Python

Cook Book

Everyone can cook delicious recipes with Python

- 100% Practical
- 300+ Code Recipes
- Easy To Follow

Hernando Abella

Introduction

Learn to cook delicious and fun recipes in Python. codes that will help you grow in the programming environment using this wonderful language.

Some of the recipes you will create will be related to: Algorithms, classes, flow control, functions, design patterns, regular expressions, working with databases, and many more things.

Learning these recipes will give you a lot of confidence when you are creating great programs, and you will have more understanding when reading live code.

Abstract Classes

Abstract classes serve as templates for creating concrete classes. They define methods that must be implemented by subclasses, ensuring a consistent interface across different implementations. By defining common behavior and enforcing specific methods, abstract classes promote code reuse and maintainability. They cannot be instantiated directly, highlighting their role as conceptual models rather than concrete entities. Abstract classes are essential in scenarios where multiple classes share common traits but also require specific implementations.

Collection of similar objects

```python
class ANamed:
    name = ""

class Flower(ANamed):
    pass

class City(ANamed):
    pass

class Star(ANamed):
    pass

rose = Flower()
rose.name = "Rose"

rome = City()
rome.name = "Rome"

sirius = Star()
sirius.name = "Sirius"

rows = [rose, rome, sirius]
names = ", ".join([r.name for r in rows])

# names is "Rose, Rome, Sirius"
```

Conformance checking (is, as)

```python
from abc import ABC

class PUID(ABC):
    id = 0

class Named(ABC):
    name = ""

class Flower(Named):
    def __init__(self, name):
        self.name = name

rose = Flower("Rose")
isPUID = isinstance(rose, PUID)

isNamed = isinstance(rose, Named)

print(isPUID) # isPId is False
print(isNamed) # isNamed is True
```

Constructor requirements

```
from abc import *

class List(ABC):
    @abstractmethod
    def __init__(self, item_count):
        self.itemCount = item_count

class SortedList(List):
    def __init__(self, item_count):
        super().__init__(item_count)
        # implementation
        print(item_count)

lst = SortedList(10)
print(lst.itemCount)

# 10
# 10
```

Declaration and initialization

```
from abc import ABC, abstractmethod

class Printable(ABC):
    @abstractmethod
    def print(self, color):
        pass
shape = Printable() # <-error
```

Inheritance of abstract classes

```
from abc import *

class AVehicle(ABC):
    @property
    @abstractmethod
    def max_speed(self):
        pass

class ATruck(AVehicle):
    @property
    @abstractmethod
    def capacity(self):
        pass

class Kamaz5320(ATruck):
    @property
    def max_speed(self):
        return 85

    @property
    def capacity(self):
        return 8000

kamaz = Kamaz5320()
maxSpeed = kamaz.max_speed
# maxSpeed is 85

print(maxSpeed) # 85
```

Methods requirements

```
from abc import *

class Car(ABC):
    @abstractmethod
    def start_engine(self):
        pass

    @abstractmethod
    def stop_engine(self):
        pass

class SportCar(Car):
    def __init__(self):
        self.started = False

    def start_engine(self):
        if self.started:
            return False
        print("start engine")
        self.started = True
        return True

    def stop_engine(self):
        print("stop engine")
        self.started = False

car = SportCar()
car.start_engine()
# start engine
```

Multiple inheritance

```
from abc import *

class PId(ABC):
    @property
    @abstractmethod
    def id(self):
        pass

class Priced(ABC):
    @property
    @abstractmethod
    def price(self):
        pass

class Goods(PId, Priced):
    def __init__(self, p_id, p_price):
        self._id = p_id
        self._price = p_price

    @property
    def id(self):
        return self._id

    @property
    def price(self):
        return self._price

def show_id_and_price(info):
    print(f"id = {info.id}, price = {info.price}")

bread = Goods(1, 5)
show_id_and_price(bread)
# printed: id = 1, price = 5
```

Properties requirements

```
from abc import *

class ACar(ABC):
    @property
    @abstractmethod
    def engine_volume(self):
        pass

    @engine_volume.setter
    @abstractmethod
    def engine_volume(self, val):
        pass

class Airwave(ACar):
    def __init__(self):
        self._engineVolume = 1500

    @property
    def engine_volume(self):
        return self._engineVolume

airwave = Airwave()
print(airwave.engine_volume) # 1500
```

Subscript requirements

```
from abc import *

class AIterable(ABC):
    @abstractmethod
    def __getitem__(self, i):
        pass

class PowerOfTwo(AIterable):
    pass

    def __getitem__(self, i):
        return pow(2, i)

power = PowerOfTwo()
p8 = power[8]
# p8 is 256

p16 = power[16]
#p16 is 65536

print(p8)
print(p16)
```

Algorithms

Algorithms are step-by-step procedures or formulas for solving problems and performing tasks. They are the backbone of computer science, enabling efficient data processing and decision-making. An algorithm takes input, processes it through a series of well-defined steps, and produces an output. They can range from simple arithmetic operations to complex data structures and sorting techniques. Effective algorithms are characterized by their efficiency, scalability, and clarity. Understanding and designing algorithms are crucial for optimizing performance and resource utilization in software development.

Sorting algorithms:

Bubble Sort

```
def bubble_sort(arr):
    items = arr[:]
    for i in range(len(items)):
        for j in range(i + 1, len(items)):
            if items[j] < items[i]:
                items[j], items[i] = items[i], items[j]
    return items

items = [4, 1, 5, 3, 2]
sort_items = bubble_sort(items)
print("Sorted items:", sort_items)
# Sorted items: [1, 2, 3, 4, 5]
```

Counting Sort

```python
def counting_sort(arr):
    maximum = max(arr)
    counts = [0] * (maximum + 1)
    items = [0] * len(arr)

    for x in arr:
        counts[x] += 1

    total = 0
    for i in range(len(counts)):
        old_count = counts[i]
        counts[i] = total
        total += old_count

    for x in arr:
        items[counts[x]] = x
        counts[x] += 1

    return items

items = [4, 1, 5, 3, 2]
sort_items = counting_sort(items)
print("Sorted items:", sort_items)
# Sorted items: [1, 2, 3, 4, 5]
```

Merge Sort

```
def merge_sort(items):
    if len(items) <= 1:
        return items

    middle = len(items) // 2
    left = items[:middle]
    right = items[middle:]

    def merge(left, right):
        result = []
        left_index = 0
        right_index = 0

        while left_index < len(left) and right_index < len(right):
            if left[left_index] < right[right_index]:
                result.append(left[left_index])
                left_index += 1
            else:
                result.append(right[right_index])
                right_index += 1

        result.extend(left[left_index:])
        result.extend(right[right_index:])
        return result

    return merge(merge_sort(left), merge_sort(right))

items = [4, 1, 5, 3, 2]
sort_items = merge_sort(items)
print("Sorted items:", sort_items)
# Sorted items: [1, 2, 3, 4, 5]
```

Quick Sort

```python
def quick_sort(items):
    def do_sort(items, fst, lst):
        if fst >= lst:
            return
        i = fst
        j = lst
        x = items[(fst + lst) // 2]

        while i <= j:
            while items[i] < x:
                i += 1
            while items[j] > x:
                j -= 1
            if i <= j:
                items[i], items[j] = items[j], items[i]
                i += 1
                j -= 1
        do_sort(items, fst, j)
        do_sort(items, i, lst)

    sorted_items = items[:]
    do_sort(sorted_items, 0, len(sorted_items) - 1)
    return sorted_items

items = [4, 1, 5, 3, 2]
sort_items = quick_sort(items)
print("Sorted items:", sort_items)
# Sorted items: [1, 2, 3, 4, 5]
```

Radix Sort

```python
def list_to_buckets(items, c_base, i):
    buckets = [[] for _ in range(c_base)]

    p_base = c_base ** i
    for x in items:
        digit = (x // p_base) % c_base
        buckets[digit].append(x)
    return buckets

def buckets_to_list(buckets):
    result = []
    for bucket in buckets:
        result.extend(bucket)
    return result

def radix_sort(arr, c_base=10):
    max_val = max(arr)
    i = 0
    while c_base ** i <= max_val:
        arr = buckets_to_list(list_to_buckets(arr, c_base, i))
        i += 1
    return arr

items = [4, 1, 5, 3, 2]
sort_items = radix_sort(items)
print("Sorted items:", sort_items)
# Sorted items: [1, 2, 3, 4, 5]
```

Searching algorithms:

Binary Search

```
def binary_search(arr, x):
    i = -1
    j = len(arr)
    while i + 1 != j:
        m = (i + j) // 2
        if x == arr[m]:
            return m
        if x < arr[m]:
            j = m
        else:
            i = m
    return None

items = [2, 3, 5, 7, 11, 13, 17]

print(binary_search(items, 1))
# Will print None
print(binary_search(items, 7))
# Will print 3
print(binary_search(items, 19))
# Will print None
```

Fast Linear Search

```python
def fast_linear_search(arr, x):
    i = 0
    count = len(arr)
    arr.append(x)
    while True:
        if arr[i] == x:
            arr.pop()  # Remove the last element
            return i if i < count else None
        i += 1

items = [2, 3, 5, 7, 11, 13, 17]

print(fast_linear_search(items, 1))
# Will print None
print(fast_linear_search(items, 7))
# Will print 3
print(fast_linear_search(items, 19))
# Will print None
```

Interpolation Search

```
def interpolation_search(arr, x):
    low = 0
    high = len(arr) - 1

    while low <= high and x >= arr[low] and x <= arr[high]:
        mid = low + ((x - arr[low]) * (high - low)) // (arr[high] -
arr[low])

        if arr[mid] < x:
            low = mid + 1
        elif arr[mid] > x:
            high = mid - 1
        else:
            return mid

    if arr[low] == x:
        return low
    if arr[high] == x:
        return high
    return None

items = [2, 3, 5, 7, 11, 13, 17]

print(interpolation_search(items, 1))
# Will print None
print(interpolation_search(items, 7))
# Will print 3
print(interpolation_search(items, 19))
# Will print None
```

Linear Search

```python
def linear_search(arr, x):
    i = 0
    count = len(arr)
    while i < count:
        if arr[i] == x:
            return i
        i += 1
    return None

items = [2, 3, 5, 7, 11, 13, 17]

print(linear_search(items, 1))  # Will print None
print(linear_search(items, 7))  # Will print 3
print(linear_search(items, 19))  # Will print None
```

Changes in new versions

In software development, new versions of a program or system often bring various changes that can include bug fixes, performance improvements, and new features. These updates are crucial for maintaining security, improving user experience, and staying competitive.

Alias type syntax

```python
# *** in version 3.10: ***
from typing import TypeAlias

Index: TypeAlias = int

# *** before: ***
Width = int
```

Comparison operators

```
# *** before: ***
b1 = 1 < "A"
# b1 is True

b2 = 1 == "A"
# b2 is False

# *** in version 3: ***
b1 = 1 < "A" # <- TypeError

b2 = 1 == "A"
# b2 is False
```

Context Variables

```
# *** in version 3.7 ***
import contextvars
number = contextvars.ContextVar("number", default="-1")
contexts = list()

def print_number():
    print(f"number: {number.get()}")

print_number()
# number: -1

# Creating contexts and setting the number
for n in [1, 2, 3]:
    ctx = contextvars.copy_context()
    ctx.run(number.set, n)
    contexts.append(ctx)

# Running print_number () function in each context
for ctx in reversed(contexts):
    ctx.run(print_number)
```

Context variable objects in Python is an interesting type of variable which returns the value of variable according to the context. It may have multiple values according to context in single thread or execution. The ContextVar class present in contextvars module, which is used to declare and work with context variables in python.

Note: This is supported in python version >= 3.7.

Data classes

```python
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    age: int
    job_title: str
    salary: float

    def give_raise(self, amount: float):
        self.salary += amount
        return self.salary

# Create an instance of the Employee class
employee1 = Employee(name="John Doe", age=30,
job_title="Software Engineer", salary=70000.0)

# Print employee details
print(employee1)

# Give the employee a raise
employee1.give_raise(5000.0)
print(f"New salary after raise: {employee1.salary}")
# New salary after raise: 75000.0
```

Dictionary Merge

```
# Define dictionaries
d1 = {1: "one", 2: "two"}
d2 = {3: "three", 4: "four"}
d3 = {5: "five"}

# Merge d1 and d2 using dictionary unpacking
dAll = {**d1, **d2}
print(dAll)
# {1: 'one', 2: 'two', 3: 'three', 4: 'four'}

# Update dAll with d3
dAll.update(d3)
print(dAll)
# {1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five'}
```

Exceptions handling

```
# before version 3

try:
    # Code that may raise an exception
    result = 10 / 0
except ZeroDivisionError:
    # Handling the specific exception
    print("Error: Division by zero!")

# after version 3

try:
    # Code that may raise an exception
    result = 10 / 0
except ZeroDivisionError as e:
    # Handling the specific exception and accessing
exception object
    print(f"Error: {e}")
```

Extended Iterable Unpacking

```python
# Example of extended iterable unpacking

# Unpacking a tuple
tuple_values = (1, 2, 3, 4, 5)
a, *b, c = tuple_values

print("a:", a)  # Output: 1
print("b:", b)  # Output: [2, 3, 4]
print("c:", c)  # Output: 5

# Unpacking a list with excess items
list_values = [1, 2, 3, 4, 5, 6, 7]
first, *middle, last = list_values

print("first:", first)    # Output: 1
print("middle:", middle)  # Output: [2, 3, 4, 5, 6]
print("last:", last)      # Output: 7

# Using extended iterable unpacking with default values
values = [1, 2]
x, y, *z = values

print("x:", x)  # Output: 1
print("y:", y)  # Output: 2
print("z:", z)  # Output: []

# Using extended iterable unpacking with an empty iterable
empty_values = []
a, *b = empty_values

print("a:", a)  # Output: None
print("b:", b)  # Output: []
```

Features of f-strings

```python
# Example before version 3

name = "Alice"
age = 30

# Using format()
formatted_string = "Name: {}, Age: {}".format(name, age)
print(formatted_string)
# Output: Name: Alice, Age: 30

# Example after version 3.12

name = "Alice"
age = 30

# Using f-strings
formatted_string = f"Name: {name}, Age: {age}"
print(formatted_string)
# Output: Name: Alice, Age: 30
```

Guaranteed dictionary order

```
# Example before version 3.5

# Define a dictionary
unordered_dict = {'b': 2, 'a': 1, 'c': 3}

# Iterate over the dictionary
for key, value in unordered_dict.items():
    print(key, value)

# Output order may vary:
# a 1
# b 2
# c 3

# Example after version 3.7

# Define a dictionary
ordered_dict = {'b': 2, 'a': 1, 'c': 3}

# Iterate over the dictionary
for key, value in ordered_dict.items():
    print(key, value)

# Output order is guaranteed to be insertion order:
# b 2
# a 1
# c 3
```

IANA time zone support

```
from datetime import datetime
import zoneinfo

# Create a timezone-aware datetime object for New York
ny_timezone = zoneinfo.ZoneInfo("America/New_York")
ny_time = datetime.now(ny_timezone)

# Create a timezone-aware datetime object for London
london_timezone = zoneinfo.ZoneInfo("Europe/London")
london_time = datetime.now(london_timezone)

# Display the timezone-aware datetimes
print("Current time in New York:", ny_time.strftime('%Y-%m-%d %H:%M:%S %Z%z'))
print("Current time in London:", london_time.strftime('%Y-%m-%d %H:%M:%S %Z%z'))
```

The **zoneinfo** module provides a concrete time zone implementation to support the IANA time zone database as originally specified in PEP 615. By default, zoneinfo uses the system’s time zone data if available; if no system time zone data is available, the library will fall back to using the first-party tzdata package available on PyPI.

Integer division

```
# *** before ***
i1 = 1 / 2
# i is 0 (type 'int')

i2 = 1 // 2
# i2 is 0 (type 'int')

# *** in version 3: ***
i1 = 1 / 2
# i1 is 0.5 (type 'float')

i2 = 1 // 2
# i2 is 0 (type 'int')

print("i1 is", i1)
print("i1 type is", type(i1))
print("i2 is", i2)
print("i2 type is", type(i2))
```

Methods of dictionaries

```
# *** before: ***
dic = {2: "two", 1: "one"}
keys = dic.keys()
keys.sort()
# keys is list
values = dic.values()
values.sort()
# values is list

# *** in version 3: ***
dic = {1: "one", 2: "two"}
keys = dic.keys()
# keys.sort() # <-Error
# keys is dict_keys
values = list(dic.values())
values.sort()
# values is list

print("keys is", keys)
print("keys type is", type(keys))
print("values is", values)
print("values type is", type(values))
```

New Type Union Operator

```python
# *** in version 3.10 ***
def sqrt(number: int | float) -> float:
    return number ** 0.5

sqrt9 = sqrt(9)
print(f"{sqrt9 = }")
sqrt16 = sqrt(16.0)
print(f"{sqrt16 = }")
```

New string methods

```
# *** in version 3.9 ***
dataString = "Substring removing"

print(dataString
      .removesuffix(' removing'))
# prints "Substring"

print(dataString
      .removeprefix('Sub'))
# prints "string removing"
```

Octal literals

```
# *** before: ***
octal = 042
# octal is 34

# *** in version 3: ***
octal = 0o42
# octal is 34

print(octal)
```

Parenthesized context managers

```
# *** in version 3.10: ***
with (open("file.out", "rb") as rf,
     open("file_copy.out", "wb") as wf):
    pass

# *** before: ***
with open("file.out", "rb") as rf:
    with open("file_copy.out", "wb") as wf:
        pass
```

Simplified asynchronous call

```
# *** in version 3.10: ***
import asyncio


async def greeting():
    print("Hello!")

asyncio.run(greeting())
```

Throw an exception

```
# *** before: ***
raise IOError, "file error"

# *** in version 3: ***
raise IOError("file error")
```

Type Hinting Generics

```python
# *** before: ***
def greet_all(names: list[str]):
    for name in names:
        print("Hello", name)

data = ["Alex", "Anna", 2]
greet_all(data)
```

Unicode strings

```
# Example before version 3 (Python 2)

# Defining a Unicode string
unicode_str = u"Hello, \u2603"  # The Unicode character \u2603 is a snowman

# Printing the Unicode string
print(unicode_str)  # Output: Hello, ☃

# Encoding the Unicode string to bytes
encoded_str = unicode_str.encode('utf-8')
print(encoded_str)  # Output: b'Hello, \xe2\x98\x83'

# Example after version 3 (Python 3)

# Defining a Unicode string
unicode_str = "Hello, \u2603"  # The Unicode character \u2603 is a snowman

# Printing the Unicode string
print(unicode_str)  # Output: Hello, ☃

# Encoding the Unicode string to bytes
encoded_str = unicode_str.encode('utf-8')
print(encoded_str)  # Output: b'Hello, \xe2\x98\x83'

# Decoding bytes back to a Unicode string
decoded_str = encoded_str.decode('utf-8')
print(decoded_str)  # Output: Hello, ☃
```

Variables for the ‘for’ loop

```
# *** before: ***
i = 1
[i for i in range(5)]
print i
# i is 4

# *** in version 3: ***

i = 1
[i for i in range(5)]
print(i)
# i is 1
```

Walrus Operator :=

```python
import re
data = "Pi is equal to 3.14"
pNumber = r'\d+\.\d+'
pWord = r'\w{3,15}'

# *** in version 3.8 ***
if m := re.search(pNumber, data):
    number = float(m.group())
    print(number)
elif m := re.search(pWord, data):
    word = m.group()
print(word)

# *** before: ***
m = re.search(pNumber, data)
if m:
    number = float(m.group())
    print(number)
else:
    m = re.search(pWord, data)
if m:
    word = m.group()
print(word)

numbers = [1, 3, 5, 7]

# *** in version 3.8: ***
if (n := len(numbers)) > 3:
    print(f"len is {n} elements, expected <= 3")

# *** before: ***
n = len(numbers)
if n > 3:
    print(f"len is {n} elements, expected <= 3")
```

Walrus-operator is another name for assignment expressions. According to the official documentation, it is a way to assign to variables within an expression using the notation NAME := expr.

f-strings support

```python
from datetime import datetime

number = 42
pi = 3.1415
text = "answer"
now = datetime.now()

# *** in version 3.8 ***
print('in version 3.8:')
print(f'{number=}')
print(f'{pi=}')
print(f'{text=}')
print(f'{now=}')

print()
# *** before: ***
print('before:')
print(f'number={number}')
print(f'pi={pi}')
print(f'text={text}')
print(f'now={now}')
```

map and filter functions

```
# *** before: ***
n1 = [1, 2, 3]
n2 = map(lambda x: x * x, n1)
# n2 is lsit

n3 = filter(lambda x: x * x, n1)
# n3 is list

# *** in version 3 ***
n1 = [1, 2, 3]
n2 = map(lambda x: x * x, n1)
#n2 is map

n3 = filter(lambda x: x % 2 == 1, n1)
# n3 is filter

n4 = list(n2)
# n4 is list

print("n2 is", n2)
print("n2 type is", type(n2))
print("n3 is", n3)
print("n3 type is", type(n3))
print("n4 is", n4)
```

match statements

```
def http_status_code_message(status_code):
    if status_code == 200:
        return "OK"
    elif status_code == 404:
        return "Not Found"
    elif status_code == 500:
        return "Internal Server Error"
    else:
        return "Unknown Status Code"

print(http_status_code_message(200))  # OK
print(http_status_code_message(404))  # Not Found
print(http_status_code_message(123))  # Unknown Status
Code


def http_status_code_message(status_code):
    match status_code:
        case 200:
            return "OK"
        case 404:
            return "Not Found"
        case 500:
            return "Internal Server Error"
        case _:
            return "Unknown Status Code"

print(http_status_code_message(200))  # OK
print(http_status_code_message(404))  # Not Found
print(http_status_code_message(123))  # Unknown Status
Code
```

print function

```
# Python 2 example
print "Hello, World!"  # Hello, World!
print "The answer is", 42  # The answer is 42

# Using a trailing comma to avoid a newline at the end
print "Hello,",
print "World!"  # Hello, World!

# Python 3 example
print("Hello, World!")  # Hello, World!
print("The answer is", 42)  # The answer is 42

# To avoid a newline at the end, use the end parameter
print("Hello,", end=" ")
print("World!")  # Hello, World!
```

range function

```
# Python 2 example using range
numbers = range(1, 10)
print numbers  # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Python 2 example using xrange
numbers = xrange(1, 10)
print numbers  # xrange(1, 10)
print list(numbers)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Python 3 example using range
numbers = range(1, 10)
print(numbers)  # range(1, 10)
print(list(numbers))  # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Classes

In object-oriented programming, classes are fundamental building blocks that define the blueprint for objects. A class encapsulates data for the object and methods to manipulate that data, promoting modularity and code reuse.

Check for reference equality

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

# Create two instances of MyClass
obj1 = MyClass(10)
obj2 = MyClass(10)
obj3 = obj1

# Check for reference equality using id()
print(id(obj1) == id(obj2))
# False, different objects in memory
print(id(obj1) == id(obj3))
# True, same object in memory
```

Constructors:

Call of the own constructor

```
class Person:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    @classmethod
    def from_full_name(cls, full_name, age):
        first_name, last_name = full_name.split()
        # Call the main constructor with first name and last name extracted from full name
        return cls(first_name, last_name, age)

    def display_person(self):
        print(f'Name: {self.first_name} {self.last_name}, Age: {self.age}')

# Create an instance using the main constructor
person1 = Person("John", "Doe", 30)
person1.display_person()
# Output: Name: John Doe, Age: 30

# Create an instance using the alternative constructor
person2 = Person.from_full_name("Jane Smith", 25)
person2.display_person()
# Output: Name: Jane Smith, Age: 25
```

Call of the parent constructor

```
class Person:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def display_person_info(self):
        print(f'Name: {self.first_name} {self.last_name}, Age: {self.age}')

class Employee(Person):
    def __init__(self, first_name, last_name, age, employee_id, position):
        # Call the parent constructor to initialize first_name, last_name, and age
        super().__init__(first_name, last_name, age)
        self.employee_id = employee_id
        self.position = position

    def display_employee_info(self):
        # Call the parent class method to display basic info
        super().display_person_info()
        print(f'Employee ID: {self.employee_id}, Position: {self.position}')

# Create an instance of Person
person = Person("John", "Doe", 45)
person.display_person_info()  # Output: Name: John Doe, Age: 45

# Create an instance of Employee
employee = Employee("Jane", "Smith", 30, "E123", "Software Engineer")
employee.display_employee_info()
# Output:
```

Name: Jane Smith, Age: 30
Employee ID: E123, Position: Software Engineer

Default constructor

```
class Book:
    def __init__(self, title="Unknown Title", author="Unknown Author", year=0):
        self.title = title
        self.author = author
        self.year = year

    def display_info(self):
        print(f'Title: {self.title}, Author: {self.author}, Year: {self.year}')

# Create an instance using the default constructor
default_book = Book()
default_book.display_info()
# Output: Title: Unknown Title, Author: Unknown Author, Year: 0

# Create an instance with custom values
custom_book = Book("1984", "George Orwell", 1949)
custom_book.display_info()
# Output: Title: 1984, Author: George Orwell, Year: 1949
```

Optional parameter values

```
class Car:
    def __init__(self, make="Unknown Make", model="Unknown Model", year=0, color="Unknown Color"):
        self.make = make
        self.model = model
        self.year = year
        self.color = color

    def display_info(self):
        print(f'Make: {self.make}, Model: {self.model}, Year: {self.year}, Color: {self.color}')

# Create an instance using the default constructor (all default values)
default_car = Car()
default_car.display_info()  # Output: Make: Unknown Make, Model: Unknown Model, Year: 0, Color: Unknown Color

# Create an instance with some custom values
custom_car1 = Car(make="Toyota", model="Corolla")
custom_car1.display_info()
# Output: Make: Toyota, Model: Corolla, Year: 0, Color: Unknown Color

# Create an instance with all custom values
custom_car2 = Car(make="Honda", model="Civic", year=2022, color="Red")
custom_car2.display_info()
# Output: Make: Honda, Model: Civic, Year: 2022, Color: Red
```

Replacement of the parent constructor

```
class Person:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def display_person_info(self):
        print(f'Name: {self.first_name} {self.last_name}, Age: {self.age}')

class Employee(Person):
    def __init__(self, first_name, last_name, age, employee_id, position):
        # Call the parent constructor to initialize first_name, last_name, and age
        super().__init__(first_name, last_name, age)
        # Initialize the additional attributes
        self.employee_id = employee_id
        self.position = position

    def display_employee_info(self):
        # Call the parent class method to display basic info
        super().display_person_info()
        print(f'Employee ID: {self.employee_id}, Position: {self.position}')

# Create an instance of Person
person = Person("John", "Doe", 45)
person.display_person_info()
# Output: Name: John Doe, Age: 45

# Create an instance of Employee
```

```python
employee = Employee("Jane", "Smith", 30, "E123",
"Software Engineer")
employee.display_employee_info()
# Output:
# Name: Jane Smith, Age: 30
# Employee ID: E123, Position: Software Engineer
```

With paramenters

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

# Creating an instance of Rectangle with specific dimensions
rectangle1 = Rectangle(5, 3)
print("Area of rectangle1:", rectangle1.area())
# Output: Area of rectangle1: 15

# Creating another instance of Rectangle with different dimensions
rectangle2 = Rectangle(7, 4)
print("Area of rectangle2:", rectangle2.area())
# Output: Area of rectangle2: 28
```

Without any paramenters

```
class MyClass:
    def __init__(self):
        print("This is the default constructor.")

    def display(self):
        print("Inside MyClass.")

# Creating an instance of MyClass
obj = MyClass()
obj.display()
```

Create a copy of the object

```python
import copy

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

    def display_info(self):
        print(f'Name: {self.name}, Age: {self.age}')

# Create an instance of Person
person1 = Person("Alice", 30)
person1.display_info()
# Output: Name: Alice, Age: 30

# Create a shallow copy of person1
person2 = copy.copy(person1)
person2.display_info()
# Output: Name: Alice, Age: 30

# Modify the copy
person2.name = "Bob"
person2.display_info()
# Output: Name: Bob, Age: 30
person1.display_info()
# Output: Name: Alice, Age: 30

# Create a deep copy of person1
person3 = copy.deepcopy(person1)
person3.display_info()
# Output: Name: Alice, Age: 30
```

Definition and initialization

```python
# Definition
class SomeClass:
    pass

# Initialization
someClass = SomeClass()
```

Descriptors

```
class AgeDescriptor:
    def __init__(self):
        self._age = None

    def __get__(self, instance, owner):
        print("Getting age")
        return self._age

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise ValueError("Age must be an integer")
        if value < 0:
            raise ValueError("Age cannot be negative")
        print("Setting age")
        self._age = value

    def __delete__(self, instance):
        print("Deleting age")
        self._age = None

class Person:
    age = AgeDescriptor()

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

    def display_info(self):
        print(f'Name: {self.name}, Age: {self.age}')

# Create an instance of Person
person = Person("Alice", 30)
person.display_info()
# Output: Name: Alice, Age: 30

# Get the age
```

```python
print(person.age)
# Output: Getting age, 30

# Set a new age
person.age = 35
# Output: Setting age

# Get the updated age
print(person.age)
# Output: Getting age, 35

# Delete the age
del person.age  # Output: Deleting age

# Try to get the deleted age
print(person.age)
# Output: Getting age, None
```

Descriptors is an object attribute with “binding behavior”, one whose attribute access has been overridden by methods in the descriptor protocol. Those methods are defined for an object; it is said to be a descriptor.

Destructor

```
class FileManager:
    def __init__(self, file_name, mode):
        self.file_name = file_name
        self.mode = mode
        self.file = open(file_name, mode)
        print(f'File {self.file_name} opened in {self.mode} mode.')

    def write_data(self, data):
        if self.file and not self.file.closed:
            self.file.write(data)
            print(f'Written data: {data}')

    def __del__(self):
        if self.file and not self.file.closed:
            self.file.close()
            print(f'File {self.file_name} closed.')

# Using the FileManager class
file_manager = FileManager('example.txt', 'w')
file_manager.write_data('Hello, world!')

# Deleting the file_manager object explicitly
del file_manager

# Output:
# File example.txt opened in w mode.
# Written data: Hello, world!
# File example.txt closed.
```

Events

```
class Event:
    def __init__(self):
        self.handlers = []

    def add_handler(self, handler):
        self.handlers.append(handler)

    def remove_handler(self, handler):
        self.handlers.remove(handler)

    def fire(self, *args, **kwargs):
        for handler in self.handlers:
            handler(*args, **kwargs)

class TemperatureSensor:
    def __init__(self):
        self.temperature_changed = Event()
        self._temperature = 0

    @property
    def temperature(self):
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        if value != self._temperature:
            self._temperature = value
            self.temperature_changed.fire(value)

class Display:
    def show_temperature(self, temperature):
        print(f'Temperature changed to {temperature}
degrees.')

# Create a TemperatureSensor instance
sensor = TemperatureSensor()
```

```
# Create a Display instance
display = Display()

# Add the display's show_temperature method as a handler for the temperature_changed event
sensor.temperature_changed.add_handler(display.show_temperature)

# Change the temperature, which triggers the event
sensor.temperature = 25

# Output:
# Temperature changed to 25 degrees.
```

Fields

```python
class Car:
    def __init__(self, make, model, year):
        self.make = make  # instance field
        self.model = model  # instance field
        self.year = year  # instance field

    def display_info(self):
        print(f'Car: {self.year} {self.make} {self.model}')

# Create an instance of Car
my_car = Car('Toyota', 'Corolla', 2021)
my_car.display_info()
# Output: Car: 2021 Toyota Corolla
```

Inheritance:

Abstract classes

```
from abc import ABC, abstractmethod
import math

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

    def description(self):
        return "This is a shape."

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def description(self):
        return f"This is a rectangle with length {self.length}
and width {self.width}."

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * (self.radius ** 2)

    def description(self):
        return f"This is a circle with radius {self.radius}."

# Instances of Rectangle and Circle
rectangle = Rectangle(5, 3)
circle = Circle(4)
```

```python
# Displaying information and calculating area
print(rectangle.description())
# Output: This is a rectangle with length 5 and width 3.
print("Area:", rectangle.area())
# Output: Area: 15

print(circle.description())
# Output: This is a circle with radius 4.
print("Area:", circle.area())
# Output: Area: 50.26548245743669
```

Base class

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

    def make_sound(self):
        raise NotImplementedError("Subclasses must implement this method")

    def describe(self):
        return f"{self.name} is a {self.species}"

# Define a derived class
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name, "Dog")
        self.breed = breed

    def make_sound(self):
        return "Bark"

    def describe(self):
        return f"{self.name} is a {self.breed} dog"

# Define another derived class
class Cat(Animal):
    def __init__(self, name, breed):
        super().__init__(name, "Cat")
        self.breed = breed

    def make_sound(self):
        return "Meow"

    def describe(self):
        return f"{self.name} is a {self.breed} cat"

# Create instances of Dog and Cat
```

```
dog = Dog("Buddy", "Golden Retriever")
cat = Cat("Whiskers", "Siamese")

# Use methods from the base class and overridden methods
print(dog.describe())  # Output: Buddy is a Golden Retriever
dog
print(dog.make_sound())  # Output: Bark

print(cat.describe())  # Output: Whiskers is a Siamese cat
print(cat.make_sound())  # Output: Meow
```

Compability check (is)

```
# Define the base class
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        raise NotImplementedError("Subclasses must implement this method")

# Define a derived class
class Dog(Animal):
    def make_sound(self):
        return "Bark"

# Define another derived class
class Cat(Animal):
    def make_sound(self):
        return "Meow"

# Define a function to check compatibility using isinstance
def check_instance(obj, cls):
    if isinstance(obj, cls):
        print(f"{obj.name} is an instance of {cls.__name__}.")
    else:
        print(f"{obj.name} is NOT an instance of {cls.__name__}.")

# Define a function to check subclass compatibility using issubclass
def check_subclass(sub, parent):
    if issubclass(sub, parent):
        print(f"{sub.__name__} is a subclass of {parent.__name__}.")
    else:
```

```
    print(f"{sub.__name__} is NOT a subclass of {parent.__name__}.")

# Create instances of Dog and Cat
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Check instance compatibility
check_instance(dog, Animal)
# Output: Buddy is an instance of Animal.
check_instance(cat, Animal)
# Output: Whiskers is an instance of Animal.
check_instance(dog, Dog)
# Output: Buddy is an instance of Dog.
check_instance(cat, Dog)
# Output: Whiskers is NOT an instance of Dog.

# Check subclass compatibility
check_subclass(Dog, Animal)
# Output: Dog is a subclass of Animal.
check_subclass(Cat, Animal)
# Output: Cat is a subclass of Animal.
check_subclass(Dog, Cat)
# Output: Dog is NOT a subclass of Cat.
```

Interface inheritance

```python
from abc import ABC, abstractmethod

# Define the abstract base class
class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass

    @abstractmethod
    def stop_engine(self):
        pass

    @abstractmethod
    def drive(self):
        pass

# Define a concrete class that inherits from Vehicle
class Car(Vehicle):
    def start_engine(self):
        return "Car engine started."

    def stop_engine(self):
        return "Car engine stopped."

    def drive(self):
        return "Car is driving."

# Define another concrete class that inherits from Vehicle
class Bike(Vehicle):
    def start_engine(self):
        return "Bike engine started."

    def stop_engine(self):
        return "Bike engine stopped."

    def drive(self):
        return "Bike is driving."
```

```python
# Create instances of Car and Bike
car = Car()
bike = Bike()

# Use the methods defined in the interface
print(car.start_engine())
# Output: Car engine started.
print(car.drive())
# Output: Car is driving.
print(car.stop_engine())
# Output: Car engine stopped.

print(bike.start_engine())
# Output: Bike engine started.
print(bike.drive())
# Output: Bike is driving.
print(bike.stop_engine())
# Output: Bike engine stopped.
```

Method override

```python
# Define the base class
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        return "Some generic sound"

    def describe(self):
        return f"This is {self.name}."

# Define a subclass that overrides the make_sound method
class Dog(Animal):
    def make_sound(self):
        return "Bark"

# Define another subclass that overrides the make_sound
method
class Cat(Animal):
    def make_sound(self):
        return "Meow"

# Create instances of Dog and Cat
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Use the overridden methods
print(dog.describe())
# Output: This is Buddy.
print(dog.make_sound())
# Output: Bark

print(cat.describe())
# Output: This is Whiskers.
print(cat.make_sound())
# Output: Meow
```

Private class members

```
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    def _display_info(self):
        return f"Name: {self._name}, Age: {self._age}"

    def get_info(self):
        return self._display_info()

# Creating an instance of Person
person = Person("Alice", 30)

# Accessing private attributes (not enforced)
print(person._name)
# Output: Alice
print(person._age)
# Output: 30

# Accessing private method (not enforced)
print(person._display_info())
# Output: Name: Alice, Age: 30

# Accessing method to retrieve information (recommended
way)
print(person.get_info())
# Output: Name: Alice, Age: 30
```

Property override

```python
import math

# Define the base class
class Shape:
    @property
    def area(self):
        return 0  # Default implementation for base class

# Define a subclass that overrides the area property
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def area(self):
        return self.width * self.height

# Define another subclass that overrides the area property
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    @property
    def area(self):
        return math.pi * (self.radius ** 2)

# Create instances of Rectangle and Circle
rectangle = Rectangle(5, 3)
circle = Circle(4)

# Access the overridden properties
print("Area of rectangle:", rectangle.area)
# Output: Area of rectangle: 15
print("Area of circle:", circle.area)
# Output: Area of circle: 50.26548245743669
```

Protected class members

```python
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    def display_info(self):
        return f"Name: {self._name}, Age: {self._age}"

# Creating an instance of Person
person = Person("Alice", 30)

# Accessing protected attributes (not enforced)
print(person._name)  # Output: Alice
print(person._age)   # Output: 30

# Accessing method to display information (recommended way)
print(person.display_info())
# Output: Name: Alice, Age: 30
```

Reduction to the base type

```
# Define the base class
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        return "Some generic sound"

# Define a subclass
class Dog(Animal):
    def make_sound(self):
        return "Bark"

# Create an instance of Dog
dog = Dog("Buddy")

# Treat the Dog object as an Animal
animal = Animal("Max")
animal = dog  # Reducing Dog to Animal

# Use methods of the base type
print(animal.name)
# Output: Buddy
print(animal.make_sound())
# Output: Bark
```

Methods:

Array of parameters

```
def sum_numbers(*args):
    total = 0
    for num in args:
        total += num
    return total

# Using the sum_numbers method with different numbers of
arguments
print(sum_numbers(1, 2, 3))
# Output: 6
print(sum_numbers(1, 2, 3, 4, 5))
# Output: 15
print(sum_numbers(10, 20, 30, 40, 50))
# Output: 150
```

Class methods

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display_info(self):
        return f"Name: {self.name}, Age: {self.age}"

    @classmethod
    def from_string(cls, string):
        name, age = string.split(',')
        return cls(name.strip(), int(age.strip()))

# Using the class method to create Person objects
person1 = Person.from_string("Alice, 30")
person2 = Person.from_string("Bob, 25")

# Displaying information of created Person objects
print(person1.display_info())
# Output: Name: Alice, Age: 30
print(person2.display_info())
# Output: Name: Bob, Age: 25
```

In/Out parameters

```
def double_numbers(numbers):
    for i in range(len(numbers)):
        numbers[i] *= 2
    return numbers

# Original list of numbers
original_numbers = [1, 2, 3, 4, 5]

# Calling the method with the original list
modified_numbers = double_numbers(original_numbers)

# Displaying the modified list
print("Modified Numbers:", modified_numbers)
# Output: Modified Numbers: [2, 4, 6, 8, 10]

# Original list remains unchanged
print("Original Numbers:", original_numbers)
# Output: Original Numbers: [1, 2, 3, 4, 5]
```

Multiple return values

```python
import math

def get_circle_info(radius):
    area = math.pi * radius**2
    circumference = 2 * math.pi * radius
    return area, circumference

# Calling the method and unpacking the returned tuple
circle_area, circle_circumference = get_circle_info(5)

# Displaying the results
print("Circle Area:", circle_area)
# Output: Circle Area: 78.53981633974483
print("Circle Circumference:", circle_circumference)
# Output: Circle Circumference: 31.41592653589793
```

Optional parameter values

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

# Calling the method with and without providing a custom message
print(greet("Alice"))
# Output: Hello, Alice!
print(greet("Bob", "Hi there"))
# Output: Hi there, Bob!
```

Variable parameters

```
def sum_numbers(*args):
    total = 0
    for num in args:
        total += num
    return total

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

# Using the sum_numbers method with different numbers of positional arguments
print("Sum:", sum_numbers(1, 2, 3))
# Output: Sum: 6
print("Sum:", sum_numbers(1, 2, 3, 4, 5))
# Output: Sum: 15
print("Sum:", sum_numbers(10, 20, 30, 40, 50))
# Output: Sum: 150

# Using the print_info method with different numbers of keyword arguments
print_info(name="Alice", age=30)
# Output: name: Alice, age: 30
print_info(name="Bob", age=25, city="New York")
# Output: name: Bob, age: 25, city: New York
```

With return value

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

# Calling the method and storing the returned value
result = add_numbers(3, 5)

# Displaying the returned value
print("Result:", result)  # Output: Result: 8
```

Without any parameters

```
from datetime import datetime

def get_current_year():
    return datetime.now().year

# Calling the method
current_year = get_current_year()

# Displaying the current year
print("Current Year:", current_year)
```

Without any return value

```python
def print_message(message):
    print("Message:", message)

# Calling the method
print_message("Hello, World!")
```

Nested class

```python
class Outer:
    def __init__(self, name):
        self.name = name
        self.inner = self.Inner()

    def display_outer(self):
        print("Outer Name:", self.name)

    class Inner:
        def display_inner(self):
            print("Inner Class")

# Creating an instance of the outer class
outer_obj = Outer("Outer Object")

# Accessing methods of the outer class
outer_obj.display_outer()
# Output: Outer Name: Outer Object

# Accessing methods of the inner class
inner_obj = outer_obj.inner
inner_obj.display_inner()
# Output: Inner Class
```

Properties:

Computed properties

```python
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def area(self):
        return math.pi * self.radius ** 2

# Creating an instance of Circle
circle = Circle(5)

# Accessing the computed property
print("Radius:", circle.radius)
# Output: Radius: 5
print("Area:", circle.area)
# Output: Area: 78.53981633974483
```

Lazy properties

```python
import math

class LazyProperty:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, owner):
        if instance is None:
            return self
        value = self.func(instance)
        setattr(instance, self.func.__name__, value)
        return value

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @LazyProperty
    def area(self):
        print("Calculating area...")
        return math.pi * self.radius ** 2

# Creating an instance of Circle
circle = Circle(5)

# Accessing the lazy property
print("Radius:", circle.radius)
# Output: Radius: 5
print("Area:", circle.area)
# Output: Calculating area... \n Area: 78.53981633974483
print("Area:", circle.area)
# Output: Area: 78.53981633974483 (no re-calculation)
```

Read-Only properties:
Computed properties

```
import math


class Circle:
    def __init__(self):
        self.radius = 0

    @property
    def area(self):
        return math.pi * pow(self.radius, 2)


circle = Circle()
circle.radius = 2
# circle.area is 12.566370614359172

print(circle.area)
```

Read-Only properties: Stored properties

```
class FilmList:
    def __init__(self):
        self.__count = 10

    @property
    def count(self):
        return self.__count


filmList = FilmList()
count = filmList.count

print(count) # count is  10
```

Stored properties

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Creating an instance of Person
person = Person("Alice", 30)

# Accessing stored properties
print("Name:", person.name)  # Output: Name: Alice
print("Age:", person.age)    # Output: Age: 30

# Modifying stored properties
person.name = "Bob"
person.age = 25

# Displaying modified properties
print("Modified Name:", person.name)
# Output: Modified Name: Bob
print("Modified Age:", person.age)
# Output: Modified Age: 25
```

Type properties

```python
class Circle:
    pi = 3.14159

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

    def calculate_area(self):
        return Circle.pi * self.radius ** 2

# Creating instances of Circle
circle1 = Circle(5)
circle2 = Circle(10)

# Accessing the type property
print("Value of pi:", Circle.pi)  # Output: Value of pi: 3.14159

# Calculating areas using type property
print("Area of circle 1:", circle1.calculate_area())
# Output: Area of circle 1: 78.53975
print("Area of circle 2:", circle2.calculate_area())
# Output: Area of circle 2: 314.159
```

Subscripts (indexer methods):

With generic parameter

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

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

    def __setitem__(self, index, value):
        self.data[index] = value

# Creating an instance of MyList
my_list = MyList()

# Using integer indices
my_list[0] = 'a'
my_list[1] = 'b'
print("Element at index 0:", my_list[0])
# Output: Element at index 0: a
print("Element at index 1:", my_list[1])
# Output: Element at index 1: b

# Using string keys
my_list['first'] = 10
my_list['second'] = 20
print("Element with key 'first':", my_list['first'])
# Output: Element with key 'first': 10
print("Element with key 'second':", my_list['second'])
# Output: Element with key 'second': 20
```

With multiple parameter

```
class Matrix:
    def __init__(self, rows, columns):
        self.rows = rows
        self.columns = columns
        self.data = [[0] * columns for _ in range(rows)]

    def __getitem__(self, indices):
        row, column = indices
        return self.data[row][column]

    def __setitem__(self, indices, value):
        row, column = indices
        self.data[row][column] = value

# Creating an instance of Matrix
matrix = Matrix(3, 3)

# Setting values using multiple indices
matrix[0, 0] = 1
matrix[1, 1] = 2
matrix[2, 2] = 3

# Getting values using multiple indices
print("Value at position (0, 0):", matrix[0, 0])
# Output: Value at position (0, 0): 1
print("Value at position (1, 1):", matrix[1, 1])
# Output: Value at position (1, 1): 2
print("Value at position (2, 2):", matrix[2, 2])
# Output: Value at position (2, 2): 3
```

With one parameter

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

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

    def __setitem__(self, index, value):
        self.data[index] = value

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

# Accessing elements using single index
print("Element at index 0:", my_list[0])
# Output: Element at index 0: 1
print("Element at index 2:", my_list[2])
# Output: Element at index 2: 3

# Modifying elements using single index
my_list[1] = 10
my_list[3] = 20
print("Modified list:", my_list.data)
# Output: Modified list: [1, 10, 3, 20, 5]
```

Type member

```python
class Employee:
    # Class variable
    company_name = "TechCorp"
    employee_count = 0

    def __init__(self, name, position):
        self.name = name
        self.position = position
        Employee.employee_count += 1

    # Class method
    @classmethod
    def set_company_name(cls, name):
        cls.company_name = name

    # Class method to get employee count
    @classmethod
    def get_employee_count(cls):
        return cls.employee_count

# Accessing and modifying class variables
print("Company Name:", Employee.company_name)
# Output: Company Name: TechCorp
print("Initial Employee Count:",
Employee.employee_count)
# Output: Initial Employee Count: 0

# Creating instances of Employee
emp1 = Employee("Alice", "Developer")
emp2 = Employee("Bob", "Designer")

# Accessing class variable via instance
print("Company Name (via emp1):",
emp1.company_name)
# Output: Company Name (via emp1): TechCorp
```

```python
print("Employee Count (via emp1):", emp1.employee_count)
# Output: Employee Count (via emp1): 2

# Using class method to set company name
Employee.set_company_name("InnoTech")
print("Updated Company Name:", Employee.company_name)
# Output: Updated Company Name: InnoTech

# Using class method to get employee count
print("Total Employees:", Employee.get_employee_count())
# Output: Total Employees: 2
```

Control Flow

Control flow in programming determines the order in which instructions are executed. It encompasses decision-making, looping, and branching mechanisms that allow a program to execute different code paths based on conditions. Key constructs include conditional statements (if, else if, else) for decision-making, switch statements for handling multiple conditions, and loops (for, while, do...while) for repeating code. Control flow also involves breaking out of loops with "break" and skipping iterations with "continue". These constructs are fundamental for creating dynamic and responsive software that can adapt to various inputs and situations.

if/else statements:

Complex conditions

```
X = 10
Y = 20
Z = 30

if Z > X and Z > Y:
    if X < Y:
        print("Z is the largest and X is smaller than Y.")
    else:
        print("Z is the largest but X is not smaller than Y.")
else:
    print("Z is not the largest.")
# Output: Z is the largest and X is smaller than Y.
```

Is not valid example

```python
# Invalid example
if latitud == 0  # SyntaxError: invalid syntax
location = "Equator"
```

Ternary operator

```python
n = -42
classify = "positive" if n > 0 else "negative"

print(classify)  # Output: negative
```

Valid example

```python
import random

def get_latitude():
    return random.randint(-90, 90)

latitude = get_latitude()
location = ""

if latitude == 0:
    location = "Equator"
elif latitude == 90:
    location = "North Pole"
elif latitude == -90:
    location = "South Pole"
else:
    location = "Not at the Equator or Pole"

print(f"latitude is {latitude}")
# Example output: latitude is -57
print(f"location is \"{location}\"")
# Example output: location is "Not at the Equator or Pole"
```

Match statements:

Different types of values

```python
monitor_inch_size = 24

match monitor_inch_size:
    case 15:
        str = "too small"
    case 16 | 17 | 18:
        str = "good for the past decade"
    case 19 | 20 | 21 | 22 | 23:
        str = "for office work"
    case 24 | 25 | 26 | 27:
        str = "great choice"
    case _:
        str = ""

print(f'str is "{str}"')
# Output: str is "great choice"
```

Example with a tuple

```
message = ("error", 404, "Not Found")

match message:
    case ("error", code, description):
        result = f"Error {code}: {description}"
    case ("warning", description):
        result = f"Warning: {description}"
    case ("info", description):
        result = f"Info: {description}"
    case ("success", code, description):
        result = f"Success {code}: {description}"
    case _:
        result = "Unknown message type"

print(result)  # Output: Error 404: Not Found
```

Match if conditions

```
numbers = [5, -2, 0, 10, -8]

for number in numbers:
    match number:
        case n if n > 0:
            print(f"{n} is positive")
        case n if n < 0:
            print(f"{n} is negative")
        case 0:
            print("Zero")
        case _:
            print("Unknown number")
```

Simple conditions

```
# Define a function to calculate the tax based on income
def calculate_tax(income):
    match income:
        case x if x <= 10000:
            tax = x * 0.1
        case x if 10000 < x <= 50000:
            tax = 10000 * 0.1 + (x - 10000) * 0.2
        case x if x > 50000:
            tax = 10000 * 0.1 + 40000 * 0.2 + (x - 50000) * 0.3
    return tax

# Test the function
print("Tax for $5000:", calculate_tax(5000))
# Tax for $5000: 500.0
print("Tax for $25000:", calculate_tax(25000))
# Tax for $25000: 4000.0
print("Tax for $75000:", calculate_tax(75000))
# Tax for $75000: 17000.0
```

Interruption of a control flow:

“break statement”

```python
# Example using a while loop
number = 0
while number < 5:
    print(number)
    if number == 3:
        break  # Exit the loop when number reaches 3
    number += 1
print("Loop ended")
```

“continue statement”

```
# Example using a for loop
for i in range(5):
    if i == 2:
        continue  # Skip the rest of the loop when i is 2
    print(i)
```

With return value

```
# Define a function to calculate the square of a number
def square(x):
    return x ** 2  # Return the square of the input value

# Call the function and store the result in a variable
result = square(5)

# Print the result
print("Square of 5 is:", result)
```

With return value

```
# Define a function to print a message and return
def print_and_return():
    print("Function execution is complete.")
    return  # No value is returned

# Call the function
print_and_return()
print("After function call")
```

Loops:

“do-while” loop

```
i = 7
f7 = 1

while i > 1:
    f7 *= i
    i -= 1

print(f'f7 is {f7}')
# Output: f7 is 5040
```

“for in range” loop

```
f7 = 1

for i in range(7, 1, -1):
    f7 *= i

print(f'f7 is {f7}')  # Output: f7 is 5040
```

“for-in” loop

```python
# Example with a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)

# apple
# banana
# cherry
```

“while” loop

```python
# Initialize a counter
i = 0

# Define a while loop
while i < 5:
    print(i)
    i += 1  # Increment the counter

# 0
# 1
# 2
# 3
# 4
```

Endless loop

```
while True:
    # statements
```

Enumerations

Enumerations, or enums, are a data type that consists of a set of named values called elements or members. Enums are used to represent a collection of related constants in a readable and maintainable way. They enhance code clarity and safety by providing meaningful names for sets of values, reducing errors from using arbitrary numbers or strings. Enums are commonly used in scenarios like defining states, categories, or types where a variable can only take one out of a small set of possible values. This makes the code more intuitive and less prone to mistakes.

Base member value

```python
from enum import Enum

# Define an enumeration class
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

# Access the value of an enumeration member
red_value = Color.RED.value
print("Value of RED:", red_value)
# Output: Value of RED: 1
```

Base type

```
from enum import Enum

# Define an enumeration class
class DataType(Enum):
    INTEGER = 42
    FLOAT = 3.14
    STRING = "hello"
    CUSTOM_OBJECT = {"name": "John", "age": 30}

# Accessing enumeration members and their data types
print("Integer value:", DataType.INTEGER.value, "Type:", type(DataType.INTEGER.value))
# Integer value: 42 Type: <class 'int'>
print("Float value:", DataType.FLOAT.value, "Type:", type(DataType.FLOAT.value))
# Float value: 3.14 Type: <class 'float'>
print("String value:", DataType.STRING.value, "Type:", type(DataType.STRING.value))
# String value: hello Type: <class 'str'>
print("Custom object value:", DataType.CUSTOM_OBJECT.value, "Type:", type(DataType.CUSTOM_OBJECT.value))
# Custom object value: {'name': 'John', 'age': 30} Type: <class 'dict'>
```

Conversion from a string

```
from enum import Enum

# Define an enumeration class
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

# Convert a string to an enumeration member
def string_to_enum(string_value):
    try:
        enum_member = Color[string_value]
        return enum_member
    except KeyError:
        print(f"No enum member found for {string_value}")
        return None

# Test the conversion
color_string = "GREEN"
color_enum_member = string_to_enum(color_string)
if color_enum_member:
    print(f"Enum member for {color_string}: {color_enum_member}")
    # Enum member for GREEN: Color.GREEN
```

Converting to a String

```
from enum import Enum

# Define an enumeration class
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

# Convert an enumeration member to a string
def enum_to_string(enum_member):
    return str(enum_member)  # Using str() function

# Test the conversion
color_enum_member = Color.GREEN
color_string = enum_to_string(color_enum_member)
print(f"String representation: {color_string}")
# String representation: Color.GREEN

# Alternatively, directly access the name attribute
color_string = color_enum_member.name
print(f"String representation (using name attribute): {color_string}")
# String representation (using name attribute): GREEN
```

Definition and initialization

```
from enum import Enum

class Season(Enum):
    Summer, Fall, Winter, Spring = range(4)

summer = Season.Summer
winter = Season.Winter

print(summer) # Season.Summer
print(winter) # Season.Winter
```

Enums comparison

```
from enum import Enum

class Size(Enum):
    xs, s, m, l, xl = range(5)

small = Size.s
large = Size.l

print("is l > s:", large.value > small.value)
# is l > s: True
```

Explicitly set base value

```python
from enum import Enum

class Season(Enum):
    Summer = 1
    Fall = 2
    Winter = 3
    Spring = 4

winter = Season.Winter
baseWinter = winter.value

print(baseWinter) # 3
```

Get the list of values

```
from enum import Enum

class Season(Enum):
    Summer, Fall, Winter, Spring = range(4)

values = list(Season)

print(values)
print(values[0])
# [<Season.Summer: 0>, <Season.Fall: 1>,
<Season.Winter: 2>, <Season.Spring: 3>]
# Season.Summer
```

Initializing from a base value

```
from enum import Enum

class Season(Enum):
    Summer = 0
    Fall = 1
    Winter = 2
    Spring = 3

winter = Season(2)
# winter is Season.Winter

print(winter) # Season.Winter
```

Exceptions Handling

Exceptions handling is a programming technique used to manage unexpected or erroneous situations that may occur during runtime. When a program encounters an exceptional condition (e.g., division by zero, file not found), it throws an exception, which disrupts the normal flow of execution.

Catch all exceptions

```
class IsNoneException(Exception):
    pass

class IsEmptyException(Exception):
    pass

def throw_when_null_or_empty(data):
    if data is None:
        raise IsNoneException()

    if len(data) == 0:
        raise IsEmptyException()

try:
    throw_when_null_or_empty(None)
except Exception as e:
    print("Error happened " + e.__class__.__name__)

# Error happened IsNoneException
```

Catch the specific exception

```python
class IsNoneException(Exception):
    pass

class IsEmptyException(Exception):
    pass

def throw_when_null_or_empty(data):
    if data is None:
        raise IsNoneException()

    if len(data) == 0:
        raise IsEmptyException()

try:
    throw_when_null_or_empty([])
except IsNoneException:
    print("list is not specified")
except IsEmptyException:
    print("list is empty")

# list is empty
```

Define an exception type

```python
class SimpleException(Exception):
    pass

raise SimpleException("Oops!")
```

Guaranteed code execution

```python
def throw_if_true(param):
    try:
        if param:
            raise OSError("test exception")
    except OSError:
        print("except")
    finally:
        print("finally")

throw_if_true(True)
# printed: "except" and "finally"
throw_if_true(False)
# printed only "finally"
```

If no exception occurred

```
def throw_if_true(param):
    try:
        if param:
            raise OSError("test exception")
    except OSError:
        print("except")
    else:
        print("else")

throw_if_true(True)
# printed: "except"
throw_if_true(False)
# printed only "else"
```

Method throwing an exception

```python
# any method can throw an error
def method_with_exception():
    raise Exception("test exception")

method_with_exception()

# Exception: test exception
```

Re-throw exceptions

```python
def method_with_exception():
    try:
        raise Exception("test exception")
    except Exception as ex:
        # implementation of any partial procesing
        # and send error to the calling code
        raise ex

try:
    method_with_exception()
except Exception as e:
    print(e.args[0])

# test exception
```

Throw an exception

```
class Seller:
    def __init__(self):
        self.cars = []

    def sell(self):
        if len(self.cars) == 0:
            raise Exception("No cars for sale")

seller = Seller()
try:
    seller.sell()
except Exception as e:
    print(e.args[0])
    # e.args[0] is "No cars for sale"
```

Extensions

Extensions in programming languages allow developers to enhance existing types or classes without modifying their source code. They provide a way to add new functionality, methods, or properties to types that are already defined.

Adding object methods

```
from math import *

excluded_methods = frozenset(["__module__",
"__qualname__"])

def class_extend(cls):
    class Meta(type):
        def __new__(mcs, name, bases, attrs):
            for name, value in attrs.items():
                if name not in excluded_methods:
                    setattr(cls, name, value)
            return cls
    return Meta

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

class Point(metaclass=class_extend(Point)):
    def distance_to(self, p2):
        d1 = pow(self.x - p2.x, 2)
        d2 = pow(self.y - p2.y, 2)
        return sqrt(d1 + d2)

point1 = Point(1, 2)
point2 = Point(2, 3)
distance = point1.distance_to(point2)

print(f"{distance = }")
# distance = 1.4142135623730951
```

Functions

Functions in programming are blocks of reusable code that perform a specific task. They allow developers to encapsulate logic, promote code reusability, and enhance readability by breaking down complex operations into smaller, manageable parts.

Array of parameters

```python
def get_avg(*values):
    if len(values) == 0:
        return 0

    sum_v = 0
    for value in values:
        sum_v += value
    return sum_v / len(values)

avg = get_avg(1, 2, 3, 4)

print(f"{avg = }") # avg is 2.5
```

In/Out parameters

```python
def swap_strings(s1, s2):
    tmp = s1[0]
    s1[0] = s2[0]
    s2[0] = tmp

s1 = ["A"]
s2 = ["B"]
swap_strings(s1, s2)

print(f"s1[0] is {s1[0]}, s2[0] is {s2[0]}")
# s1[0] is "B", s2[0] is "A"
```

Multiple return values

```
def get_first_last(ar):
    if len(ar) == 0:
        return -1, -1
    return ar[0], ar[-1]

ar = [2, 3, 5]
first, last = get_first_last(ar)

print(f"first is {first}")  # first is 2
print(f"last is {last}")    # last is 5
```

Optional parameter values

```python
# Using Default Parameter Values in Python
def say_goodbye(message="Goodbye!"):
    print(message)

say_goodbye()
# prints "Goodbye!"

say_goodbye("See you")
# prints "See you"

# Before Using Default Parameters
def old_say_goodbye(message=None):
    if message is None:
        message = "Goodbye!"
    print(message)

old_say_goodbye()
# prints "Goodbye!"

old_say_goodbye("See you")
# prints "See you"
```

Out parameters

```python
# in Python, you can't change param reference
def get_sum(summ, n1, n2):
    summ.append(n1 + n2):

ar_sum = []
get_sum(ar_sum, 5, 3)
# ar_sum is [13]
```

Recursion

```python
def fibonacci(x):
    return x if x <= 1 else fibonacci(x - 1) + fibonacci(x - 2)

f10 = fibonacci(10)

print(f"f10 is {f10}")  # f10 is 55
```

Variable parameters

```
def print5(data):
    if len(data) > 5:
        data = data[0: 5]

    print(data)

print5("1234567") # prints: 12345
```

With return value

```python
def get_sum(n1, n2):
    return n1 + n2

result = get_sum(5, 3)

print(f"{result = }") # result is 8
```

Without any parameters

```
def say_goodbye():
    print("Goodbye!")

say_goodbye()
```

Without any return value

```python
def add_3_and_print(value):
    print(value + 3)

add_3_and_print(5) # 8
```

Generic Types

Generic types in programming languages allow developers to define classes, functions, or interfaces that can work with various data types without specifying them beforehand. This flexibility enhances code reusability and type safety by enabling components to be more generic and adaptable to different scenarios.

Class conformity

```
from typing import TypeVar, Generic

class Vehicle:
    def test(self):
        print(f"test: {self}")

class Car(Vehicle):
    pass

class Truck:
    pass

T = TypeVar('T', bound=Vehicle)

class Service(Generic[T]):
    def __init__(self):
        self.v_list = list[T]()

    def add(self, item: T):
        self.v_list.append(item)

    def test(self):
        for item in self.v_list:
            item.test()

service = Service[Vehicle]()
service.add(Vehicle())
service.add(Car())
# Warning: Expected type 'Vehicle'
service.add(Truck())
service.test()
```

Default value

```
from typing import TypeVar, Generic, Type T = TypeVar('T')

class Size(Generic[T]):
    def __init__(self, width: T, height: T):
        self.width = width
        self.height = height

    def reset(self):
        self.width = type(self.width)()
        self.height = type(self.height)()

    def print(self):
        print(f{[{self.width}; {self.height}]})

size_int = Size[int](5, 9)
size_int.print()
# prints: [5; 9]
size_int.reset()
size_int.print()
# prints: [0; 0]
```

Generic classes

```
from typing import TypeVar, Generic
T = TypeVar('T')

class Size(Generic[T]):
    def __init__(self, width: T, height: T):
        self.width = width
        self.height = height

    def as_text(self):
        return f"[{self.width}; {self.height}]"

size_int = Size[int](5, 8)
text_int = size_int.as_text()
# text_int is "[5; 8]"

size_float = Size[float](3.7, 1.58)
text_float = size_float.as_text()
# textFloat is "[3.7; 1.58]"

print(f"{text_int=}")
print(f"{text_float=}")
```

Generic collections

```
# List of integer
int_list = list[int]()
int_list.append(5)
print(f"{int_list = }")

# Dictionary
dic = dict[int, str]()
dic[1] = "one"
print(f"{dic = }")

# Set
set_float = set[float]()
set_float.add(3.14)
print(f"{set_float = }")

# nt_list = [5]
# dic = {1: 'one'}
# set_float = {3.14}
```

Generic methods

```
from typing import TypeVar
T = TypeVar('T')

def swap(v1: list[T], v2: list[T]):
    v1[0], v2[0] = v2[0], v1[0]

n1 = [5]
n2 = [7]
swap(n1, n2)
# n1[0] is 7, n2[0] is 5

s1 = ["cat"]
s2 = ["dog"]
swap(s1, s2)
# s1[0] is "B", s2[0] is "A"

print(f'{n1 = }, {n2 = }')
print(f'{s1 = }, {s2 = }')
```

Interface conformity

```
from abc import ABC, abstractmethod
from typing import TypeVar, Generic

class Vehicle(ABC):
    @abstractmethod
    def test(self):
        pass

class Car(Vehicle):
    def test(self):
        print(f"test {self}")

T = TypeVar('T', bound=Vehicle)

class Service(Generic[T]):
    def __init__(self):
        self.v_list = list[T]()

    def add(self, item: T):
        self.v_list.append(item)

    def test(self):
        for item in self.v_list:
            item.test()

service = Service[Car]()
service.add(Car())
service.test()
```

Substitution principle

```python
class Vehicle:
    def test(self):
        print(f"test {self}")

class Car(Vehicle):
    pass

class Truck(Vehicle):
    pass

lst = list[Vehicle]()
lst.append(Vehicle())
lst.append(Car())
lst.append(Truck())

for vehicle in lst:
    vehicle.test()
```

Initializing of Types

Initializing types refers to the process of setting initial values or states for variables, objects, or data structures in a program. This process ensures that entities in the program start with predefined values, which are often crucial for correct functioning and behavior.

Classes:

With a constructor

```
class Phone:
    def __init__(self, model):
        self.model = model

class Employee:
    def __init__(self, first_name, last_name, phone):
        self.first_name = first_name
        self.last_name = last_name
        self.phone = phone

# Create instances
nokia_phone = Phone("Nokia 6610")
kim = Employee("Victorya", "Kim", Phone("IPhone 11 Pro"))

# Access and print phone model
print(kim.phone.model) # Iphone 11 Pro
```

Without any constructor

```
class Phone:
    pass  # No explicit constructor needed

class Employee:
    pass  # No explicit constructor needed

# Create instances and assign attributes
nokia_phone = Phone()
nokia_phone.model = "Nokia 6610"

kim = Employee()
kim.firstName = "Victorya"
kim.lastName = "Kim"
kim.phone = Phone()
kim.phone.model = "IPhone 5"

# Access and print phone model
print(kim.phone.model) # Iphone 5
```

Collections:

Dictionaries

```
# Dictionary<String, String>
languages = {"ru": "russian", "en": "english"}

# Dictionary<Int, String>
numbers = {1: "one", 2: "two", 3: "three"}

# Dictionary<Int, Employee>
class Employee:
    def __init__(self, first_name, last_name):
        self.firstName = first_name
        self.lastName = last_name

employees = {
    1: Employee("Anton", "Pavlov"),
    2: Employee("Elena", "Kirienko")
}

print(f"{languages = }")
# languages = {'ru': 'russian', 'en': 'english'}
print(f"{numbers = }")
# numbers = {1: 'one', 2: 'two', 3: 'three'}
print(f"{employees = }")
# employees = {1: <__main__.Employee object at 0x000001B63A33C950>, 2: <__main__.Employee object at 0x000001B63A33C980>}
```

Lists

```
# list of integer
primeNumbers = [2, 3, 5, 7, 11, 13, 17, 19]

# list of string
gameList = ["soccer", "hockey", "basketball"]


# list of Employee
class Employee:
    def __init__(self, first_name, last_name):
        self.firstName = first_name
        self.lastName = last_name


employess = [Employee("Pavlov", "Anton"), Employee("Kirienko", "Elena")]

print(f"{primeNumbers = }")
# primeNumbers = [2, 3, 5, 7, 11, 13, 17, 19]
print(f"{gameList = }")
# gameList = ['soccer', 'hockey', 'basketball']
print(f"{employess = }")
# employess = [<__main__.Employee object at 0x0000015D2F5FC830>, <__main__.Employee object at 0x0000015D2F5FC860>]
```

Set

```
intHashSet = {2, 3, 5, 7, 11, 13, 17, 19}

print(intHashSet)
# {2, 3, 5, 7, 11, 13, 17, 19}
```

Enumerations

```python
from enum import Enum

class PreciousMetal(Enum):
    Platinum = 1
    Gold = 2
    Silver = 3

class Season(Enum):
    Summer, Fall, Winter, Spring = range(4)

Planet = Enum('Planet', 'Mercury Venus Earth')

gold = PreciousMetal.Gold
fall = Season.Fall
earth = Planet.Earth

print(f"{gold = }")
# gold = <PreciousMetal.Gold: 2>
print(f"{fall = }")
# fall = <Season.Fall: 1>
print(f"{earth = }")
# earth = <Planet.Earth: 3>
```

Simple types

```
import sys
from typing import Final

# "Final" for constants

# Int
number: int = 42
otherNumber = 37
maxInt = sys.maxsize
MB: Final = 103876

# Float
exp: float = 2.71828
billion = 1E+9

# String
greeting: Final[str] = "Hello"

# MultiLine String
text1 = "this is some\n + \
    multiLine text"

text2: str = """this is some
multiLine text"""

text3 = ("this is some\n"
        "multiLine text")

# Bool
sunIsStar = True
earthIsStar = False

# Character "A"
charA = 'A' # '\u0041', chr(65);

# Tuple (Int, String)
one = (1, "one")
```

```python
print(f"{number = }")
# number = 42
print(f"{otherNumber = }")
# otherNumber = 37
print(f"{maxInt = }")
# maxInt = 9223372036854775807
print(f"{MB = }")
# MB = 103876
print(f"{exp = }")
# exp = 2.71828
print(f"{billion = }")
# billion = 1000000000.0
print(f"{greeting = }")
# greeting = 'Hello'
print(f"{text1 = }")
# text1 = 'this is some\n +      multiLine text'
print(f"{text2 = }")
# text2 = 'this is some\nmultiLine text'
print(f"{text3 = }")
# text3 = 'this is some\nmultiLine text'
print(f"{sunIsStar = }")
# sunIsStar = True
print(f"{earthIsStar = }")
# earthIsStar = False
print(f"{charA = }")
# charA = 'A'
print(f"{one = }")
# one = (1, 'one')
```

Structures:

With a constructor

```python
# The Python language has no structure
class Size:
    def __init__(self, width, height):
        self.width = width
        self.height = height

class Point:
    def __init__(self, top, left):
        self.top = top
        self.left = left

class Rectangle:
    def __init__(self, p_size, p_point):
        self.size = p_size
        self.point = p_point

size = Size(10, 10)
point = Point(5, 5)
rect = Rectangle(size, point)

print(rect.point.left) # 5
```

Without any constructor

```python
# The Python language has no structures
class Size:
    width = 0
    height = 0

class Point:
    top = 0
    left = 0

class Rectangle:
    size = Size()
    point = Point()

rect = Rectangle()
rect.size.width = 10
rect.size.height = 10
rect.point.top = 5
rect.point.left = 5

print(rect.point.left)
```

Lambda Expressions

Lambda expressions, also known as anonymous functions, provide a concise way to define small, inline functions in programming languages that support functional programming paradigms. They are used primarily for short and simple functions without the overhead of traditional function declaration syntax. Lambda expressions are especially useful in functional-style programming where functions are treated as first-class citizens and can be passed as arguments to other functions. They typically use arrow notation (`=>`) for defining the function body and are widely used in languages like Python, JavaScript, Java, C#, and more.

Capture of variables

```python
def make_increment(n):
    return lambda x: x + n

inc3 = make_increment(3)
value = 5
inc5 = make_increment(value)

x1 = inc3(10)
# x1 is 13

x2 = inc5(50)
# x2 is 55

print(f"{x1 = }")
print(f"{x2 = }")
```

Currying

```python
def carry(f):
    return lambda a: lambda b: f(a, b)

def avg(a, b): return (a + b) / 2

n1 = avg(1, 3)
# n1 is 2.0

# first universal method
avg1 = carry(avg)(1)
# avg1 is avg func with first param = 1
n2 = avg1(5)
# n2 is 3.0 = (1 + 5) / 2

print("n1 is", n1)
print("n2 is", n2)
```

Function as a parameter

```
numbers = [2, 3, 1, 7, 9]
numbers1 = list(map(lambda x: x * 2 + 1 , numbers))
# numbers1 is [5, 7, 3, 15, 19]

numbers2 = list(filter(lambda x: x % 3 == 0, numbers1))
# numbers2 is [3, 9]

print(numbers1) # [5, 7, 3, 15, 19]
print(numbers2) # [3, 15]
```

Function as a return value

```
def make_sum_func():
    return lambda a, b: a + b

sumFunc = make_sum_func()
sumValue = sumFunc(5, 8)

print(f"{sumValue = }") # sumValue is 13
```

Memoization

```python
from datetime import datetime

def memoize(f):
    memo = dict()

    def memo_fun(x):
        if x in memo:
            return memo[x]
        r = f(x)
        memo[x] = r
        return r
    return memo_fun

def fibonacci(x):
    return x if (x <= 1) else fibonacci(x - 1) + fibonacci(x - 2)

mem_fibonacci = memoize(fibonacci)
for i in range(1, 3):
    start = datetime.now()
    f37 = mem_fibonacci(37)
    delta = datetime.now() - start
    seconds = delta.total_seconds()
    print(f"{i}: f37 is {f37}" )
    print(f"{i}: seconds is {seconds}")

# prints:
# 1: f37 is 24157817
# 1: seconds is 7.296308
# 2: f37 is 24157817
# 2: seconds is 0.0

start = datetime.now()
f38 = mem_fibonacci(38)
delta = datetime.now() - start
seconds = delta.total_seconds()
print(f"f38 is {f38}")
```

```
print(f"seconds is {seconds}")
# f38 is 39088169
# seconds is 12.796998
```

Memoization (Recursive)

```
from datetime import datetime

def memoize(f):
    memo = dict()

    def memo_fun(x):
        if x in memo:
            return memo[x]
        r = f(memo_fun, x)
        memo[x] = r
        return r

    return memo_fun

def fib(f, x):
    return x if (x <= 1) else f(x - 1) + f(x - 2)

mem_fibonacci = memoize(fib)
for i in range(1, 3):
    start = datetime.now()
    f37 = mem_fibonacci(37)
    delta = datetime.now() - start
    microseconds = delta.seconds * 1000000 + delta.microseconds
    print(f"{i}: f37 is {f37}")
    print(f"{i}: microseconds is {microseconds}")

# prints:
# 1: f37 is 24157817
# 1: microseconds is 10003
# 2: f37 is 24157817
# 2: microseconds is 0

start = datetime.now()
f38 = mem_fibonacci(38)
delta = datetime.now() - start
```

```
microseconds = delta.seconds * 1000000 + delta.microseconds
print(f"f38 is {f38}")
print(f"microseconds is {microseconds}")
# f38 is 39088169
# microseconds is 23187
```

Modify captured variables

```
x = 5

addYtoX = lambda y: x += y # <- Error
```

Recursion

```python
def fibonacci(x):
    return x if x <= 1 else fibonacci(x - 1) + fibonacci(x - 2)

f10 = fibonacci(10)
print(f"f10 is {f10}")  # Output: f10 is 55
```

Void function as a parameter

```
def check_and_process(number, process):
    if number < 10:
        process(number)

check_and_process(5, lambda number: print(number * 10))
# printed: 50
```

With multiple operators

```
from math import *

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

# you can't put multiple statements in a lambda
def get_distance(p1, p2):
    d1 = pow(p1.x - p2.x, 2)
    d2 = pow(p1.y - p2.y, 2)
    return sqrt(d1 + d2)

point1 = Point(0, 0)
point2 = Point(5, 5)
distance = get_distance(point1, point2)
# distance is 7.071
print(f"{distance = }")
```

With multiple parameters

```
# not recommended in PEP 8
avg_lambda = lambda a, b: (a + b) / 2
avg1 = avg_lambda(3, 5)
# avg1 is 4.0

# recommended
def avg_func(a, b):
    return (a + b) / 2

avg2 = avg_func(2, 7)
# avg2 is 4.5

print(f"avg1 = {avg1}")
print(f"avg2 = {avg2}")
```

With one parameter

```python
# not recommended in PEP 8
powOfTwo = lambda power: pow(2.0, power)
pow8 = powOfTwo(8)
# pow8 is 256.0

# recommended
def pow_of_three(power):
    return pow(3.0, power)

pow3 = powOfTwo(3)
# pow3 is 27.0

print(f"{pow8 =}")
print(f"{pow3 =}")
```

Without return value

```
# not recommended in PEP 8
add2AndPrint = lambda a: print(a + 2)
add2AndPrint(5)
# printed 7

# recommended
def add3_and_print(a):
    print(a + 3)

add3_and_print(7)
# printed 10
```

Lists and Collections

Lists and collections refer to data structures that allow grouping and managing multiple elements in programming. These structures are essential for storing, accessing, and manipulating data efficiently. Lists, often synonymous with arrays in some languages, are ordered collections where each element is indexed starting from zero. They can hold elements of the same type or even mixed types depending on the language's flexibility.

Dictionaries:

Adding and removing of elements

```
dic = {1: "one", 2: "two"}
print(f"{dic = }")

dic[3] = "three"
# dic is {1: 'one', 2: 'two', 3: 'three'}
print(f"{dic = }")

dic[3] = "three"
# dic is {1: 'one', 2: 'two', 3: 'three'}
print(f'{dic = }')

dic.pop(3)
# dic is {1: 'one', 2: 'two'}
print(f'{dic = }')

del dic[2]
# dic is {1: 'one'}
print(f'{dic = }')

dic.clear()
# dic is empty
print(f'{dic = }')
```

Amount of elements

```
dic = {1: "one", 2: "two"}
count = len(dic)
# count is 2

print(f'{count = }')
```

Checking of presence of a key

```
dic = {1: "one", 2: None}
exists1 = 1 in dic
# exists1 is True

exists2 = 2 in dic
# exists2 is True

exists3 = 3 in dic
# exists3 is False

print(f'{exists1 = }')
print(f'{exists2 = }')
print(f'{exists3 = }')
```

Converting a dictionary

```
dic = {1: "one", 2: "two"}
upperDic = {k: v.upper() for k, v in dic.items()}

print(f'{upperDic = }')
```

Default value

```python
dic = {1: "A", 2: "B"}
# value1 = dic[3] # <- Error
# value1 is nil
value2 = dic.get(3, "-")
# value2 is "-"

print(f'{value2 = }')
```

Dictionaries initialization

```
# Empty dictionary
d1 = {}
d2 = dict()

# init with some data
d3 = {1: "one", 2: "two"}
d4 = dict(one=1, two=2)
# d4 is {'one': 1, 'two': 2}

d5 = dict(d4, three=3)
#d4 is {'one': 1, 'two': 2, 'three' : 3}

print(f'{d1 = }')
print(f'{d2 = }')
print(f'{d3 = }')
print(f'{d4 = }')
print(f'{d5 = }')
```

Dictionary Merge

```
d1 = {1: "one"}
d2 = {2: "two"}
d3 = {3: "three"}

dAll = d1 | d2
print(f'{dAll = }')
# dAll is {1: 'one', 2: 'two'}

dAll |= d3
print(f'{dAll =}')
# dAll is {1: 'one', 2: 'two', 3: 'three'}
```

Filtering of elements

```
dic = {1: "one", 2: "two", 3: "three"}
oddDic = {k: v for k, v in dic.items() if k % 2 == 1}

# oddDic is {1: 'one', 3: 'three'}

print(f'{oddDic = }')
```

Get value by key

```python
d = {1: "one", 2: "two"}

one = d[1]
# one is "one"

two = d[2]
# two is "two"

# three = d[3] # <-Error

print(f'{one = }')
print(f'{two = }')
```

Getting keys by value

```
dic = {1: "A", 2: "B", 3: "A"}
valueTwo = "A"
keys = []

for key, value in dic.items():
    if value == valueTwo:
        keys.append(key)

# keys is [1, 3]
print(f'{keys = }')
```

Getting of a list of keys

```
dic = {1: "one", 2: "two"}
keys = list(dic.keys())
# keys is [1, 2]

print(f'{keys = }')
```

Getting of a list of values

```python
dic = {1: "one", 2: "two"}
values = list(dic.values())
# values is ["one", "two"]

print(f'{values = }')
```

Grouping collection

```python
numbers = [1, 2, 3, 4, 5]
arr = [[y for y in numbers if y % 2 == x] for x in [0, 1]]
dic = {"even": arr[0], "odd": arr[1]}
# dic is {'even': [2, 4], 'odd': [1, 3, 5]}

print(f"{dic = }")
```

Iterating over a dictionary

```
dic = {1: "one", 2: "two"}

str1 = ""
for key, value in dic.items():
    str1 += ("{" if str1 == "" else ", ") + f"{key} : \"
{value}\""

str1 += "}"
# str1 is "{1: "one", 2: "two"}"

str2 = ""
for value in dic.values():
    str2 += ("" if str2 == "" else ", ") + value

# str2 is "one, two"

print(f'{str1 = }')
print(f'{str2 = }')
```

Sort dictionary by keys

```
import operator

dic = {3: 'three', 1: 'one', 2: 'two'}

sorted_dic = sorted(dic.items(), key=operator.itemgetter(0))
# sorted_dic is {1: 'one', 2: 'two', 3: 'three'}

print(f'{sorted_dic = }')
```

Sort dictionary by values

```
import operator

dic = {3: 'B', 1: 'C', 2: 'A'}

sorted_dic = sorted(dic.items(), key=operator.itemgetter(1))
# sorted_dic is {2: 'A', 3: 'B', 1: 'C'}

print(f'{sorted_dic = }')
```

Iterators and generators:

Reverse generator

```
def reverse(data):
    current = len(data)
    while current >= 1:
        current -= 1
        yield data[current]

for c in reverse("string"):
    print(c)
# printed: g, n, i, r, t, s

for i in reverse([1, 2, 3]):
    print(i)
# printed: 3, 2, 1
```

Reverse iterator

```
class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index -= 1
        return self.data[self.index]

# Testing the Reverse iterator with a string
for c in Reverse("string"):
    print(c)
# Output: g, n, i, r, t, s

# Testing the Reverse iterator with a list
for i in Reverse([1, 2, 3]):
    print(i)
# Output: 3, 2, 1
```

Simple generator

```
def counter(low, high, step):
    current = low
    while current <= high:
        yield current
        current += step

for c in counter(3, 9, 2):
    print(c)
# printed 3, 5, 7, 9
```

Simple iterator

```
class Counter:
    def __init__(self, low, high, step):
        self.current = low
        self.high = high
        self.step = step

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.high
            raise StopIteration
        else:
            result = self.current
            self.current += self.step
            return result

for c in Counter(3, 9, 2):
    print(c)
# printed 3, 4, 7, 9
```

Lists:

Adding and removing of elements

```
primeNumbers = [2, 5, 7]
print(primeNumbers)

primeNumbers.append(11)
# primeNumbers is [2, 5, 7, 11]
print(primeNumbers)

primeNumbers.insert(1, 3)
# primeNumbers is [2, 3, 5, 7, 11]
print(primeNumbers)

primeNumbers.remove(2)
# primeNumbers is [3, 5, 7, 11]
print(primeNumbers)

del primeNumbers[1]
# primeNumbers is [3, 7, 11]

primeNumbers.extend([13, 17])
# primeNumbers is [3, 7, 11, 13, 17]
print(primeNumbers)

primeNumbers.clear()
# primeNumbers is []
print(primeNumbers)
```

Arrays comparing

```
ar1 = [1, 2, 4, 3]
ar2 = [1, 2, 3, 4, 5]

diff = set(ar2) - set(ar1)
# diff is {5}
print(f'{diff = }')
```

Checking equality of lists

```python
n1 = [1, 2, 3]
n2 = [1, 2, 3]
n3 = [3, 2, 1]

equal1 = n1 == n2
# equal1 is True

equal2 = n1 == n3
# equal2 is False

equal3 = set(n1) == set(n3)
# equal3 is True

print(f"{equal1 = }")
print(f"{equal2 = }")
print(f"{equal3 = }")
```

Converting of a list

```
numbers = [1, 2, 3, 4, 5]
numbers = [x * 3 for x in numbers]
# numbers is [3, 6, 9, 12, 15]
print(f'{numbers =}')

numbers = list(map(lambda x: x*2, numbers))
# numbers is [6, 12, 18, 24, 30]
print(f'{numbers = }')
```

Dynamic lists

```
count = 5
lst_int = [0] * count
lst_int[0] = 1
# lst_int = [1, 0, 0, 0, 0]

print(f'{lst_int = }')
```

Filtering of elements

```
numbers = [1, 2, 3, 4, 5]
odd_items = [item for item in numbers if item % 2]
# odd_items is [1, 3, 5]

print(f'{odd_items = }')
```

Finding a list item

```
numbers = [2, 3, 5, 7, 11, 13, 17]
contain5 = 5 in numbers
# contain5 is True

index5 = 10 in numbers
# contain10 is False

number2 = [1, 9, 8, 3, 1, 6, 7]
containNum = number2.count(1)
# containNum is 2

print(f'{contain5 = }')
print(f'{index5 = }')
print(f'{contain10 = }')
print(f'{containNum = }')
```

Getting Min and Max values

```
numbers = [11, 2, 5, 7, 3]

minValue = min(numbers)
# minValue is 2
maxValue = max(numbers)
# max is 11

print(f"{minValue = }")
print(f"{maxValue = }")
```

Getting part of a list

```
numbers = [2, 3, 5, 7, 11]

first2 = numbers[:2]
# first2 is [2, 3]
last3 = numbers[2:]
# last3 is [5, 7, 11]

print(f"{first2 = }")
print(f"{last3 = }")
```

Getting unique values

```
numbers = [1, 3, 2, 1, 3]
unique = list(set(numbers))
# unique is [2, 3, 1]

print(f'{unique = }')
```

Iterating over an array (recursive)

```
numbers = [2, 3, 5, 7, 11, 13, 17]
string = ""
for i in reversed(numbers):
    string = string + str(i) + "; "

# string is "17; 13; 11; 7; 5; 3; 2 "

print(f"{string = }")
```

Iterating over a list

```
numbers = [2, 3, 5, 7, 11, 13, 17]
string = ""
for i in numbers:
    string = string + str(i) + "; "

# string is "2; 3; 5; 7; 11; 13; 17; "

print(f"{string = }")
```

Iterating over a list with index

```
numbers = [2, 3, 5, 7, 11, 13, 17]
string = ""
for i in range(0, len(numbers)):
    string += str(numbers[i])
    if i < (len(numbers) - 1):
        string += "; "

# string is "2; 3; 5; 7; 11; 13; 17"
print(f"{string = }")
```

List copying

```python
import copy
numbers1 = [1, 2, 3, 4, 5]

# the first method
numbers2 = list(numbers1)

# the second method
numbers3 = numbers1[:]

# the third method with deep copy
numbers4 = copy.deepcopy(numbers1)

print(f"{id(numbers1) = }")
print(f"{id(numbers2) = }")
print(f"{numbers2 = }")
print(f"{id(numbers3) = }")
print(f"{numbers3 = }")
print(f"{id(numbers4) = }")
print(f"{numbers4 = }")
```

List length

```
numbers = [1, 2, 3]
length = len(numbers)
# length is 3

print(f"{length = }")
```

List with a default value

```
value = 5
count = 3
lst = [value] * count
# array is [5, 5, 5]

print(f"{lst = }")
```

List initialization

```
# Empty array
n1 = []
n2 = list()

# Single-dimensional array
n3 = [1, 2, 3]
n4 = ["1", "2", "3"]

# Multidimensional array
n5 = [[1, 2], [3, 4, 5]]
```

List merging

```
firstNumbers = [2, 3, 5]
secondNumbers = [7, 11, 13]

allNumbers = firstNumbers + secondNumbers
# allNumbers is [2, 3, 5, 7, 11, 13]

print(f'{allNumbers = }')
```

Sorting of elements

```
numbers = [11, 2, 5, 7, 3]

numbers.sort()
# numbers is [2, 3, 5, 7, 11]
print(f'{numbers = }')

# descending
numbers.sort(reverse=True)
# numbers is [11, 7, 5, 3, 2]
print(f'{numbers = }')

lst = [['B', 3], ['A', 2], ['C', 1]]
lst.sort(key=lambda i: i[1], reverse=True)
# arr is [['B', 3], ['A', 2], ['C', 1]]
print(f'{lst = }')
```

Sum of elements

```
numbers = [2, 3, 5, 7, 11]
numbers_sum = sum(numbers)
# numbers_sum is 28

strings = ["A", "B", "C"]
strings_sum = ''.join(strings)
# strings_sum is 'ABC'

print(f"{numbers_sum = }")
print(f"{strings_sum = }")
```

every() and some() methods

```
from collections import deque

intQueue = deque()
intQueue.append(1)
intQueue.append(3)
intQueue.append(5)

first = intQueue.popleft()
# first is 1
second = intQueue.popleft()
# second is 3
third = intQueue.popleft()

print(f"{first = }")
print(f"{second = }")
print(f"{third = }")
```

Sets:

Adding and removing of elements

```
set1 = {"A", "B", "C"}
set1.add("D")
# set1 is {'C', 'D', 'A', 'B'}
print(f"{set1 = }")

set1.remove("A")
# set1 is {'C', 'B', 'D'}
print(f"{set1 = }")

set1.pop()
# set1 is {'B', 'D'}
print(f"{set1 = }")

set1.clear()
# set1 is {}
print(f"{set1 = }")
```

Converting of a set

```
set1 = {1, 2, 3}
set3 = [x * 3 for x in set1]
# set3 is [3, 6, 9]
print(f"{set3 = }")
```

Filtering of elements

```
set1 = {1, 2, 3}
oddArr = [i for i in set1 if i % 2]
# oddArr is [1, 3]

print(f'{oddArr = }')
```

Iterating over a set

```
chars = {"A", "B", "C", "D"}
s = ""
for c in chars:
    s += ("" if s == "" else "; ") + c

# s is "B; A; C; D"
print(f"{s = }")
```

Search for an element

```
chars = {"A", "B", "C", "D"}
containA = "A" in chars
# containA is True

containE = "E" in chars
# containE is False

chars2 = {"A", "B"}
containAll = chars > chars2
# containAll is True

print(f"{containA = }")
print(f"{containE = }")
print(f"{containAll = }")
```

Sets comparison

```
first = {1, 2}
second = {2, 1}
third = {1, 2, 3}

isEqual = first == second
print(f'{isEqual = }')
# isEqual is True

isIntersects = not first.isdisjoint(third)
# intersects is True
print(f'{isIntersects = }')

isSubset = third.issubset(first)
# isSubset is False
print(f'{isSubset = }')

isSubset = first.issubset(third)
# isSubset is True
print(f'{isSubset = }')
```

Sets initialization

```
int_set = {1, 2, 3}
str_set = {"one", "two", "three"}

print(f'{int_set = }')
print(f'{str_set = }')
```

Sets operations

```python
first = {1, 2, 3}
second =  {3, 4, 5}

# union
third1 = first | second
# third1 is {1, 2, 3, 4, 5}

# difference
third2 = first - second
# third2 is {1, 2}

# intersection
third3 = first & second
# third3 is {3}

# symmetric difference
third4 = first ^ second
# third4 is {1, 2, 4, 5}

print(f"{third1 = }")
print(f"{third2 = }")
print(f"{third3 = }")
print(f"{third4 = }")
```

Sorting of elements

```
chars = {"A", "B", "C", "D"}
s = "; ".join(chars)
# s is "C; B; D; A"
print(f'{s = }')

sortedChars = sorted(chars)
s = "; ".join(sortedChars)
# s is "A; B; C; D"
print(f'{s = }')
```

Stack<T> (LIFO)

```
from collections import deque

intStack = deque()
intStack.append(1)
intStack.append(3)
intStack.append(5)

first = intStack.pop()
# first is 5
second = intStack.pop()
# second is 3
third = intStack.pop()
# third is 1

print(f"{first = }")
print(f"{second = }")
print(f"{third = }")
```

Multi-threaded Operations

Multi-threaded operations refer to the ability of a program or application to execute multiple threads concurrently. Threads are independent sequences of instructions within a program that can run simultaneously, allowing for parallel execution and efficient utilization of multi-core processors.

Keywords "async" and "await"

```
import asyncio

async def async_task(name, delay):
    print(f"Task {name} started, will take {delay} seconds.")
    await asyncio.sleep(delay)
    print(f"Task {name} completed.")

async def main():
    tasks = [async_task("A", 2), async_task("B", 3),
async_task("C", 1)]
    await asyncio.gather(*tasks)

# Run the main function to execute the tasks
asyncio.run(main())

# Task A started, will take 2 seconds.
# Task B started, will take 3 seconds.
# Task C started, will take 1 seconds.
# Task C completed.
# Task A completed.
# Task B completed.
```