady=10)

def on_change():
    selected = selected_option.get()
    label.config(text="Selected: " + selected)

# Variable to hold the selected radio button
default_selected = "None"
selected_option = tk.StringVar(value=default_selected)

# Create radio buttons in a loop
options=["Option 1","Option 2", "Option 3"]
for o in options:
    rb = tk.Radiobutton(root,
                        text=o,
                        value=o,
                        variable=selected_option,
                        command=on_change)
    rb.pack(anchor=tk.W)

label = tk.Label(text="Selected: " + default_selected)
label.pack(pady=10)

root.mainloop()
```

In the context of the radio buttons, `rb.pack(anchor=tk.W)` means that each radio button will be anchored to the west (left) side of its allocated space. This parameter is optional.

- **`text`:** The `text` parameter in a `Radiobutton` widget specifies the label text that will be displayed alongside the radio button. Each radio button typically represents an option, and `text` is what labels that option.
- **`value`:** In a `Radiobutton` widget, the `value` parameter sets the value associated with that particular radio button. When a user selects a radio button, the corresponding value is assigned to the variable associated with the group of radio buttons.
- **`variable`:** The `variable` parameter in a `Radiobutton` widget associates a `StringVar`, `IntVar`, or `BooleanVar` variable with a group of radio buttons. This association ensures that only one radio button in the group can be selected at a time. When a radio button is selected, its value is stored in the associated variable.
- **`command`:** The `command` parameter in a `Radiobutton` widget specifies a function that will be called when the radio button is selected. This function typically performs some action in response to the selection of the radio button.

4.2.20 Checkboxes

Checkboxes are GUI elements used to allow users to make multiple selections from a set of options. They are used when users need to make one or more selections from a list of options. They are commonly used in forms, preference panels, settings, and other interfaces where users need to indicate their choices among multiple options.

```python
import tkinter as tk

def show_checkbox_state():
    if check_var.get() == 1:
        status_label.config(text="Checkbox is checked")
    else:
        status_label.config(text="Checkbox is unchecked")

root = tk.Tk()
root.title("Checkbox Example")
root.geometry("300x150")

# Variable to hold the state of the checkbox
check_var = tk.IntVar()

# Create a checkbox
checkbox = tk.Checkbutton(root,
                          text="Check me",
                          variable=check_var,
                          command=show_checkbox_state)
checkbox.pack(pady=30)

# Label to display the status of the checkbox
status_label = tk.Label(root, text="")
status_label.pack()

root.mainloop()
```

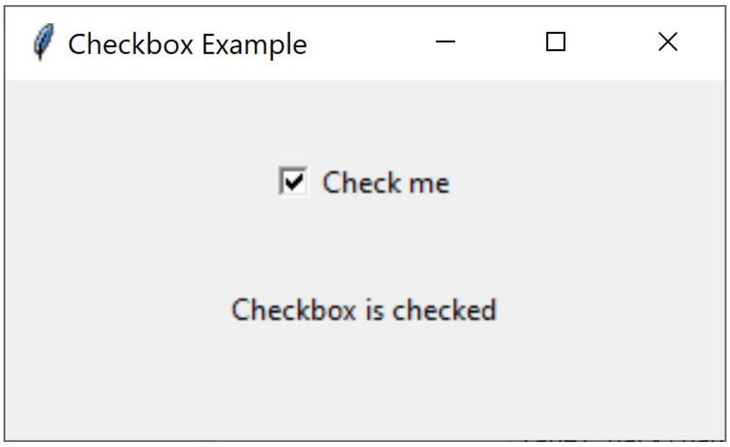


Figure 7: Tkinter

5 Network Programming

5.1 Terms and Networking Concepts

Network programming refers to the practice of creating software that communicates over computer networks. Python provides several modules and libraries that facilitate network communication, making it relatively easy to develop networked applications.

- **REST (Representational State Transfer)**
  REST is a style of software architecture for distributed systems, commonly used in networked applications like web services. It utilizes standard HTTP methods (GET, POST, PUT, DELETE) to manipulate resources, typically represented in formats such as JSON or XML. RESTful services follow principles such as statelessness, uniform interface, and resource-based interactions, making them scalable, flexible, and easily understandable.
- **Network Sockets**
  Network sockets are endpoints for communication between devices over a network. Sockets are used to establish connections and exchange data between a client and a server. They allow bidirectional data flow, enabling applications to send and receive data across networks using protocols such as TCP or UDP. Sockets provide an abstraction for handling network communication, allowing developers to create networked applications efficiently.
- **Protocols**
  Protocols are rules or standards that define how data is transmitted and received between devices on a network. They specify formats for data packets, error handling procedures, and other aspects of communication. Examples of network protocols include TCP/IP (Transmission Control Protocol/Internet Protocol), HTTP (Hypertext Transfer Protocol), and SMTP (Simple Mail Transfer Protocol).
- **Domains**
  Domains refer to the hierarchical naming system used to identify resources on the internet. They are part of the Domain Name System (DNS) and are used to assign memorable names to numerical IP addresses. For example, ”google.com” is a domain name.
- **Addresses**
  Addresses are identifiers used to locate resources on a network, such as computers, servers, or other devices. In the context of the internet, addresses typically refer to IP addresses (Internet Protocol addresses), which are numerical labels assigned to each device connected to a computer network. For example, an IP address might be `192.0.2.1`.
- **Ports**
  Ports are virtual endpoints used by communication protocols to identify specific processes or services on a networked device. They allow multiple services or applications to operate concurrently on a single device. Ports are identified by numbers, and each service typically uses a specific port number. For example, web servers commonly use port 80 for HTTP and port 443 for HTTPS.
- **Services**
  Services refer to the software applications or processes running on a networked device that provide specific functions or features. These functions can include serving web pages, sending emails, transferring files, and more. Each service typically operates using one or more network protocols and may use specific ports for communication. Examples of services include web servers (e.g., Apache, Nginx), email servers (e.g., Microsoft Exchange, Postfix), and file transfer protocols (e.g., FTP, SFTP).

5.1.1 Connection-oriented vs. Connectionless

Network communication refers to the exchange of data between devices like computers, servers, routers, etc. over a network. This communication enables devices to share information, resources, and services. In networking, communication methods play a crucial role in determining how data is transmitted between devices. Two fundamental approaches are connection-oriented and connectionless communication. Connection-oriented communication establishes a dedicated link between sender and receiver for reliable data transfer, as seen in TCP. Conversely, connectionless communication, exemplified by UDP, sends independent packets without a prior connection, prioritizing speed over reliability.

- **Connection-Oriented Communication:** In connection-oriented communication, a dedicated connection is established between the sender and receiver before any data transfer occurs. This connection remains active throughout the entire communication session. Protocols like TCP (Transmission Control Protocol) provide connection-oriented communication, ensuring reliable and ordered delivery of data.
- **Connectionless Communication:** In contrast, connectionless communication does not require a pre-established connection between the sender and receiver. Each data packet is sent independently and may take a different route to reach its destination. Protocols like UDP (User Datagram Protocol) operate in a connectionless manner. While connectionless communication is faster and more efficient for certain types of data, it may result in lost or out-of-order packets.

5.1.2 Protocols: TCP and UDP

A protocol is a set of rules or guidelines that govern how data is transmitted and received between devices, systems, or components in a networked environment. These rules define the format, timing, sequencing, and error handling of data transmission, ensuring that communication occurs in an organized, reliable, and efficient manner.

- **Transmission Control Protocol (TCP)**
  - Connection-oriented protocol
  - Provides reliable, ordered, and error-checked delivery of data
  - Uses acknowledgments and retransmissions for reliability
  - Slower than UDP due to overhead of connection setup and error checking
  - Suitable for applications requiring guaranteed delivery and data integrity, such as web browsing, email, and file transfer
- **User Datagram Protocol (UDP)**
  - Connectionless protocol
  - Provides fast, but unreliable, delivery of data
  - No guarantees on delivery, ordering, or duplicate protection
  - Minimal overhead, making it faster than TCP
  - Suitable for real-time applications, such as streaming media, online gaming, and VoIP, where speed is prioritized over reliability

5.1.3 Clients and Servers

In computer networking, clients and servers play integral roles in facilitating communication and accessing resources over a network. Clients, such as devices, programs, or processes, initiate requests for services or resources from servers.

- **Clients:** Devices, programs, or processes that request services or resources from other devices or programs, known as servers. In a client-server model, clients initiate communication by sending requests to servers. These requests can vary widely depending on the type of service being requested. For example, a web browser acts as a client when it requests web pages from a web server, while an email client requests emails from an email server. Clients typically do not provide services directly to other devices or clients; instead, they rely on servers to fulfill their requests.
- **Servers:** Devices, programs, or processes that provide services, resources, or data to clients over a network. They listen for incoming requests from clients and respond by providing the requested services or resources. Servers can offer a wide range of services, including web hosting, email hosting, file storage, and database management. In a client-server model, servers are responsible for handling incoming requests from clients and delivering the appropriate responses. Unlike clients, servers do not typically initiate communication; instead, they wait for incoming requests from clients and respond accordingly.

The requests can vary depending on the service required, such as web pages or emails. Servers, on the other hand, respond to client requests by providing the requested services or resources. They listen for incoming requests and deliver appropriate responses, offering services like web hosting, email hosting, and file storage. Together, clients and servers form the foundation of the client-server model, enabling efficient communication and resource sharing across networks.

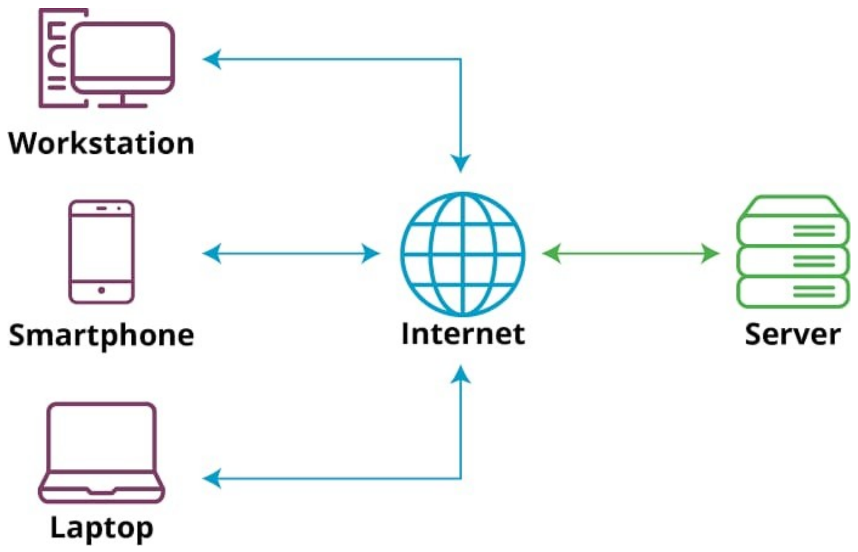


Figure 8: Client server communication

5.2 Socket Programming

Socket programming is a key aspect of network programming that enables communication between processes across a network. It forms the foundation for building various networked applications, allowing computers to exchange data over the internet or within a local network. In socket programming, a socket represents an endpoint for communication between two machines. A socket is an endpoint for communication between two machines. It consists of an IP address and a port number.

- **IP Address:** An IP (Internet Protocol) address uniquely identifies a device on a network. In socket programming, IP addresses are used to establish connections between machines. An IP address is typically represented as a **32-bit number**, which is divided into four octets (each containing 8 bits) separated by periods.
- **Port Number:** Port numbers are used to identify specific processes or services running on a machine. They facilitate multiplexing, allowing multiple applications to use the network simultaneously.

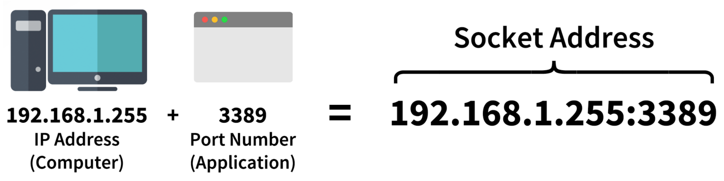


Figure 9: Socket address

5.2.1 Types of Sockets

- **Stream Sockets (TCP):** Stream sockets provide a reliable, connection-oriented communication channel. They ensure that data is delivered in the correct order without loss or duplication. TCP (Transmission Control Protocol) is commonly used for stream socket communication.
- **Datagram Sockets (UDP):** Datagram sockets provide an unreliable, connectionless communication channel. They allow for fast, low-overhead communication but do not guarantee delivery or order of messages. UDP (User Datagram Protocol) is commonly used for datagram socket communication.

```python
import socket

# Creating a TCP socket
tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Creating a UDP socket
udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

By specifying `socket.SOCK_STREAM` for TCP and `socket.SOCK_DGRAM` for UDP, the respective socket types are indicated to the `socket.socket()` function, allowing the creation of TCP or UDP sockets accordingly.

5.2.2 How Sockets Work

server.py

```python
import socket
import threading

SERVER = socket.gethostbyname(socket.gethostname()) # IP address of the server
PORT = 5050  # Port to listen on
ADDR = (SERVER, PORT)  # Address to bind the server
HEADER_SIZE = 64  # Size of the header containing message length
FORMAT = 'utf-8'  # Format to encode/decode messages
DISCONNECT_MSG = "!DISCONNECT"  # Message to disconnect a client

# Create a new socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(ADDR)

# Function to handle client connections
def handle_client(conn, addr):
    print(f"[NEW CONNECTION] {addr} connected")
    connected = True
    while connected:
        # Receive the header containing the message length
        header = conn.recv(HEADER_SIZE).decode(FORMAT)
        if header:
            msg_length = int(header)
            msg = conn.recv(msg_length).decode(FORMAT)
            if msg == DISCONNECT_MSG:
                connected = False
            print(f"[{addr}] {msg}")
            # Send acknowledgment to the client
            conn.send("[MESSAGE RECEIVED]".encode(FORMAT))
    conn.close()

# Function to start the server
def start():
    print(f"[LISTENING] server is listening on {SERVER}")
    server.listen()
    while True:
        # Wait for a new connection and store IP address and port
        conn, addr = server.accept()
        # Create a new thread to handle the client
        thread = threading.Thread(target=handle_client, args=(conn, addr))
        thread.start()
        print(f"[ACTIVE CONNECTIONS] {threading.active_count() - 1}")

if __name__ == "__main__":
    print("[STARTING] server is starting...")
    start()  # Start the server
```

client.py

```python
import socket

# Get the IP address of the server dynamically
SERVER = socket.gethostbyname(socket.gethostname())
# Alternatively, specify IP address manually from ipconfig (IPv4 Address)
# SERVER = "192.168.12.30"
PORT = 5050  # Port on which the server is listening
ADDR = (SERVER, PORT)  # Address of the server
HEADER_SIZE = 64  # Size of the header containing message length
FORMAT = 'utf-8'  # Format to encode/decode messages
DISCONNECT_MSG = "!DISCONNECT"  # Message to disconnect from the server

# Create a new socket for the client
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(ADDR)

def send(message):
    # Encode the string message into bytes object
    message_encoded = message.encode(FORMAT)
    # Calculate the length of the encoded message
    message_length = str(len(message_encoded)).encode(FORMAT)
    # Pad the message length to ensure it matches HEADER_SIZE
    message_length += b' ' * (HEADER_SIZE - len(message_length))
    # Send the padded message length and the encoded message to the server
    client.send(message_length)
    client.send(message_encoded)

    # Receive messages from the server and print them
    print(client.recv(2048).decode(FORMAT))

# Send an initial message to the server
send("Hello, server!")

while True:
    # Continuously prompt the user for input
    msg = input("")

    if msg == "EXIT":
        # Send DISCONNECT_MSG to inform the server and break the loop
        send(DISCONNECT_MSG)
        break

    if msg:
        # If the user input is not empty, send it to the server
        send(msg)
```

5.2.3 Connecting Sockets to HTTP Servers

```python
import socket

def send_http_request(host, port, request):
    # Create a socket object
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        try:
            # Connect to the server
            s.connect((host, port))

            # Send the HTTP request
            s.sendall(request.encode())

            # Receive the response
            response = s.recv(4096)

            # Return the response
            return response.decode()

        except socket.error as e:
            print(f"Socket error: {e}")

        except Exception as e:
            print(f"An unexpected error occurred: {e}")

        finally:
            # Close the connection
            s.close()

# Example usage
if __name__ == "__main__":
    HOST = 'example.com'
    PORT = 80
    REQUEST = 'GET / HTTP/1.1\r\nHost: example.com\r\n\r\n'
    res = send_http_request(HOST, PORT, REQUEST)
    print(res)
```

```python
REQUEST = "GET / HTTP/1.1\r\nHost: www.example.com\r\n\r\n"
```

This line is a string representing an HTTP request. It follows the HTTP protocol format:

- **GET**: It's the HTTP method used to request data from a specified resource.
- /: It's the path of the resource being requested, in this case, it's the root path.
- **HTTP/1.1**: It specifies the version of the HTTP protocol being used.
- **r**\n: It's a carriage return and line feed, used as a line break in HTTP requests.
- **Host: example.com**: It's a header specifying the domain name of the server.
- **r**\n: Another line break.
- **r**\n: An empty line indicating the end of the HTTP request headers.

5.2.4 Exceptions

When working with Python sockets, which are used for network communication, there are several common exceptions. Handling these exceptions appropriately in your code ensures robustness and graceful error recovery during network communication.

- **socket.error** or **socket.timeout**: These exceptions are the most general ones when working with sockets. They can occur for various reasons such as connection issues, timeouts, or other low-level socket errors.
- **ConnectionRefusedError**: This exception occurs when attempting to connect to a remote server that actively refuses the connection. It could indicate that the server is not running or is not accepting connections on the specified port.
- **ConnectionResetError**: This exception occurs when the connection is unexpectedly closed by the peer. It might happen due to network issues or if the peer terminates the connection unexpectedly.
- **BrokenPipeError**: This exception occurs when writing to a socket that has been closed at the other end. It typically happens when the connection is broken before all data has been sent or received.
- **TimeoutError**: This exception occurs when a socket operation times out. It can happen when attempting to establish a connection or during data transmission if there's no response within the specified timeout period.
- **OSError**: This is a general exception for various operating system-related errors. It can include errors related to file descriptors, permissions, or other low-level issues that might affect socket operations.
- **socket.gaierror**: This exception occurs when the address resolution fails. It could happen if the hostname cannot be resolved or if there's an issue with the network configuration.
- **socket.herror**: This exception occurs when a DNS-related error happens. It might occur when the DNS server is unreachable or if there's an issue with the hostname.
- **BlockingIOError**: This exception occurs when an operation would block if the `socket` is in non-blocking mode. It usually happens during non-blocking socket operations.
- **PermissionError**: This exception occurs when there's a permission-related issue, such as trying to bind to a privileged port without sufficient permissions.

5.3 REST Clients

A REST (Representational State Transfer) client is a program or module that sends HTTP requests to a RESTful web service or API to interact with resources on the server. REST is an architectural style for designing networked applications, commonly used in web services development.

There are several libraries and frameworks available for creating REST clients. One popular choice is the `requests` library, which provides a simple and elegant way to send HTTP requests and handle responses.

5.3.1 The `requests` Module

The `requests` module simplifies the process of working with HTTP requests and responses, making it an essential tool for web scraping, web development, and interacting with web APIs.

```python
import requests

# Making a GET request to the XKCD website
r = requests.get('https://xkcd.com/353/')

# Printing the available attributes and methods of the response object 'r'
#print(dir(r))

# Printing the help documentation for the response object 'r'
#print(help(r))

# Making a GET request to download the image from the XKCD website
img = requests.get('https://imgs.xkcd.com/comics/python.png')

# Printing the content of the image (binary data)
print(img.content)

# Printing the status code of the image request
print(img.status_code)

# If the response from the XKCD website is successful
if r.ok:
    # If response is ok, open a file named 'comic.png' in binary write mode
    with open('comic.png', 'wb') as f:
        # Write the binary content of the image to the file
        f.write(img.content)
```

5.3.2 CRUD Operations

CRUD is an acronym that stands for Create, Read, Update, and Delete. It represents the four basic operations that can be performed on data stored in a database or managed by an application:

- **Create**: This operation involves adding new data or records to the database. In a database context, it typically means inserting a new row into a table. In an application context, it could involve creating new objects or entities.
- **Read**: This operation involves retrieving existing data or records from the database. In a database context, it typically means querying the database to retrieve specific data based on certain criteria. In an application context, it could involve fetching and displaying data to the user.
- **Update**: This operation involves modifying existing data or records in the database. In a database context, it typically means updating the values of one or more columns in a specific row. In an application context, it could involve allowing users to edit and update information.
- **Delete**: This operation involves removing existing data or records from the database. In a database context, it typically means deleting a row from a table. In an application context, it could involve allowing users to delete objects or entities.

CRUD operations form the foundation of many applications and are commonly used in various software systems, including web applications, mobile apps, and desktop applications. These operations allow users to interact with and manage data effectively, enabling applications to create, retrieve, update, and delete information as needed.

5.3.3 HTTP Methods: GET, POST, PUT, DELETE

HTTP (Hypertext Transfer Protocol) methods define the actions that a client (such as a web browser) can perform on a server. Each HTTP method is associated with a specific type of action that the client wants to perform on a resource identified by a URL.

- **GET**: Used to retrieve data from a server, such as fetching a web page or an image.
- **POST**: Used for submitting data, uploading files, or making changes to a server's state.
- **PUT**: Used to update existing resources with new data.
- **DELETE**: Used to delete resources from a server, such as deleting a user account or a file.

```python
import requests

# Target URL for HTTP requests: https://httpbin.org/

# GET request with parameters passed directly in the URL
# r = requests.get('https://httpbin.org/get?page=2&count=25') # errorprone

# GET request with params
params = {'page':2, 'count': 25}
r = requests.get('https://httpbin.org/get', params=params)
print(r.url)
print(r.text)

# POST request with form data
form_data = {'username': 'corey', 'password': 'testing'}
r = requests.post('https://httpbin.org/post', data=form_data)
print(r.text)
r_dict = r.json()
print(r_dict['form'])

# GET request with Basic Authentication
auth = ('corey', 'testing')
r = requests.get('https://httpbin.org/basic-auth/corey/testing', auth=auth)
print(r)
print(r.text)

# Example of setting a timeout for a request
r = requests.get('https://httpbin.org/delay/1', timeout=2)
print(r)

# Example causing a timeout
r = requests.get('https://httpbin.org/delay/5', timeout=3)
print(r)
```

5.3.4 Response Status Codes

Response status codes are standardized three-digit integers that HTTP servers use to communicate the outcome of a client's request. They provide information about whether a request was successful, encountered an error, or requires further action. The status code is accompanied by a short, human-readable phrase that provides additional context. HTTP status codes are grouped into five categories, each represented by the first digit of the status code.

- **1xx - Informational:**
  - Request has been received and understood, and that further action is required by the client.
- **2xx - Success:**
  - **200 OK:** The request was successful.
  - **201 Created:** The request has been fulfilled, and a new resource has been created.
  - **204 No Content:** Request has been successfully processed by the server, but it is not returning any content.
- **3xx - Redirection:**
  - Further action is needed by the client to fulfill the request. They typically indicate that the client should take additional steps to complete the request.
- **4xx - Client Error:**
  - **400 Bad Request:** Server cannot process the request due to a client error, like invalid syntax.
  - **401 Unauthorized:** Request requires authentication, but the client has not provided valid credentials.
  - **404 Not Found:** Server cannot find the requested resource.
- **5xx - Server Error:**
  - **500 Internal Server Error:** Server encountered an unexpected condition that prevented it from fulfilling the request.

Response status codes are essential for understanding the outcome of HTTP requests and diagnosing issues with web applications and APIs. By interpreting these status codes, clients can respond appropriately to the Server's response and handle errors or redirections effectively.

5.3.5 Analyzing the Server's Response

Analyzing a server's response with the `requests` library involves examining various attributes of the response object returned by the requests module after making an HTTP request. Here's a basic overview of how to analyze a server's response using requests:

```python
import requests

# Send a GET request
response = requests.get('https://api.example.com/data')

# Print the status code
print('Status Code:', response.status_code)

# Print the response headers
print('Headers:', response.headers)

# Print the response content as text
print('Response Text:', response.text)

# Print the URL of the response
print('URL:', response.url)

# Print any cookies sent by the server
print('Cookies:', response.cookies)

# Convert response content to JSON (if applicable)
try:
    json_data = response.json()
    print('Response JSON:', json_data)
except ValueError:
    print('Response is not in JSON format')

# Print time elapsed between sending the request and receiving the response
print('Elapsed Time:', response.elapsed.total_seconds(), 'seconds')
```

6 File Processing and Communicating with a Program's Environment

6.1 SQLite

6.1.1 SQL

SQL (Structured Query Language) is a domain-specific language used in programming and managing relational databases. It provides a standardized way of interacting with databases for tasks such as querying data, modifying data, defining schema, and managing access control.

- **SELECT**
  The SELECT statement is used to retrieve data from one or more database tables. It can retrieve all columns, specific columns, or rows based on specified conditions.

```
-- Select all columns from a table
SELECT * FROM table_name;

-- Select specific columns from a table
SELECT column1, column2 FROM table_name;

-- Select with conditions
SELECT * FROM table_name WHERE condition;
```

- **INSERT INTO**
  The INSERT INTO statement is used to add new records (rows) into a table.

```
-- Insert into all columns
INSERT INTO table_name VALUES (value1, value2, ...);

-- Insert into specific columns
INSERT INTO table_name (col1, col2, ...) VALUES (val1, val2, ...);
```

- **UPDATE**
  The UPDATE statement is used to modify existing records in a table.

```
-- Update values in a table
UPDATE table_name SET col1 = val1, col2 = val2 WHERE condition;
```

- **DELETE**
  The DELETE statement is used to remove records from a table.

```
-- Delete all records from a table
DELETE FROM table_name;

-- Delete specific records based on condition
DELETE FROM table_name WHERE condition;
```

6.1.2 SQLite

The sqlite module is a built-in library that allows developers to interact with SQLite databases. SQLite is a lightweight, serverless database engine that stores data in a single file, making it convenient for small to medium-scale applications.

6.1.3 Create a database

This Python script connects - or creates if necessary - to an SQLite database named `customer.db`, creates a table named `customers` with three columns `first_name`, `last_name` and `email`, inserts one row, and inserts multiple rows using parameterized queries. Finally, it commits changes and closes the connection.

```python
import sqlite3

# Connect to database, creating it if it doesn't exist
conn = sqlite3.connect('customer.db')

# Create a cursor
c = conn.cursor()

# Create a table with 3 columns
c.execute("""CREATE TABLE customers (
    first_name TEXT,
    last_name TEXT,
    email TEXT
)""")

# Insert one row
c.execute("""INSERT INTO customers VALUES
            ('John', 'Elder', 'john@codemy.com')""")

# Insert multiple rows
many_customers = [
                 ('Alice', 'Smith', 'alice@example.com'),
                 ('Bob', 'Johnson', 'bob@example.com'),
                 ('Emma', 'Davis', 'emma@example.com'),
            ]
c.executemany("INSERT INTO customers VALUES (?,?,?)", many_customers)

# Commit command
conn.commit()

# Close the connection
conn.close()
```

- **`c.execute()`:** Executes a single SQL statement. Takes a string containing the SQL statement as its argument.
- **`c.executemany()`:** Executes multiple SQL statements with varying parameter values. Takes two arguments: a string containing the SQL statement with placeholders for parameters, and a sequence (e.g., list, tuple, or iterable) containing the parameter values.
- **`conn.commit()`:** Commits the current transaction. This means that all the changes made in the transaction are permanently saved to the database.
- **`conn.close()`:** Closes the database connection. It's important to close the connection after you're done using it to free up resources and ensure proper cleanup.

6.1.4 Data Types

SQLite supports the following data types:

- **NULL**: Used to indicate a missing or unknown value.
- **INTEGER**: Used for storing integer values.
- **REAL**: Used for storing floating-point numbers. It stores approximate numeric values.
- **TEXT**: Used for storing text strings, such as character data. It can store strings of any length.
- **BLOB**: Used for storing binary data, such as images, audio, or other types of files, of any size.

6.1.5 Query a Database

This Python script connects to an SQLite database named ’customer.db’, queries the ’customers’ table, fetches all data, prints it, commits changes, and closes the connection.

```python
import sqlite3

conn = sqlite3.connect('customer.db')
c = conn.cursor()

# Query the databse
c.execute("SELECT * FROM customers")

#c.fetchone()
#c.fetchmany(3)
rows = c.fetchall()

for row in rows:
    print(row)

conn.commit()
conn.close()
```

- **`c.fetchone()`:** Retrieves the next single row from the result set.
- **`c.fetchmany(n)`:** Retrieves the next set of rows, where `n` determines how many rows to fetch.
- **`c.fetchall()`:** Retrieves all remaining rows from the result set and returns them as a list of tuples.

6.1.6 Updating and Deleting Data

This code connects to an SQLite database file named ”customer.db”, creates a cursor object, updates all occurrences of ’John’ to ’Tom’ in the ’first_name’ column, deletes rows where the first name is ’Alice’ and the last name is ’Smith’, commits the changes, and finally closes the connection to the database.

```python
import sqlite3

conn = sqlite3.connect('customer.db')
c = conn.cursor()

# Update 'John' to 'Tom'
c.execute(
    "UPDATE customers SET first_name = 'Tom' WHERE first_name = 'John'"
)

# Delete the rows where first name is 'Alice' and last name is 'Smith'
c.execute("""DELETE FROM customers
             WHERE first_name = 'Alice'
             AND last_name = 'Smith'""")

conn.commit()
conn.close()
```

Note: The command `sqlite.connect()` creates a database if it doesn’t exist.

6.1.7 Transaction Demarcation

Transaction demarcation refers to the delineation or boundary that separates one transaction from another within a database management system (DBMS). A transaction in this context is a sequence of operations (such as inserts, updates, or deletes) that are treated as a single logical unit of work.

6.2 JSON Files

JSON (JavaScript Object Notation) files are a popular data interchange format used to store and exchange data between different platforms and programming languages. They are human-readable and easy to parse, making them a common choice for configuration files, web APIs, and data exchange between client and server. JSON files consist of key-value pairs organized into objects and arrays. The official JSON specification, requires double quotes for string values.

Here's a basic example of a JSON file:

```json
{
  "name": "John Doe",
  "age": 30,
  "city": "New York",
  "is_student": false,
  "hobbies": ["reading", "traveling", "photography"],
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "zip": "12345"
  }
}
```

6.2.1 Serialization and Deserialization

- **json.dump(data, file.json)**
  This function is used to *serialize* Python objects into a JSON formatted file. It takes two arguments: the Python object to be serialized and a file object to which the JSON data will be written.
- **json.load(data, file.json)**
  This function is used to *deserialize* a JSON file or string. It takes a file object or a string containing JSON data as input and returns a Python object (typically a dictionary or a list) representing the JSON data.
- **json.dumps(data)**
  This function is used to *serialize* Python objects into a JSON formatted string. It takes a Python object as input and returns a string containing the JSON data.
- **json.loads(data)**
  This function is used to *deserialize* a JSON string. It takes a string containing JSON data as input and returns a Python object representing the JSON data.

```python
import json

# JSON-formatted string
json_string = '[4, 3, "5.5", true, null, "code", {"one": "uno"}]'

# Deserialize JSON string into Python data structures
data = json.loads(json_string)
print(data) # [4, 3, '5.5', True, None, 'code', {'one': 'uno'}]
```

In this example, `json.loads()` parses the JSON data stored in the `json_string` variable and returns the corresponding Python data structure, which is then stored in the variable `data`.

6.2.2 Conversion

Type	Python	JSON
Strings	`"hello" / 'hello'`	*”hello” - wrapped in double quotes*
Numbers	`123`	*No conversion needed*
Boolean Values	`True` / `False`	`true` / `false`
Sequence Types	Lists `[1, 2, 3]` / Tuples `(1, 2, 3)`	JSON arrays, represented by square brackets
Dictionaries	`{"key": "value"}`	JSON objects, represented by curly braces
None / null	`None`	`null`

Table 1: Conversion of Python datatypes to JSON

6.2.3 Working with the `json` Module

```python
import json

# Write the contents to the JSON file 'fruits.json'
with open('fruits.json', 'w') as file:
    data = []
    banana = {'name': 'Banana', 'calories': 100}
    data.append(banana)
    json.dump(data, file)
```

Initially, a dictionary named `banana` with the keys `name` and `calories` is created. The dictionary is then appended to the list `data`. Subsequently, the list `data` is converted into JSON format using `json.dump()` and saved in the file `'fruits.json'`.

```python
import json

# Read the contents from the JSON file 'fruits.json'
with open('fruits.json', 'r') as file:
    data = json.load(file)
    print(data) # Output: [{'name': 'Banana', 'calories': 100}]
    print(data[0]['name']) # Output: Banana
```

To read from this JSON file, `json.load()` is used. It deserializes the JSON data from `'fruits.json'` and returns a list with the `banana` dictionary representing the JSON object.

6.3 CSV Files

6.3.1 Creating and Reading CSV Files

CSV (Comma-Separated Values) files are plain text files that store tabular data in a structured format. Each line in a CSV file typically represents a row of data, and within each line, individual values are separated by commas or other delimiters, such as semicolons or tabs. CSV files are widely used for storing and exchanging data between different systems because they are simple, lightweight, and easily readable by both humans and computers.

```python
import csv

# Data to be written to the CSV file
data = [
    ['Name', 'Age', 'Major'],
    ['Alice', 25, 'Computer Science'],
    ['Bob', 30, 'Engineering'],
    ['Charlie', 28, 'Mathematics'],
    ['David', 35, 'Physics']
]

# Open the file in write mode
with open('students.csv', 'w', newline='') as csvfile:
    # Create a CSV writer object
    writer = csv.writer(csvfile, delimiter=',')

    # Write the data to the CSV file
    writer.writerows(data)

print("CSV file 'students.csv' has been created successfully.")
```

```python
import csv

with open('students.csv', newline='') as csvfile:
    reader = csv.DictReader(csvfile, delimiter=',')
    print(f"Fieldnames: {reader.fieldnames}\n")
    for row in reader:
        for field, value in row.items():
            print(f"{field}: {value}")
        print()  # Print an empty line between each student's data
```

6.3.2 Excel

Excel files are binary files created by Microsoft Excel or other spreadsheet software. They store data in a tabular format similar to CSV files but offer more features, such as formatting options, formulas, charts, and multiple sheets within a single file.

CSV files and Excel files are related in the sense that you can import CSV files into Excel or export data from Excel to CSV format. This interchangeability makes CSV files a common format for exchanging data between different programs and platforms, including Excel. Excel provides built-in functionality to open and save CSV files, allowing users to work with CSV data directly in Excel or import/export data between Excel and other software applications that support CSV format.

6.3.3 Processing CSV Files

The `reader`, `writer`, `DictReader`, and `DictWriter` are classes provided by the `csv` module for reading and writing CSV files.

- `reader`
  This class is used for reading data from a CSV file. It provides a simple interface for iterating over the lines in the CSV file, where each line is represented as a list of fields.

```
import csv

with open('data.csv', 'r') as file:
    csv_reader = csv.reader(file)
    for row in csv_reader:
        print(row)
```

- `writer`
  This class is used for writing data to a CSV file. It provides methods for writing rows of data to the CSV file.

```
import csv

with open('data.csv', 'w', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerow(['Name', 'Age', 'Country'])
    csv_writer.writerow(['Alice', 30, 'USA'])
    csv_writer.writerow(['Bob', 25, 'Canada'])
```

- `DictReader`
  This class is similar to `reader`, but it returns each row as a dictionary where the keys are the field names and the values are the corresponding values in the row. This is useful when the CSV file has a header row.

```
import csv

with open('data.csv', 'r') as file:
    csv_dict_reader = csv.DictReader(file)
    for row in csv_dict_reader:
        print(row['Name'], row['Age'], row['Country'])
```

- `DictWriter`
  This class is similar to `writer`, but it accepts dictionaries as input for writing rows. Each dictionary represents a row where keys correspond to field names.

```
import csv

with open('data.csv', 'w', newline='') as file:
    fieldnames = ['Name', 'Age', 'Country']
    csv_dict_writer = csv.DictWriter(file, fieldnames=fieldnames)
    csv_dict_writer.writeheader()
    csv_dict_writer.writerow(
    {'Name': 'Alice', 'Age': 30, 'Country': 'USA'})
    csv_dict_writer.writerow(
    {'Name': 'Bob', 'Age': 25, 'Country': 'Canada'})
```

6.4 HTML / XML Documents

HTML (Hypertext Markup Language) and XML (eXtensible Markup Language) are both markup languages used for structuring and presenting information on the internet. While they have similarities, they serve different purposes and have different syntax.

HTML

- Used for creating web pages and web applications.
- Focuses on defining the structure and presentation of content.
- Contains predefined tags for elements like headings, paragraphs, lists, links, etc.
- Supports hypertext links for navigation.
- Often used in conjunction with CSS for styling.
- Example: `<h1>Hello, World!</h1>`

XML

- Used for storing and transporting data.
- Focuses on describing the structure and meaning of data.
- Allows users to define custom tags to represent data.
- Hierarchical structure with nested elements.
- Platform and language agnostic, suitable for data exchange between systems.
- Example:
  `<book><title>The Kite Runner</title><author>Khaled Hosseini</author></book>`

6.4.1 Beautiful soup

Beautiful soup is a Python library used for web scraping, particularly for parsing and extracting data from HTML and XML documents. While it can be used in combination with other libraries to fetch data from servers, it is not the primary library for making HTTP requests.

```python
import requests
from bs4 import BeautifulSoup

# URL of the website you want to scrape
url = 'https://docs.python.org/3/'

# Sending a GET request to the URL
response = requests.get(url)

# Parsing the HTML content using BeautifulSoup
soup = BeautifulSoup(response.text, 'html.parser')

# Extracting the title of the webpage
title = soup.title.string
print("Title of the webpage:", title)
```

6.4.2 Sample XML Document

```xml
<?xml version="1.0" encoding="UTF-8"?>
<library>
    <book id="1">
        <title>Animal Farm</title>
        <author>George Orwell</author>
        <genre>Dystopian Fiction</genre>
        <year>1945</year>
    </book>
    <book id="2">
        <title>7 Habits of Highly Effective People</title>
        <author>Stephen R. Covey</author>
        <genre>Self-help</genre>
        <year>1989</year>
    </book>
    <book id="3">
        <title>Rich Dad Poor Dad</title>
        <author>Robert T. Kiyosaki</author>
        <genre>Personal Finance</genre>
        <year>1997</year>
    </book>
</library>
```

6.4.3 Document Type Definition (DTD)

DTD and XML are two related concepts often used together to define and structure data in a document. DTD is a way to formally describe the structure and content of XML documents. It defines the rules that an XML document must follow. DTD specifies the elements and attributes that are allowed within the document, as well as their relationships and content types.

```
<!DOCTYPE library [
  <!ELEMENT library (book+)>
  <!ELEMENT book (title, author, genre, year)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT genre (#PCDATA)>
  <!ELEMENT year (#PCDATA)>
]>
```

6.4.4 The XML Tree Structure

XML can be visualized and understood as a tree structure. An XML tree structure is a hierarchical representation of XML data, where elements are organized in a parent-child relationship.

- **Root Element**
  - The topmost element in the XML document.
  - Acts as the parent element of all other elements.
  - There can be only one root element in an XML document.
  - Example: `<library>`
- **Elements**
  - Basic building blocks of an XML document.
  - Represented by tags enclosed in angle brackets (< and >).
  - Can contain other elements, text content, or attributes.
  - Example: `<book>`
- **Parent-Child Relationships**
  - Elements can contain other elements, creating a hierarchical structure.
  - The containing element is referred to as the parent, and the contained elements are its children.
  - Children are nested within their parent elements.
  - Example:

    ```
    <book>
      <title>Animal Farm</title>
      <author>George Orwell</author>
    </book>
    ```
- **Attributes:**
  - Provide additional information about elements.
  - Attributes are specified within the start tag of an element.
  - Each attribute consists of a name and a value.
  - Example: `<book id="123">`
- **Text Content**
  - Elements can contain text content.
  - Text content is the data encapsulated within an element.
  - Text content is stored as a child node of the element in the XML tree.
  - Example: `<title>Animal Farm</title>`
- **Sibling Elements**
  - Elements that share the same parent are called siblings.
  - Sibling elements are positioned at the same level within the XML hierarchy.
  - They have the same parent but different names or content.

6.4.5 Parsing XML Files with `xml.etree.ElementTree`

Parsing XML in Python can be done using various libraries, but one of the most commonly used libraries is the built-in `xml.etree.ElementTree`. The `xml.etree.ElementTree` library provides a simple and efficient API for working with XML documents.

- **Parsing XML**
  It provides functions to parse XML documents into an ElementTree object, which represents the hierarchical structure of the XML document. This allows easy traversal and manipulation of XML elements.
- **Creating XML**
  It allows the creation of new XML documents by constructing Element objects and building up the XML tree structure.
- **Navigating XML**
  `ElementTree` provides methods like `find()`, `findall()`, and `iter()` to navigate through XML elements, search for specific elements, and iterate over elements efficiently.
- **Accessing Element Attributes and Text**
  It provides methods to access attributes and text content of XML elements, making it easy to extract data from XML documents.
- **Modifying XML**
  `ElementTree` allows modification of XML documents by adding, removing, or modifying elements and their attributes.
- **Serializing XML**
  It supports serialization of the XML tree back into a string representation or writing it directly to a file.

6.4.6 Searching Data with find() and findall()

The find() and findall() methods are used for searching and retrieving elements from an XML tree structure. Both methods traverse the XML tree starting from the element on which they are called and search through its descendants. They allow for efficient navigation and retrieval of elements based on their tag names. These methods are commonly used when processing XML data to extract specific information or manipulate the XML structure.

```python
import xml.etree.ElementTree as ET

# Sample XML data
xml_data = '''
<people>
    <person>
        <name>John Doe</name>
        <age>30</age>
        <city>New York</city>
    </person>
    <person>
        <name>Jane Smith</name>
        <age>25</age>
        <city>Los Angeles</city>
    </person>
</people>
'''

# Parse the XML data
people = ET.fromstring(xml_data)

# Iterate over each 'person' element
for person in people.findall('person'):
    name = person.find('name').text
    age = int(person.find('age').text)
    city = person.find('city').text
    print(f"Name: {name}, Age: {age}, City: {city}")
```

In this example, xml.etree.ElementTree is used to parse the XML data. The fromstring() method is used to parse the XML string and return the root element. Then, findall() method is used to find all the 'person' elements within the root element, and for each 'person' element, find() method is used to find child elements like 'name', 'age', and 'city'. Finally, their text content is extracted using the text property.

6.4.7 XPath Expressions

XPath (XML Path Language) expressions are used to navigate through elements and attributes in an XML document. XPath provides a way to select nodes in an XML document by using a path-like syntax, similar to filesystem paths. XPath expressions can be used to pinpoint specific elements or attributes within an XML document based on their structure, content, or relationship to other elements.

XPath expressions are widely used in XML processing technologies such as XSLT (Extensible Stylesheet Language Transformations), XQuery, and XML parsers like lxml in Python or XPath libraries in other programming languages.

```python
from lxml import etree

# XML document string
xml_string = """
<universities>
  <university name="Harvard University">
    <location>Cambridge, MA</location>
    <rank>1</rank>
  </university>
  <university name="Stanford University">
    <location>Palo Alto, CA</location>
    <rank>2</rank>
  </university>
  <university name="Massachusetts Institute of Technology (MIT)">
    <location>Cambridge, MA</location>
    <rank>3</rank>
  </university>
</universities>
"""

# Parse the XML string
root = etree.fromstring(xml_string)

# XPath expression to find names of all universities in "Cambridge, MA"
xpath_expr = "/universities/university[location='Cambridge, MA']/@name"

# Apply the XPath expression
selected_universities = root.xpath(xpath_expr)

# Print the selected university names
for university in selected_universities:
    print(university)
```

The XPath expression to select the names of all universities located in ”Cambridge, MA” is: **`/universities/university[location=’Cambridge, MA’]/@name`**.

The expression **`//university/@name`** selects the `name` attribute of all `<university>`-elements.

- `//university` selects all `<university>` elements, regardless of their depth within the hierarchy. `//element` selects all elements at any level of the XML hierarchy.
- `/@name` selects the `name` attribute of each selected <university>element.

6.4.8 Building XML Documents using the `Element` Class and the `SubElement` Function

Creating XML documents can be done using the `Element` class and the `SubElement` function provided by the `xml.etree.ElementTree` module.

- **Element class**
  - The `Element` class represents an XML element.
  - It is used to create individual XML elements with specific tag names.
  - Syntax: `element = Element(tag)`, where `tag` is the name of the XML element.
  - Example: `person_element = Element("person")`.

- **SubElement function**
  - The `SubElement` function is used to create child elements and add them to a parent element.
  - It takes three parameters: the parent element, the tag name of the child element, and optional attributes for the child element.
  - Syntax: `SubElement(parent, tag, attrib={})`.
  - Example: `name_element = SubElement(person_element, "name")`.

```python
import xml.etree.ElementTree as ET

# Create root element
people = ET.Element("people")

# Create first person element
person1 = ET.SubElement(people, "person", id="1")
person1.text = "John Doe"
ET.SubElement(person1, "age").text = "30"
ET.SubElement(person1, "city").text = "New York"

# Create second person element
person2 = ET.SubElement(people, "person", id="2")
person2.text = "Jane Smith"
ET.SubElement(person2, "age").text = "25"
ET.SubElement(person2, "city").text = "Los Angeles"

# Create ElementTree object
tree = ET.ElementTree(people)

# Write XML to file or print it
tree.write("people.xml", encoding="utf-8", xml_declaration=True)
```

6.5 Logging

Logging is a mechanism used to record events that occur during program execution. It allows developers to track the flow of their program, identify errors, and monitor its behavior.The documentation for logging in Python can be found here.

- **Importing the logging module**
  Logging functionality is provided by the logging module, so the first step is usually to import it using `import logging`.
- **Configuring logging**
  Before logging anything, you typically configure the logging system. This includes setting the logging level, specifying the logging format, and determining where log messages should be stored or displayed. The `basicConfig()` function is commonly used to configure logging settings.
- **Creating loggers**
  Loggers are objects used to log messages. Each module or component in your program can have its own logger. You create a logger using `logging.getLogger(name)`, where `name` is typically the name of the module or component.
- **Logging messages**
  After creating loggers, you can utilize various logging methods to record messages at different severity levels:
  - `debug()`: Used for low-level system details or debugging information.
  - `info()`: Provides general informational messages about the system's operation.
  - `warning()`: Logs potential issues that do not necessarily disrupt the functionality
  - `error()`: Logs error messages for significant issues that affect the functionality.
  - `critical()`: Logs critical issues that require immediate attention.
- **Logging levels:** CRITICAL - ERROR - WARNING - INFO - DEBUG - NOTSET

6.5.1 Configuration

The `basicConfig` enables quick setup of logging by specifying basic options like log format, output destination, and minimum logging level. LogRecord attributes can be found here.

```python
import logging

# Configure logging
logging.basicConfig(level=logging.INFO, filename="log.log", filemode="w",
                    format="%(asctime)s - %(levelname)s - %(message)s")

# Log messages at different levels
logging.debug("debug")          # Displayed by default
logging.info("info")            # Displayed if level is set to INFO or lower
logging.warning("warning")      # Always displayed
logging.error("error")          # Always displayed
logging.critical("critical")    # Always displayed
```

6.5.2 Logging Variables or Exceptions

This demonstrates basic logging functionality using the logging module.

```python
import logging

# Setting up basic configuration for logging
logging.basicConfig(level=logging.DEBUG, filename="log.log", filemode="w",
                    format="%(asctime)s - %(levelname)s - %(message)s")

# Logging a variable
var = 2
logging.debug(f"The value of var is {var}")

# Logging an exception
try:
    1 / 0
except ZeroDivisionError as e:
    logging.error("Encountered a ZeroDivisionError", exc_info=True)
    logging.exception("Encountered a ZeroDivisionError")
```

6.5.3 Creating a Custom Logger, Handler and Formatter

This code sets up a custom logging configuration, which includes configuring a logger, defining a file handler to direct log messages to a separate file, specifying a custom log message format, and logging a test message to verify the setup.

```python
import logging

# Configure the basic logging settings
logging.basicConfig(level=logging.DEBUG, filename="log.log", filemode="w",
                    format="%(asctime)s - %(levelname)s - %(message)s")

# Creates a new logger if it doesn't already exist
# Using __name__ as the logger name ensures uniqueness and context relevance
logger = logging.getLogger(__name__)

# Define a file handler for logging to a separate file
handler = logging.FileHandler('tutorial.log')

# Define a custom formatter for the file handler
formatter = logging.Formatter(
    '%(asctime)s -  %(name)s - %(levelname)s - %(message)s')
handler.setFormatter(formatter)

# Add handler to the logger
logger.addHandler(handler)

# Log an informational message
logger.info("Test the custom logger")
```

6.5.4 LogRecord Attributes

The LogRecord has a number of attributes, most of which are derived from the parameters to the constructor. (Note that the names do not always correspond exactly between the LogRecord constructor parameters and the LogRecord attributes.) These attributes can be used to merge data from the record into the format string. The following table lists (in alphabetical order) the attribute names, their meanings, and the corresponding placeholder.

Attribute	**Format**	**Description**
args	Not formatted directly	Tuple of arguments merged into msg to produce message or dictionary values used for the merge (when only one argument and it is a dictionary).
asctime	%(asctime)s	Human-readable time when LogRecord was created (e.g., ’2003-07-08 16:49:45,896’ with millisecond portion after comma).
created	%(created)f	Time when LogRecord was created (as returned by time.time()).
exc_info	Not formatted directly	Exception tuple (à la sys.exc_info) or None if no exception occurred.
filename	%(filename)s	Filename portion of pathname.
funcName	%(funcName)s	Name of function containing the logging call.
levelname	%(levelname)s	Text logging level for the message (’DEBUG’, ’INFO’, ’WARNING’, ’ERROR’, ’CRITICAL’).
levelno	%(levelno)s	Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL).
lineno	%(lineno)d	Source line number where logging call was issued (if available).
message	%(message)s	Logged message, computed as msg % args. Set when Formatter.format() is invoked.
module	%(module)s	Module (name portion of filename).
msecs	%(msecs)d	Millisecond portion of time when LogRecord was created.
msg	Not formatted directly	Format string passed in original logging call, merged with args to produce message.
name	%(name)s	Name of logger used to log the call.
pathname	%(pathname)s	Full pathname of source file where logging call was issued (if available).
process	%(process)d	Process ID (if available).
processName	%(processName)s	Process name (if available).
relativeCreated	%(relativeCreated)d	Time in milliseconds when LogRecord was created, relative to time the logging module was loaded.
stack_info	Not formatted directly	Stack frame information (where available) from bottom of stack in current thread, up to and including stack frame of logging call which resulted in creation of this record.
thread	%(thread)d	Thread ID (if available).
threadName	%(threadName)s	Thread name (if available).
taskName	%(taskName)s	asyncio.Task name (if available).

LogRecord attributes can be found here.

6.6 INI Files

INI (initialization) files are simple text-based configuration files commonly used to store configuration settings for software applications. The name ”INI” comes from the filename extension ”.ini” often used for such files.

INI files consist of sections, each containing key-value pairs that represent configuration settings. The format is straightforward, typically resembling the following:

```
[DEFAULT]
host = localhost # This is a comment.

[mariadb]
name = hello
user = user
password = password

[redis]
port = 6379
db = 0
```

- **Sections:** Sections are enclosed in square brackets ([]) and provide a way to group related configuration settings together. Sections help organize settings and prevent naming conflicts between keys.
- **Keys and Values:** Each key-value pair represents a configuration setting. The key is separated from the value by an equal sign (=). Keys are unique within a section and are used to identify specific configuration options, while values represent the corresponding settings.

6.6.1 Interpolating Values in .ini Files

Interpolating values in .ini files refers to the process of replacing placeholders or variables within the configuration file with actual values. This is commonly done to make configuration files more dynamic and reusable.

For example, consider an .ini file with the following content:

```
[Paths]
base_path = /home/user
data_path = %(base_path)s/data
log_path = %(base_path)s/logs
```

In this file, `%()` syntax is used to define placeholders. When values are interpolated, the placeholders are replaced with the actual values they represent. For instance, if `base_path` is `/home/user`, then after interpolation, `data_path` will become `/home/user/data` and *log_path* will become `/home/user/logs`.

Interpolating values in .ini files allows for more flexible and reusable configuration management, as it enables referencing and reusing values defined elsewhere in the configuration file. This can be particularly useful for managing paths, URLs, and other repetitive configurations.

6.6.2 The configparser

The `configparser` object is part of the standard library module `configparser`. It provides a way to work with configuration files in a consistent, easy-to-use manner. Configuration files are typically used to store settings and parameters for applications.

```python
from configparser import ConfigParser

config = ConfigParser()

config["Server"] = {
    "hostname": "example.com",
    "port": "8080",
    "protocol": "http"
}

config["Database"] = {
    "db_name": "example_db",
    "db_user": "admin",
    "db_password": "strong_password"
}

with open("app_config.ini", "w") as f:
    config.write(f)
```

This creates the following INI File:

```
[Server]
hostname = example.com
port = 8080
protocol = http

[Database]
db_name = example_db
db_user = admin
db_password = strong_password
```

```python
from configparser import ConfigParser

config = ConfigParser()
config.read("app_config.ini")

# Extracting configuration from the sections
server_config = config["Server"]
database_config = config["Database"]

# Displaying configuration values
print("Server Configuration:")
print(f"Hostname: {server_config['hostname']}")
print(f"Port: {server_config['port']}")
print(f"Protocol: {server_config['protocol']}")
```

7 PCPP1 Certification

7.1 Exam Syllabus

The exam syllabus can be accessed here.

Section	Items	Max Score
Section 1: Advanced Object-Oriented Programming	15	42 (35%)
Section 2: Coding Conventions, Best Practices, and Standarization	7	14 (12%)
Section 3: GUI Programming	8	24 (20%)
Section 4: Network Programming	8	22 (18%)
Section 5: File Processing and Communicating with a Program's Environment	7	18 (15%)

Table 3: Exam Syllabus