ZAAWANSOWANE TECHNIKI PROGRAMOWANIA

DR INŻ. MARCIN KULIK
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI
KATEDRA AUTOMATYZACJI NAPĘDÓW I ROBOTYKI

Testy jednostkowe

Testy jednostkowe w Pythonie

Testy jednostkowe:

- technika programistyczna polegająca na tworzeniu automatycznych testów sprawdzających poprawność działania pojedynczych fragmentów kodu, zwanych jednostkami. Celem testów jednostkowych jest zapewnienie, że poszczególne komponenty oprogramowania działają zgodnie z oczekiwaniami i nie powodują błędów.
- Narzędziem często używanym do tworzenia testów jednostkowych w Pythonie jest moduł **unittest**

Testy jednostkowe w Pythonie

Kluczowe cechy testów jednostkowych:

- Izolacja: Testy jednostkowe powinny być izolowane od pozostałych części kodu. Oznacza to, że testy nie powinny zależeć od innych jednostek ani od zasobów zewnętrznych, takich jak np. bazy danych.
- Powtarzalność: Testy jednostkowe powinny dawać te same wyniki za każdym razem, gdy są uruchamiane, niezależnie od środowiska wykonawczego.
- Automatyzacja: Testy jednostkowe są automatycznie wykonywane, co pozwala na częste i powtarzalne testowanie kodu bez potrzeby ręcznego interweniowania.
- Dokumentacja: Testy jednostkowe stanowią rodzaj dokumentacji, która opisuje oczekiwane zachowanie kodu. Można łatwo zrozumieć, co dana jednostka robi, patrząc na jej testy.

Testy jednostkowe w Pythonie

Przykład testu jednostkowego w Pythonie przy użyciu modułu **unittest**:

```
import unittest

# Funkcja, którą będziemy testować
def dodaj(a, b):
    return a + b

# Klasa testowa dziedzicząca po unittest.TestCase
class TestDodawania(unittest.TestCase):

    # Metoda testowa sprawdzająca poprawność dodawania
    def test_dodawania(self):
        wynik = dodaj(2, 3)
        self.assertEqual(wynik, 5)  # Sprawdź, czy wynik jest równy oczekiwanemu

if __name__ == '__main__':
    unittest.main()
```

Testy jednostkowe w Pythonie

Przykład testu jednostkowego w Pythonie przy użyciu modułu **unittest**:

```
import unittest
from scipy.integrate import trapz

# Funkcja, którą będziemy testować - obliczanie całki numerycznej
def oblicz_calka(f, a, b, n):
    x = [a + i * (b - a)/(n - 1) for i in range(n)]
    y = list(map(f, x))
    return trapz(y, x)

# Klasa testowa dziedzicząca po unittest.TestCase
class TestCalki(unittest.TestCase):

    # Test obliczania całki numerycznej dla funkcji kwadratowej
    def test_calka_dla_funkcji_kwadratowej(self):
        a, b, n = 0, 1, 1000
        f = lambda x: x**2
        wynik = oblicz_calka(f, a, b, n)
        self.assertAlmostEqual(wynik, 1/3, places=5)

if __name__ == '__main__':
    unittest.main()
```

Testy jednostkowe w Pythonie

Przykłady innych metod sprawdzających:

- **assertGreater** i **assertLess**:

```
self.assertGreater(10, 5)
self.assertLess(5, 10)
```

- **assertTrue** i **assertFalse**:

```
self.assertTrue(5 < 10)
self.assertFalse(10 < 5)
```

- **assertIn** i **assertNotIn**:

```
self.assertIn(2, [1, 2, 3])
self.assertNotIn(4, [1, 2, 3])
```

- **AssertRaises**:

```
def dzielenie(a, b):
    return a / b

with self.assertRaises(ZeroDivisionError):
    dzielenie(1, 0)
```

Testy jednostkowe w Pythonie

Przykłady innych metod sprawdzających:

- **assertIs** i **assertIsNot:**

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]

self.assertIs(x, y)
self.assertIsNot(x, z)
```

- **assertIsNone** i **assertIsNotNone:**

```
value = None

self.assertIsNone(value)

value = 42
self.assertIsNotNone(value)
```

Wzorce projektowe

Wzorce projektowe

Wzorce projektowe to powtarzające się rozwiązania dla powszechnych problemów związanych z projektowaniem oprogramowania. Wzorce reprezentują najlepsze praktyki i ewoluowały w czasie, w miarę jak doświadczeni programiści znaleźli wydajne i efektywne sposoby rozwiązywania określonych wyzwań projektowych. W Pythonie, podobnie jak w innych językach programowania zorientowanych obiektowo, wzorce projektowe znacznie poprawiają organizację kodu, jego czytelność, oraz możliwość ponownego wykorzystania.

Wzorce projektowe

Wzorzec Singleton (Singleton Pattern):

- Ten wzorzec zapewnia, że klasa ma tylko jedną instancję i udostępnia globalny dostęp do niej.

W Pythonie można to zrealizować za pomocą zmiennej klasowej do przechowywania pojedynczej instancji, a konstruktor zapewnia, że zawsze zostanie zwrócona ta sama instancja.

Wzorce projektowe

```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance

obj1 = Singleton()
obj2 = Singleton()
print(obj1 is obj2)  # True
```

Wzorce projektowe

Wzorzec Fabryki (Factory Pattern):

- Wzorzec fabryki definiuje interfejs do tworzenia obiektów, ale pozwala podklasom zmieniać rodzaj tworzonych obiektów. W Pythonie fabryki mogą być implementowane jako klasy lub metody, które produkują obiekty różnych klas pochodnych na podstawie określonego parametru.

Wzorce projektowe

```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

def get_pet(pet="dog"):
    pets = dict(dog=Dog(), cat=Cat())
    return pets[pet]

pet = get_pet("dog")
print(pet.speak())  # Woof!
```

Wzorce projektowe

Wzorzec Dekorator (Decorator Pattern):

- Wzorzec dekoratora umożliwia dynamiczne dołączanie dodatkowych funkcji do obiektu bez zmieniania jego struktury. Można to osiągnąć w Pythonie, tworząc klasy dekoratora, które dodają funkcjonalność do obiektu bazowego poprzez kompozycję.

Wzorce projektowe

```
class Coffee:
    def cost(self):
        return 5

class MilkDecorator:
    def __init__(self, coffee):
        self._coffee = coffee

    def cost(self):
        return self._coffee.cost() + 2

coffee = Coffee()
coffee_with_milk = MilkDecorator(coffee)

print(f"Kawa kosztuje: ${coffee.cost()}")
print(f"Kawa z mlekiem kosztuje: ${coffee_with_milk.cost()}")
```

Wzorce projektowe

Wzorzec Obserwator (Observer Pattern):

- Ten wzorzec definiuje zależności między obiektami. Gdy jeden obiekt zmienia stan, wszystkie jego zależności są automatycznie informowane i aktualizowane. W Pythonie można to zrealizować, tworząc klasy podmiotu i obserwatora, gdzie obserwatorzy rejestrują się u podmiotu, aby otrzymywać aktualizacje, gdy zmienia się stan podmiotu.

Wzorce projektowe

```
class Subject:
    def __init__(self):
        self._observers = []

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

    def notify(self, message):
        for observer in self._observers:
            observer.update(message)

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

    def update(self, message):
        print(f"{self.name} otrzymał wiadomość: {message}")
```

```
subject = Subject()
observer1 = Observer("Obserwator 1")
observer2 = Observer("Obserwator 2")

subject.attach(observer1)
subject.attach(observer2)

subject.notify("Wydarzenie A")
# Wynik:
# Obserwator 1 otrzymał wiadomość: Wydarzenie A
# Obserwator 2 otrzymał wiadomość: Wydarzenie A
```

Wzorce projektowe

Wzorzec Strategii (Strategy Pattern):

- Wzorzec strategii definiuje rodzinę algorytmów, hermetyzuje każdy z nich i sprawia, że są one wymienne. Python umożliwia zdefiniowanie zestawu algorytmów jako oddzielne klasy i dynamiczne przełączanie między nimi w klasie kontekstu.

Wzorce projektowe

```
class SortStrategy:
    def sort(self, data):
        pass

class QuickSort(SortStrategy):
    def sort(self, data):
        return sorted(data)

class BubbleSort(SortStrategy):
    def sort(self, data):
        n = len(data)
        for i in range(n - 1):
            for j in range(0, n - i - 1):
                if data[j] > data[j + 1]:
                    data[j], data[j + 1] = data[j + 1], data[j]
        return data
```

```
class Sorter:
    def __init__(self, strategy):
        self._strategy = strategy

    def sort(self, data):
        return self._strategy.sort(data)

data = [7, 2, 5, 1, 8]
quick_sorter = Sorter(QuickSort())
bubble_sorter = Sorter(BubbleSort())

print("Quick Sort:", quick_sorter.sort(data))
print("Bubble Sort:", bubble_sorter.sort(data))
```

Wzorce projektowe

Wzorzec Polecenia (Command Pattern):

- Wzorzec polecenia hermetyzuje żądanie jako obiekt, umożliwiając jego parametryzację i kolejkowanie. W Pythonie można to osiągnąć, tworząc obiekty polecenia, które hermetyzują akcję i jej parametry.

Wzorce projektowe

```
class Light:
    def turn_on(self):
        print("Light is on")

    def turn_off(self):
        print("Light is off")

class LightOnCommand:
    def __init__(self, light):
        self._light = light

    def execute(self):
        self._light.turn_on()

class LightOffCommand:
    def __init__(self, light):
        self._light = light

    def execute(self):
        self._light.turn_off()
```

```
class RemoteControl:
    def __init__(self):
        self._commands = []

    def add_command(self, command):
        self._commands.append(command)

    def press_button(self, button_index):
        if 0 <= button_index < len(self._commands):
            self._commands[button_index].execute()

light = Light()
remote = RemoteControl()

remote.add_command(LightOnCommand(light))
remote.add_command(LightOffCommand(light))

remote.press_button(0)  # Turns the light on
remote.press_button(1)  # Turns the light off
```

ZAAWANSOWANE TECHNIKI PROGRAMOWANIA

DR INŻ. MARCIN KULIK
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI
KATEDRA AUTOMATYZACJI NAPĘDÓW I ROBOTYKI

Wzorce projektowe

Wzorzec Stanowy (State Pattern):

- Pozwala obiektowi zmienić swoje zachowanie, gdy zmieni się jego stan wewnętrzny. Obiekt wydaje się zmieniać swoją klasę. Należy zdefiniować klasy stanów, które hermetyzują różne zachowania i pozwalają obiektowi kontekstowemu przełączać się między tymi stanami.

Wzorce projektowe

```
class State:
    def do_action(self, context):
        pass

class StartState(State):
    def do_action(self, context):
        print("Player is in start state.")
        context.set_state(self)

class StopState(State):
    def do_action(self, context):
        print("Player is in stop state.")
        context.set_state(self)
```

```
class Context:
    def __init__(self):
        self._state = None

    def set_state(self, state):
        self._state = state

    def get_state(self):
        return self._state

context = Context()

start_state = StartState()
start_state.do_action(context)
print(context.get_state())

stop_state = StopState()
stop_state.do_action(context)
print(context.get_state())
```

Wzorce projektowe

Wzorzec Kompozytowy (Composite Pattern):

- Komponowanie obiektów w struktury drzewiaste w celu reprezentowania hierarchii część-całość.
- Jest używany do tworzenia hierarchii obiektów, gdzie zarówno pojedyncze obiekty, jak i ich złożone grupy (kompozyty) są traktowane w taki sam sposób.
- Implementacja: Stworzenie interfejsu komponentu, który implementują zarówno klasy liści, jak i kompozytów, umożliwiając klientom pracę z pojedynczymi obiektami i kompozycjami.

Wzorce projektowe

```
class Graphic:
    def draw(self):
        pass
class Circle(Graphic):
    def draw(self):
        print("Drawing a circle")
class Square(Graphic):
    def draw(self):
        print("Drawing a square")
class Picture(Graphic):
    def __init__(self):
        self._graphics = []
    def add(self, graphic):
        self._graphics.append(graphic)
    def draw(self):
        print("Drawing a picture:")
        for graphic in self._graphics:
            graphic.draw()
```

```
circle = Circle()
square = Square()

picture = Picture()
picture.add(circle)
picture.add(square)

picture.draw()
```

Wzorce projektowe

Wzorzec Pełnomocnika (Proxy Pattern):

- Strukturalny wzorzec projektowy, który pozwala na kontrolowane dostarczanie dostępu do innych obiektów poprzez dostarczenie alternatywnej reprezentacji (proksy) tych obiektów. Proksy działają jako pośrednicy między klientem a rzeczywistym obiektem, co pozwala na dodatkową kontrolę nad dostępem do obiektów i umożliwia wykonywanie dodatkowych działań w trakcie dostępu lub manipulacji danymi.

Wzorce projektowe

Wzorzec Pełnomocnika (Proxy Pattern):

Wzorzec Proxy jest szczególnie przydatny w sytuacjach, gdzie:

- Potrzebujemy kontroli dostępu do pewnych zasobów, takich jak pliki, usługi sieciowe, duże obrazy czy kosztowne operacje.
- Chcemy odroczyć ładowanie i inicjalizację kosztownych obiektów, aż będą one rzeczywiście potrzebne (leniwe ładowanie).
- Musimy zaimplementować mechanizmy bezpieczeństwa, takie jak sprawdzanie uprawnień dostępu.

Wzorce projektowe

```
class RealSubject:
    def request(self):
        print("RealSubject: Obsługuje żądanie.")
class Proxy:
    def __init__(self, real_subject):
        self._real_subject = real_subject
    def request(self):
        if self.check_access():
            self._real_subject.request()
    def check_access(self):
        print("Proxy: Sprawdzam dostęp.")
        return True  # W rzeczywistym zastosowaniu sprawdzalibyśmy dostęp klienta

# Użycie wzorca Proxy
real_subject = RealSubject()
proxy = Proxy(real_subject)
proxy.request()
```

Wzorce projektowe

Wzorce projektowe MVC (Model-View-Controller) i MVP (Model-View-Presenter)

- MVC i MVP to wzorce architektoniczne, które dzielą aplikację na warstwy lub komponenty odpowiedzialne za różne aspekty funkcjonowania aplikacji, takie jak logika biznesowa, logika prezentacji oraz kontrola przepływu danych.
- Separacja odpowiedzialności: Oddzielenie logiki aplikacji od prezentacji sprawia, że kod staje się bardziej modularny i łatwiejszy do zarządzania.
- Ułatwienie testowania: Logikę aplikacji można testować niezależnie od interfejsu użytkownika.
- Łatwiejsze utrzymanie: Zmiany w logice lub interfejsie użytkownika można wprowadzać niezależnie od siebie.

Wzorce projektowe

Wzorzec Model-Widok-Kontroler (MVC (Model-View-Controller) Pattern):

- wzorzec projektowania powszechnie stosowany w rozwoju oprogramowania, który pozwala na rozdzielenie zadań w aplikacji na trzy wzajemnie powiązane komponenty: Model, Widok i Kontroler. Ten wzorzec pomaga w zachowaniu organizacji kodu, jego ponownego wykorzystywania i łatwości utrzymania. Chociaż MVC jest zazwyczaj kojarzone z frameworkami do tworzenia stron internetowych, takimi jak Django i Flask, można również zaimplementować uproszczoną wersję MVC w języku Python bez konieczności korzystania z konkretnego frameworka.

Wzorce projektowe

Wzorzec Model-Widok-Kontroler (MVC (Model-View-Controller) Pattern):

- **Model**: Model reprezentuje dane aplikacji i logikę biznesową. Odpowiada za zarządzanie danymi oraz regułami określającymi, w jaki sposób dane mogą być zmieniane. W aplikacji napisanej w języku Python może to być zestaw klas, funkcji lub struktur danych, które zarządzają i manipulują danymi.

```
class UserModel:
    def __init__(self, name, email):
        self.name = name
        self.email = email
class UserController:
    def create_user(self, name, email):
        user = UserModel(name, email)
        return user

user_controller = UserController()
user = user_controller.create_user("John", "john@example.com")
```

Wzorce projektowe

Wzorzec Model-Widok-Kontroler (MVC (Model-View-Controller) Pattern):

- **Widok**: Widok jest odpowiedzialny za prezentowanie danych użytkownikowi oraz obsługę interakcji interfejsu użytkownika. Widok pobiera dane z modelu i wyświetla je w odpowiednim formacie. W aplikacji napisanej w języku Python Widok można zaimplementować przy użyciu bibliotek do tworzenia interfejsów graficznych (GUI), takich jak Tkinter, lub frameworków webowych, na przykład Flask do renderowania szablonów HTML.

Wzorce projektowe

Wzorzec Model-Widok-Kontroler (MVC (Model-View-Controller) Pattern):

- **Widok**:

```
# Przykład użycia Tkinter do tworzenia prostego GUI
import tkinter as tk

class UserView:
    def __init__(self, root):
        self.root = root
        self.label = tk.Label(self.root, text="Informacje o użytkowniku:")
        self.label.pack()
        self.user_info_label = tk.Label(self.root, text="")
        self.user_info_label.pack()

root = tk.Tk()
user_view = UserView(root)
root.mainloop()
```

Wzorce projektowe

Wzorzec Model-Widok-Kontroler (MVC (Model-View-Controller) Pattern):

- **Kontroler**: Kontroler działa jako pośrednik między Modelem a Widokiem. Otrzymuje dane od użytkownika za pośrednictwem Widoku, przetwarza je, a następnie współdziała z Modelem w celu aktualizacji danych lub pobrania danych do wyświetlenia.

Wzorce projektowe

Wzorzec Model-Widok-Kontroler (MVC (Model-View-Controller) Pattern):

- **Kontroler**:

```
class UserController:
    def __init__(self, user_view):
        self.user_view = user_view
        self.model = None

    def create_user(self, name, email):
        self.model = UserModel(name, email)
        self.update_view()

    def update_view(self):
        if self.model:
            user_info = f"Imię: {self.model.name}, Email: {self.model.email}"
            self.user_view.user_info_label.config(text=user_info)

user_view = UserView(root)
user_controller = UserController(user_view)
```

Wzorce projektowe

```
class Model:
    def __init__(self):
        self._data = []

    def get_data(self):
        return self._data

    def set_data(self, data):
        self._data = data

class View:
    def show_data(self, data):
        print(f"Data: {data}")
```

```
class Controller:
    def __init__(self, model, view):
        self._model = model
        self._view = view

    def update_view(self):
        data = self._model.get_data()
        self._view.show_data(data)

model = Model()
view = View()
controller = Controller(model, view)

data = [1, 2, 3, 4, 5]
model.set_data(data)
controller.update_view()
```

Wzorce projektowe

Wzorzec Model-Widok-Prezenter (MVP (Model-View-Presenter) Pattern):

- wzorzec projektowy często wykorzystywany w programowaniu do tworzenia interfejsów użytkownika. W języku Python można zaimplementować MVP, rozdzielając logikę aplikacji na trzy odrębne komponenty: Model (Model), Widok (View) i Prezenter (Presenter).

Wzorce projektowe

Wzorzec Model-Widok-Prezenter (MVP (Model-View-Presenter) Pattern):

Model (Model):

- Model reprezentuje dane i logikę biznesową aplikacji.
- Powinien być odpowiedzialny za manipulację danymi, ich przechowywanie i pobieranie.
- Tutaj definiowane są struktury danych i komunikacje z bazami danych lub interfejsami API.

Widok (View):

- Widok jest odpowiedzialny za interfejs użytkownika i prezentację.
- W przypadku aplikacji desktopowej może to być interfejs graficzny (GUI).
- W aplikacji internetowej może to być strona internetowa lub jej część.
- W MVP jest mniej samodzielny niż w MVC. Widok jest bardziej zależny od prezentera.

Wzorce projektowe

Wzorzec Model-Widok-Prezenter (MVP (Model-View-Presenter) Pattern):

Prezenter (Presenter):

- Prezenter działa jako pośrednik między Modelem a Widokiem. W odróżnieniu od MVC, to prezenter bezpośrednio manipuluje widokiem i modelem
- Obsługuje wejście użytkownika, aktualizuje Widok i komunikuje się z Modelem w celu pobierania lub aktualizacji danych.
- Prezenter zawiera logikę aplikacji i reguły biznesowe. Odpowiada za przetwarzanie danych i ich formatowanie przed wyświetleniem oraz za bezpośrednią manipulację widokiem.

Wzorce projektowe

```
# Model
class CalculatorModel:
    def add(self, num1, num2):
        return num1 + num2

# View
class CalculatorView:
    def get_user_input(self):
        return float(input("Enter a number: "))

    def display_result(self, result):
        print(f"Result: {result}")
```

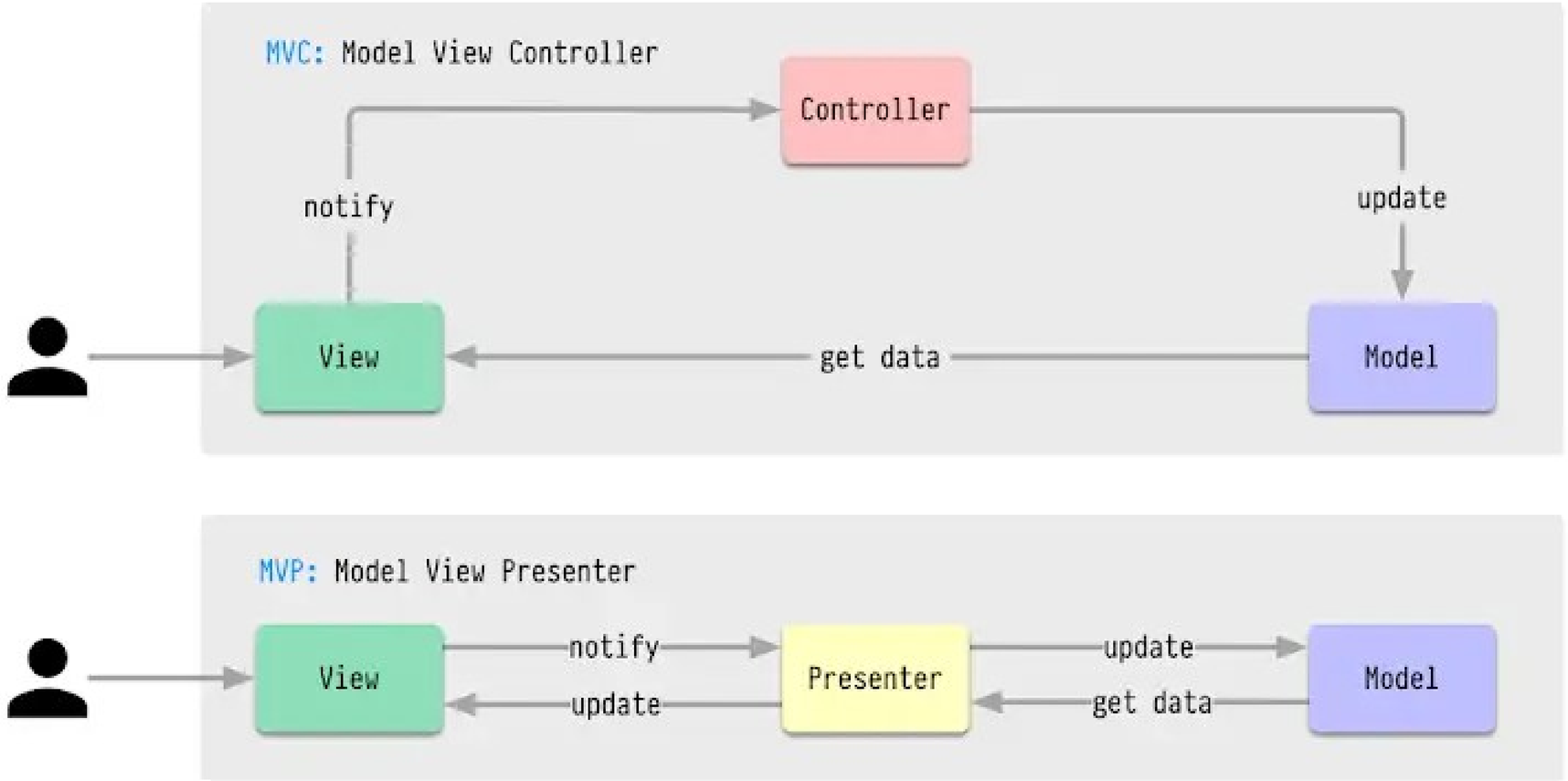
```
# Presenter
class CalculatorPresenter:
    def __init__(self, model, view):
        self.model = model
        self.view = view

    def run(self):
        num1 = self.view.get_user_input()
        num2 = self.view.get_user_input()
        result = self.model.add(num1, num2)
        self.view.display_result(result)

if __name__ == "__main__":
    model = CalculatorModel()
    view = CalculatorView()
    presenter = CalculatorPresenter(model, view)
    presenter.run()
```

Wzorce projektowe

Różnice między MVC a MVP



Różnice między MVC a MVP

Właściwość	MVC	MVP
Odpowiedzialność widoku	Więcej logiki prezentacyjnej	Minimalna logika prezentacyjna
Odpowiedzialność kontrolera/prezentera	Odpowiada za interakcje i kontrolę przepływu	Odpowiada za logikę prezentacji
Interakcja z widokiem	Kontroler reaguje na zmiany w widoku	Prezenter sam kontroluje widok
Testowanie	Może być trudniejsze	Łatwiejsze dzięki wyraźnemu oddzieleniu prezentera od widoku

Wzorce projektowe

Wzorzec Model-Widok-ModelWidoku (MVVM (Model-View-ViewModel) Pattern):

- wzorzec projektowania powszechnie stosowany w rozwoju oprogramowania, zwłaszcza w kontekście tworzenia interfejsów graficznych (GUI). Choć MVVM jest często kojarzony z językami takimi jak C# i frameworkami takimi jak WPF czy Xamarin, można zastosować zasady MVVM również w Pythonie, zwłaszcza pracując z bibliotekami GUI takimi jak PyQt czy Kivy.

Wzorce projektowe

Wzorzec Model-Widok-ModelWidoku (MVVM (Model-View-ViewModel) Pattern):

- **Model**: Model reprezentuje dane aplikacji i logikę. Powinien być niezależny od interfejsu użytkownika. W Pythonie można tworzyć klasy modelu za pomocą zwykłych klas Pythona lub struktur danych.

```
class Task:
    def __init__(self, name, description, completed=False):
        self.name = name
        self.description = description
        self.completed = completed
```

Wzorce projektowe

Wzorzec Model-Widok-ModelWidoku (MVVM (Model-View-ViewModel) Pattern):

- **View**: Widok jest odpowiedzialny za wyświetlanie danych użytkownikowi i przechwytywanie danych wejściowych od użytkownika. W Pythonie można użyć biblioteki GUI, takiej jak PyQt czy Kivy, do tworzenia widoków (zapewniają pewien poziom reaktywności). Definiujemy interfejs użytkownika przy użyciu widżetów i układów biblioteki.

Wzorzec Model-Widok-ModelWidoku (MVVM (Model-View-ViewModel) Pattern):

- **View**:

```
# Przykład korzystania z PyQt
from PyQt5.QtWidgets import QWidget, QVBoxLayout, QLabel, QPushButton, QLineEdit

class TaskView(QWidget):
    def __init__(self):
        super().__init__()
        self.layout = QVBoxLayout()
        self.label = QLabel("Nazwa zadania:")
        self.name_input = QLineEdit()
        self.save_button = QPushButton("Zapisz")
        self.layout.addWidget(self.label)
        self.layout.addWidget(self.name_input)
        self.layout.addWidget(self.save_button)
        self.setLayout(self.layout)
```

Wzorce projektowe

Wzorzec Model-Widok-ModelWidoku (MVVM (Model-View-ViewModel) Pattern):

- **ViewModel**: ViewModel jest odpowiedzialny za udostępnianie danych z modelu widokowi i obsługę interakcji użytkownika. Działa jako most między modelem a widokiem. Widok i ViewModel są połączone mechanizmem "data binding" (wiązań danych), który umożliwia automatyczną aktualizację widoku, gdy dane w ViewModelu się zmieniają.

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

    def pobierz_nazwe_zadania(self):
        return self.model.name

    def ustaw_nazwe_zadania(self, nazwa):
        self.model.name = nazwa
```

Wzorce projektowe

Wzorzec Model-Widok-ModelWidoku (MVVM (Model-View-ViewModel) Pattern):

- **Wiązanie danych**: W niektórych implementacjach MVVM można używać wiązania danych do automatycznego synchronizowania danych między ViewModel a Widokiem. Jednak jest to bardziej powszechne w językach i frameworkach zaprojektowanych z myślą o MVVM. W Pythonie może być konieczne ręczne połączenie danych i zdarzeń między ViewModel a Widokiem.

```
# Wewnątrz klasy Widoku PyQt
def ustaw_view_model(self, view_model):
    self.view_model = view_model
    self.name_input.textChanged.connect(self.view_model.ustaw_nazwe_zadania)
    self.save_button.clicked.connect(self.zapisz_zadanie)

def zapisz_zadanie(self):
    # Zapisz zadanie za pomocą ViewModel i zaktualizuj Widok według potrzeb
    # Przykład: self.view_model.zapisz_zadanie()
```

Wzorce projektowe

```
# Importuj odpowiednie moduły
import sys
from PyQt5.QtWidgets import QApplication

# Tworzenie instancji modelu (Task)
task_model = Task("Zadanie domowe", "Przygotować referat")

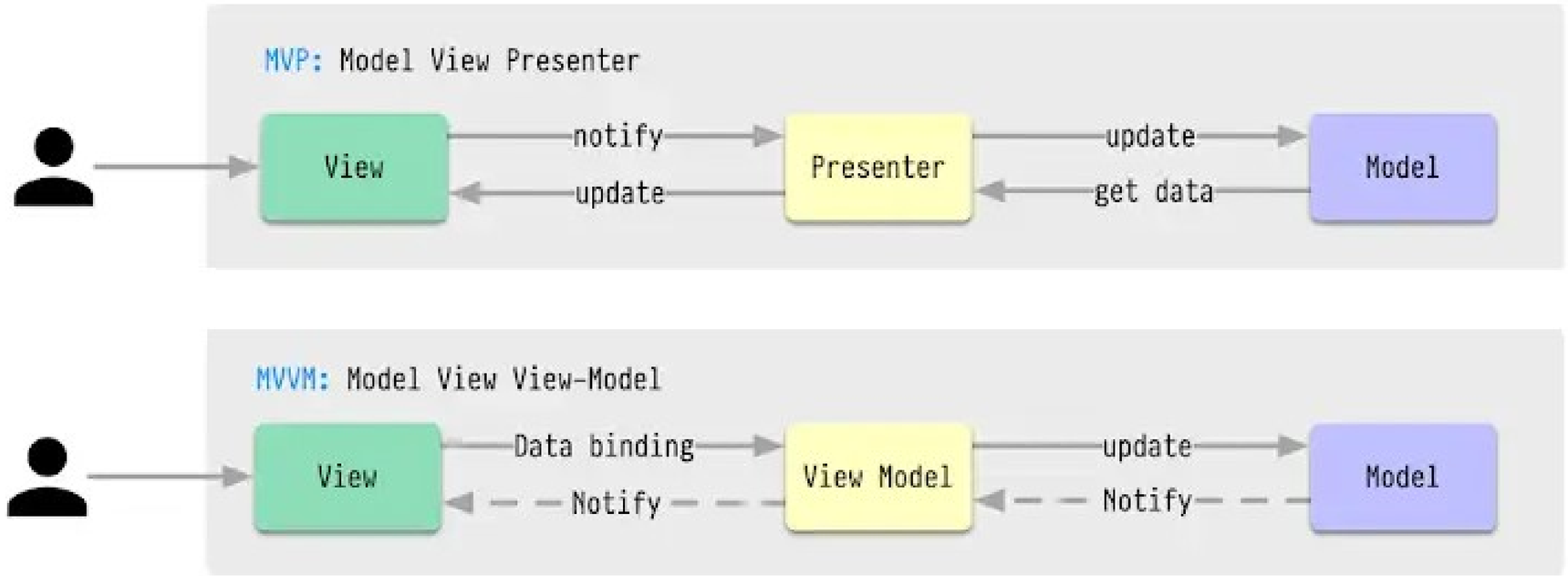
# Tworzenie instancji ViewModel (TaskViewModel)
task_view_model = TaskViewModel(task_model)

# Tworzenie instancji widoku (TaskView)
app = QApplication(sys.argv)
task_view = TaskView()
task_view.set_view_model(task_view_model)
task_view.show()

# Uruchom aplikację
sys.exit(app.exec_())
```

Wzorce projektowe

Różnice między MVP a MVVM



Różnice między MVP a MVVM

Właściwość	MVP	MVVM
Sposób komunikacji	Prezenter bezpośrednio manipuluje widokiem	Widok i ViewModel komunikują się przez wiązania danych
Zależność widoku	Widok jest zależny od prezentera	Widok jest reaktywnie powiązany z ViewModelem
Automatyczna aktualizacja widoku	Ręczna aktualizacja przez prezenter	Automatyczna dzięki wiązaniom danych
Testowanie	Testowanie prezenterów jest łatwiejsze	Testowanie ViewModeli jest łatwe, zwłaszcza w reaktywnym środowisku
Środowisko	Popularny w aplikacjach desktopowych i mobilnych, które nie wspierają data binding	Popularny w środowiskach z obsługą data binding, np. Angular, WPF, Vue, React

ZAAWANSOWANE TECHNIKI PROGRAMOWANIA

DR INŻ. MARCIN KULIK
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI
KATEDRA AUTOMATYZACJI NAPĘDÓW I ROBOTYKI

Wielozadaniowość

Wielozadaniowość

Wielozadaniowość (multitasking):

- Wielozadaniowość odnosi się do zdolności systemu do wykonywania wielu zadań jednocześnie lub w krótkich odstępach czasu. W przypadku systemów operacyjnych, oznacza to, że wiele programów lub procesów może być aktywnych jednocześnie. W kontekście programowania wielozadaniowość obejmuje również równoczesne wykonywanie wielu operacji w obrębie jednej aplikacji.

Wielozadaniowość

Python oferuje różne narzędzia do pracy z wielowątkowością oraz współbieżnością, co pozwala na efektywne zarządzanie zadaniami równoległymi. W tej grupie bibliotek znajdziemy moduły, które umożliwiają tworzenie wątków, procesów, kolejek do wymiany danych między nimi, planowanie zadań oraz zarządzanie sygnałami. Wielowątkowość i współbieżność są kluczowe w aplikacjach, które muszą wykonywać wiele zadań jednocześnie, takich jak serwery lub aplikacje obliczeniowe.

Z uwagi na GIL (Global Interpreter Lock) w Pythonie wykonuje się tylko jeden wątek. Mimo tego, wielowątkowość jest w Pythonie użyteczna np. w zastosowaniach sieciowych. W wersji Pythona 3.13 można korzystać z pełnoprawnej wielowątkowości (na razie testowo)

Wielozadaniowość

Przykłady multitaskingu:

- **Interaktywne Aplikacje**: Wielozadaniowość jest szczególnie ważna w interaktywnych aplikacjach, gdzie użytkownicy mogą jednocześnie wykonywać różne czynności. Przykłady to edytory tekstowe, przeglądarki internetowe czy aplikacje biurowe.
- **Systemy Operacyjne**: W systemach operacyjnych wielozadaniowość pozwala na równoczesne wykonywanie wielu programów, co przekłada się na płynne użytkowanie. Każdy program działa w swoim własnym procesie lub wątku.
- **Serwery Aplikacji**: W środowiskach serwerowych wielozadaniowość pozwala obsługiwać jednocześnie wiele żądań od klientów. Serwery HTTP, bazodanowe czy serwisy API to przykłady, gdzie równoczesna obsługa wielu klientów jest kluczowa.

Wielozadaniowość

Jednozadaniowość vs Wielozadaniowość:

- **Jednozadaniowość**: W jednozadaniowym podejściu system lub aplikacja wykonuje jedno zadanie na raz. Po zakończeniu jednego zadania przechodzi do następnego.
- **Wielozadaniowość**: W wielozadaniowym środowisku wiele zadań jest wykonywanych równocześnie lub w krótkich odstępach czasu. To podejście umożliwia lepsze wykorzystanie zasobów systemowych.

Wielozadaniowość

Proces (process) i Wątek (thread):

- **Proces** to program w trakcie wykonywania. Każdy proces posiada swoją własną przestrzeń pamięci, zasoby oraz środowisko wykonawcze. Procesy działają niezależnie od siebie, co oznacza, że nie współdzielą ze sobą pamięci ani innych zasobów.
- **Wątek** to najmniejsza jednostka wykonywania w systemie operacyjnym. Wielowątkowość odnosi się do możliwości jednoczesnego wykonywania wielu wątków w obrębie jednego procesu. Wątki współdzielą zasoby procesu, takie jak pamięć, ale posiadają własne liczniki instrukcji, rejestr stanu i stos wywołań.

Różnice między wątkiem a procesem

Wątek:

- Działa w obrębie procesu.
- Współdzieli zasoby z innymi wątkami w ramach tego samego procesu.
- Mniej kosztowny w zasoby niż procesy.
- Bardziej efektywny w komunikacji między wątkami.

Proces:

- Niezależna jednostka wykonawcza.
- Posiada własną przestrzeń pamięci.
- Bardziej kosztowny w zasoby niż wątki.
- Wymaga mechanizmów komunikacji międzyprocesowej (IPC) do współpracy z innymi procesami.

Multithreading

Zalety i wady wielowątkowości:

Zalety:

- Wydajność: Wielowątkowość może znacznie przyspieszyć operacje równoległe, zwłaszcza w przypadku operacji wejścia/wyjścia (I/O-bound) czy obliczeń równoległych.
- Łatwość komunikacji: Wątki mogą łatwo komunikować się między sobą, ponieważ współdzielą tę samą przestrzeń pamięci.

Multithreading

Zalety i wady wielowątkowości:

Wady:

- Bezpieczeństwo: Współdzielenie zasobów między wątkami może prowadzić do problemów bezpieczeństwa, takich jak race conditions czy deadlocki.
- Trudności w debugowaniu: Błędy związane z wielowątkowością mogą być trudne do zidentyfikowania i naprawienia.

Multithreading

Deadlocki:

- Sytuacje, w których wątki wzajemnie blokują się, czekając na siebie nawzajem.

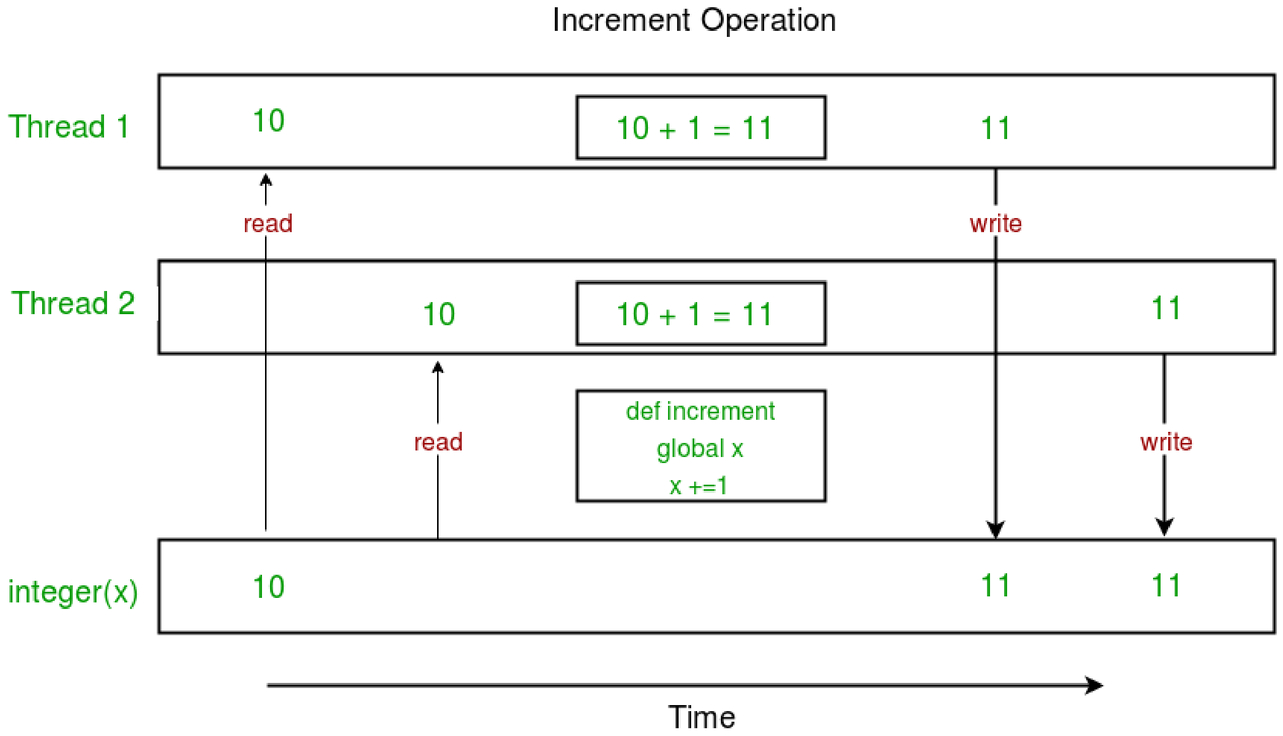
Race conditions:

- Problemy związane z równoczesnym dostępem do współdzielonych zasobów, prowadzące do nieprzewidywalnych wyników.

Multithreading

Race conditions:

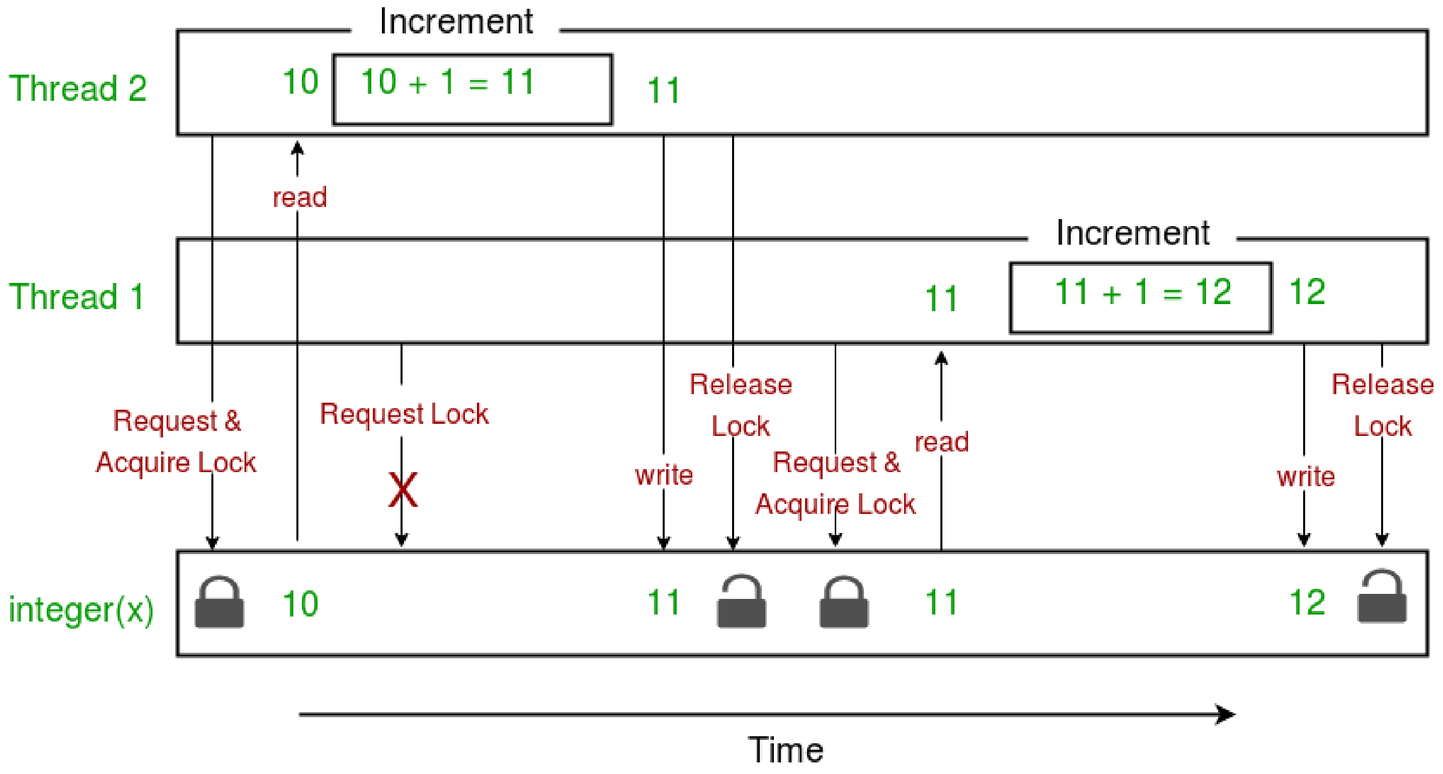
- Problemy związane z równoczesnym dostępem do współdzielonych zasobów, prowadzące do nieprzewidywalnych wyników.



Multithreading

Race conditions:

- Problemy związane z równoczesnym dostępem do współdzielonych zasobów, prowadzące do nieprzewidywalnych wyników.



Multithreading

Przykłady zastosowań wielowątkowości:

Interaktywne Aplikacje:

- Wielowątkowość umożliwia płynne reagowanie na interakcje użytkownika w czasie rzeczywistym, jednocześnie wykonując inne operacje w tle.

Operacje Wejścia/Wyjścia (I/O-bound):

- Wielowątkowość jest skuteczna w operacjach, gdzie większość czasu jest poświęcona na oczekiwanie na dane (np. operacje sieciowe, operacje na dysku).

Multithreading

Global Interpreter Lock (GIL):

- GIL to Global Interpreter Lock, który jest mechanizmem stosowanym w interpreterze CPython (dominującym interpreterze Pythona). GIL ma na celu zabezpieczenie dostępu do obiektów Pythona przed równoczesnymi modyfikacjami przez wątki.
- GIL sprawia, że tylko jeden wątek może wykonywać kod Pythona w danym czasie, nawet jeśli na maszynie są dostępne wielordzeniowe procesory. Wynika to z tego, że GIL blokuje dostęp do obiektów Pythona podczas wykonywania kodu.

Multithreading

Problemy z GIL:

- Ograniczenie Wielowątkowości: GIL stanowi ograniczenie wydajności dla programów, które intensywnie korzystają z wielowątkowości. Wątki w Pythonie często nie mogą efektywnie wykorzystać dostępnych zasobów.
- Wydajność w Operacjach CPU-bound: W zadaniach CPU-bound, gdzie wykonywane są intensywne obliczenia, GIL może prowadzić do spadku wydajności, ponieważ tylko jeden wątek może pracować w danym momencie.

Multithreading

Alternatywne podejścia:

- Używanie wielu procesów: Zamiast wielowątkowości, można skorzystać z wieloprocesowości, gdzie każdy proces działa w odrębnym interpreterze Pythona i jest niezależny od GIL.
- Użycie innych interpreterów: Istnieją inne implementacje Pythona, takie jak Jython czy IronPython, które nie stosują GIL. Mogą być używane, jeśli specyfika projektu pozwala na użycie innej implementacji Pythona.
- W wersji Pythona 3.13 można korzystać z pełnoprawnej wielowątkowości (na razie testowo)

Multiprocessing

Zalety i wady wieloprocesowości:

Zalety:

- Izolacja błędów: Problemy w jednym procesie nie wpływają na pozostałe procesy.
- Stabilność: W razie awarii jednego procesu, pozostałe mogą nadal działać.

Wady:

- Koszt zasobów: Procesy wymagają więcej zasobów niż wątki, co może prowadzić do większego zużycia pamięci i obciążenia systemu.

Multiprocessing

Przykłady zastosowań wieloprocesowości:

Stabilność Systemów:

- Wieloprocesowość jest stosowana w systemach, gdzie stabilność jest kluczowa. Awaria jednego procesu nie powinna wpływać na pozostałe.

Obliczenia Równoległe:

- Wykonywanie intensywnych obliczeń równoległych, wykorzystując dostępne rdzenie procesora.

Izolacja Aplikacji:

- Każda aplikacja może działać jako odrębny proces, co zapewnia izolację i ochronę przed awariami.

Wielozadaniowość - wbudowane moduły Pythona

threading – Wielowątkowość

Moduł **threading** pozwala na tworzenie i zarządzanie wątkami w Pythonie. Dzięki wątkom możemy wykonywać wiele operacji równocześnie w tej samej aplikacji, co może znacząco zwiększyć jej wydajność w przypadku operacji I/O.

Przykładowe funkcje:

threading.Thread() – Tworzenie nowego wątku.

threading.Lock() – Używanie blokad (mutexów) do synchronizacji wątków.

threading.Event() – Komunikacja między wątkami poprzez sygnały.

Wielozadaniowość - wbudowane moduły Pythona

multiprocessing – Wieloprocesowość

Moduł **multiprocessing** pozwala na tworzenie nowych procesów, które mogą wykonywać zadania niezależnie od głównego procesu programu. Procesy mają własną pamięć i mogą pracować równolegle na różnych rdzeniach procesora, co zwiększa wydajność w zadaniach obliczeniowych.

Przykładowe funkcje:

multiprocessing.Process() – Tworzenie nowego procesu.

multiprocessing.Queue() – Wymiana danych między procesami.

multiprocessing.Pool() – Zarządzanie pulą procesów.

Wielozadaniowość - wbudowane moduły Pythona

concurrent.futures – Wysokopoziomowa współbieżność

Moduł **concurrent.futures** umożliwia zarządzanie zadaniami współbieżnymi na wyższym poziomie abstrakcji, przy pomocy wątków (ThreadPoolExecutor) lub procesów (ProcessPoolExecutor). Dzięki niemu możemy łatwo uruchamiać wiele zadań równocześnie i uzyskiwać ich wyniki.

Przykładowe funkcje:

ThreadPoolExecutor() – Uruchamianie zadań w wątkach.

ProcessPoolExecutor() – Uruchamianie zadań w procesach.

submit() – Wysyłanie zadań do wykonania.

Wielozadaniowość - wbudowane moduły Pythona

queue – Kolejki

Moduł **queue** dostarcza klasy do zarządzania kolejkami (FIFO, LIFO, priorytetowe), które mogą być używane do bezpiecznej wymiany danych między wątkami. Jest to przydatne, gdy różne wątki lub procesy muszą komunikować się ze sobą.

Przykładowe funkcje:

queue.Queue() – Tworzy kolejkę FIFO.

queue.LifoQueue() – Tworzy kolejkę LIFO (stos).

queue.PriorityQueue() – Tworzy kolejkę priorytetową.

Wielozadaniowość - wbudowane moduły Pythona

sched – Planowanie zadań

Moduł **sched** pozwala na planowanie zadań do wykonania po określonym czasie. Możemy zdefiniować zadania, które będą wykonywane w przyszłości lub po upływie określonego czasu.

Przykładowe funkcje:

sched.scheduler() – Tworzy obiekt planera.

enter() – Planowanie zadania do wykonania po określonym czasie.

Wielozadaniowość - wbudowane moduły Pythona

signal – Obsługa sygnałów

Moduł **signal** pozwala na obsługę sygnałów w systemie operacyjnym, takich jak SIGINT (Ctrl+C), SIGTERM i inne. Sygnały mogą być używane do przerwania działania programu lub do komunikacji między procesami.

Przykładowe funkcje:

signal.signal() – Rejestruje funkcję obsługującą sygnał.

signal.pause() – Oczekuje na sygnał.

ZAAWANSOWANE TECHNIKI PROGRAMOWANIA

DR INŻ. MARCIN KULIK
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI
KATEDRA AUTOMATYZACJI NAPĘDÓW I ROBOTYKI

Wielozadaniowość - Numba

Numba to biblioteka Python przeznaczona do przyspieszania kodu poprzez kompilację funkcji Pythona na kod maszynowy w czasie wykonywania (ang. Just-In-Time Compilation, JIT). Wykorzystuje technologię LLVM (Low-Level Virtual Machine) do generowania wydajnego kodu maszynowego, co sprawia, że operacje matematyczne i algorytmy obliczeniowe stają się znacznie szybsze.

Wielozadaniowość - Numba

Główne cechy Numba

1. Kompilacja JIT (Just-In-Time):

- Kod jest kompilowany do kodu maszynowego podczas jego pierwszego uruchomienia, co pozwala osiągnąć wydajność porównywalną z C.

2. Obsługa NumPy:

- Funkcje korzystające z tablic NumPy mogą być przyspieszone przez Numba, dzięki czemu operacje na dużych zbiorach danych są znacznie szybsze.

3. Wsparcie dla obliczeń równoległych:

- Możliwość równoległego wykonywania kodu na CPU (@njit(parallel=True)).

Wielozadaniowość - Numba

Główne cechy Numba

4. Obsługa GPU:

•Numba umożliwia wykonywanie obliczeń na kartach graficznych (GPU), wspierając CUDA.

5. Łatwa integracja:

•Prosta implementacja - wystarczy dekorator @jit lub @njit, aby przyspieszyć kod.

•Można przyspieszać istniejący kod bez jego większej modyfikacji.

Aplikacje asynchroniczne i zdarzeniowe

Aplikacje asynchroniczne i zdarzeniowe

Aplikacje asynchroniczne i zdarzeniowe to aplikacje, które pozwalają na wykonywanie wielu operacji równolegle, nie blokując głównego wątku programu. Dzięki temu aplikacja może obsługiwać wiele zapytań lub zadań jednocześnie, nawet na jednym rdzeniu procesora. Jest to możliwe dzięki współbieżności opartej na zdarzeniach i korutynach, a nie na równoległości opartej na wielu wątkach lub procesach.

Aplikacje asynchroniczne i zdarzeniowe

- Asynchroniczność oznacza, że zadania mogą się przeplatać i nie muszą czekać na zakończenie jednego, aby rozpocząć kolejne.
- Zdarzeniowość polega na tym, że aplikacja reaguje na różne zdarzenia (np. przychodzące żądania HTTP, zakończenie zadania) i wywołuje odpowiednie procedury.

Dzięki tym technikom aplikacje asynchroniczne i zdarzeniowe są niezwykle wydajne i mogą obsługiwać setki tysięcy równoczesnych zapytań, co jest kluczowe w dzisiejszych wymagających systemach rozproszonych.

Aplikacje asynchroniczne i zdarzeniowe

W Pythonie asynchroniczność została wprowadzona do standardowej biblioteki z pomocą modułu **asyncio** i słów kluczowych **async** oraz **await**.

- **asyncio** jest biblioteką, która pozwala na tworzenie i zarządzanie zadaniami asynchronicznymi oraz obsługuje pętlę zdarzeń.
- Korutyny to funkcje, które zawierają słowo kluczowe **async**, co oznacza, że mogą być wykonywane asynchronicznie. Korutyny mogą „wstrzymać” swoje działanie, oddając kontrolę do pętli zdarzeń i czekać na inne zadania.
- Pętla zdarzeń to centralny element, który zarządza wykonaniem korutyn i innych zdarzeń.

Aplikacje asynchroniczne i zdarzeniowe

Korzyści aplikacji asynchronicznych i zdarzeniowych

- Wydajność - Asynchroniczność pozwala na wykonywanie wielu zadań równocześnie bez blokowania głównego wątku programu, dzięki czemu aplikacja jest znacznie bardziej wydajna.
- Skalowalność - Dzięki aplikacjom asynchronicznym można obsługiwać więcej żądań jednocześnie, co jest kluczowe w środowisku serwerów sieciowych o dużym ruchu, takich jak serwisy webowe.
- Responsywność - Asynchroniczność zwiększa responsywność aplikacji, ponieważ nie trzeba czekać na zakończenie jednego zadania, aby rozpocząć kolejne. W aplikacjach interaktywnych, takich jak aplikacje webowe, oznacza to szybszą reakcję na zapytania użytkowników.

Aplikacje asynchroniczne i zdarzeniowe

Narzędzia i biblioteki do aplikacji asynchronicznych w Pythonie

- **asyncio** to główna biblioteka asynchroniczna w Pythonie, która dostarcza mechanizmy takie jak pętla zdarzeń, korutyny oraz zarządzanie zadaniami.
- **aiohttp** to popularna biblioteka asynchroniczna do obsługi serwerów i klientów HTTP. Pozwala tworzyć aplikacje webowe oparte na asynchronicznych żądaniach HTTP.
- **FastAPI** to framework webowy, który wykorzystuje asynchroniczność do obsługi żądań HTTP i pozwala budować bardzo wydajne API.

Aplikacje asynchroniczne i zdarzeniowe

Narzędzia i biblioteki do aplikacji asynchronicznych w Pythonie

- **Twisted** jest frameworkiem, który wspiera programowanie zdarzeniowe i asynchroniczne, szczególnie popularny w aplikacjach sieciowych.
- **Celery** to narzędzie do obsługi kolejek zadań, przydatne w aplikacjach asynchronicznych do zarządzania długimi zadaniami w tle.

Aplikacje asynchroniczne i zdarzeniowe

Jak tworzyć aplikacje asynchroniczne i zdarzeniowe?

- Używanie korutyn i słowa **await**

  Każda funkcja asynchroniczna w Pythonie jest oznaczona słowem kluczowym **async**, a jej wywołania wymagają użycia **await**. W momencie napotkania słowa **await**, korutyna przekazuje kontrolę z powrotem do pętli zdarzeń, pozwalając innym zadaniom się wykonać, podczas gdy oczekiwane zadanie jeszcze trwa.

- Tworzenie i zarządzanie zadaniami

  W **asyncio** możemy tworzyć zadania (tasks) za pomocą **asyncio.create_task**, co pozwala na równoległe wykonywanie korutyn.

Aplikacje asynchroniczne i zdarzeniowe

Przykład: Asynchroniczne pobieranie danych z kilku URL

```
import asyncio
import aiohttp

async def fetch_data(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text()

async def main():
    url1 = 'https://example.com/data1'
    url2 = 'https://example.com/data2'

    task1 = asyncio.create_task(fetch_data(url1))
    task2 = asyncio.create_task(fetch_data(url2))

    data1, data2 = await asyncio.gather(task1, task2)

    print("Dane z URL 1:", data1)
    print("Dane z URL 2:", data2)

asyncio.run(main())
```

Aplikacje asynchroniczne i zdarzeniowe

Zarządzanie błędami w aplikacjach asynchronicznych

Obsługa błędów jest równie istotna w aplikacjach asynchronicznych, ponieważ błędy w korutynach mogą blokować działanie całej aplikacji.

- Można użyć **try** i **except** w korutynach.
- W przypadku **gather** można ustawić **return_exceptions=True**, aby zebrać wyniki i błędy.

Aplikacje asynchroniczne i zdarzeniowe

Praktyczne zastosowania aplikacji asynchronicznych i zdarzeniowych

- Serwery webowe - Frameworki jak **FastAPI** czy **aiohttp** umożliwiają tworzenie bardzo wydajnych serwerów HTTP, które obsługują wiele żądań jednocześnie.
- Przetwarzanie w tle i kolejki zadań - **Celery** umożliwia asynchroniczne przetwarzanie w tle, np. wysyłanie wiadomości e-mail czy generowanie raportów.
- Przetwarzanie danych w czasie rzeczywistym - Aplikacje asynchroniczne są stosowane w przetwarzaniu strumieni danych, np. w przemyśle IoT, monitoringu sieci czy obliczeniach finansowych.

ZAAWANSOWANE TECHNIKI PROGRAMOWANIA

DR INŻ. MARCIN KULIK
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI
KATEDRA AUTOMATYZACJI NAPĘDÓW I ROBOTYKI

Web Scrapping w Pythonie

Web Scrapping w Pythonie

Web scrapping (zbieranie danych ze stron internetowych) polega na automatycznym pobieraniu danych ze stron WWW. Za pomocą programów komputerowych można pobierać i przetwarzać informacje dostępne publicznie na stronach internetowych w celu ich późniejszej analizy. Jest to szczególnie przydatne w przypadkach, gdy dane nie są dostępne przez publiczne API lub gdy API ma ograniczenia w dostępie.

Web Scrapping w Pythonie

Web scrapping jest szeroko stosowany w wielu branżach i dziedzinach, między innymi do:

- Analizy cen – zbieranie informacji o cenach produktów z różnych sklepów w celu monitorowania rynku lub budowania aplikacji porównujących ceny.
- Analizy opinii – pozyskiwanie recenzji i opinii o produktach lub usługach z platform takich jak Yelp, Amazon czy TripAdvisor.
- Analizy mediów – automatyczne pobieranie artykułów i informacji z serwisów informacyjnych w celu analizy trendów, nastrojów społecznych czy słów kluczowych.
- Analizy danych rynkowych – zbieranie danych finansowych ze stron, takich jak Yahoo Finance, w celu analizy trendów rynkowych i wskaźników giełdowych.

Web Scrapping w Pythonie

- Badań naukowych – pozyskiwanie danych naukowych i artykułów z otwartych repozytoriów w celu analizy i badań naukowych.

Web scrapping przyspiesza i automatyzuje proces gromadzenia danych, który w innym przypadku mógłby wymagać dużo ręcznej pracy.

Web Scrapping w Pythonie

Python oferuje szereg bibliotek, które ułatwiają tworzenie aplikacji do web scrappingu. Najpopularniejsze z nich to:

- BeautifulSoup – biblioteka do parsowania HTML/XML, która pozwala na łatwe nawigowanie po strukturze dokumentu HTML, wyszukiwanie elementów, filtrowanie i przekształcanie danych.
- Requests – biblioteka do wykonywania zapytań HTTP, która umożliwia łatwe pobieranie stron internetowych. Używana w połączeniu z BeautifulSoup do pozyskiwania treści stron.
- Scrapy – framework do web scrappingu o większej funkcjonalności. Umożliwia tworzenie zaawansowanych skryptów, które wykonują złożone operacje na stronach i obsługują wiele stron jednocześnie.

Web Scrapping w Pythonie

- Selenium – narzędzie do automatyzacji przeglądarki, pozwala na interakcję ze stronami internetowymi jak użytkownik. Używane szczególnie do stron, które wymagają interakcji JavaScript (np. AJAX) do załadowania danych.
- Puppeteer (lub Pyppeteer) – narzędzie do automatyzacji przeglądarki Chromium, które pozwala na bardziej precyzyjną kontrolę nad przeglądarką. Stosowane przy bardziej zaawansowanych operacjach web scrappingu.

Każda z tych bibliotek ma swoje unikalne cechy i zalety, dlatego warto je dobierać zależnie od specyficznych wymagań projektu.

Aplikacje
Machine Learning/Neural Networks

Aplikacje naukowe w Pythonie

Python oferuje szeroką gamę bibliotek dedykowanych uczeniu maszynowemu i sieciom neuronowym:

- Tradycyjne ML: scikit-learn, ...
- Deep Learning: TensorFlow, PyTorch, Keras, ...
- Przetwarzanie danych: Pandas, NumPy, Dask, ...
- Wizualizacja: Matplotlib, Seaborn, Plotly, ...

Każda z tych bibliotek jest dobrze udokumentowana i aktywnie wspierana przez społeczność, co ułatwia ich naukę i implementację.

Aplikacje naukowe w Pythonie

Scikit-learn to jedna z najpopularniejszych bibliotek Python do uczenia maszynowego, która oferuje szeroki zestaw narzędzi do:

- Przetwarzania danych.
- Budowy i trenowania modeli.
- Oceny i optymalizacji algorytmów ML.

Zalety Scikit-learn:

- Prosta i intuicyjna składnia.
- Wszechstronność: obsługuje różnorodne algorytmy ML (klasyfikacja, regresja, clustering).
- Doskonała dokumentacja i szerokie wsparcie społeczności.

Aplikacje naukowe w Pythonie

Struktura działania w Scikit-learn:

1. Przygotowanie danych: Przetwarzanie, skalowanie, podział na zbiory treningowe i testowe.
2. Tworzenie modelu: Wybór odpowiedniego algorytmu (np. regresja, klasyfikacja).
3. Trenowanie modelu: Dopasowanie modelu do danych treningowych.
4. Ewaluacja modelu: Analiza wyników na zbiorze testowym.
5. Predykcja: Prognozowanie wyników dla nowych danych.

Aplikacje naukowe w Pythonie

Kluczowe moduły w Scikit-learn:

Moduł	Funkcja
sklearn.datasets	Zbiory danych do nauki i testowania.
sklearn.preprocessing	Przetwarzanie i skalowanie danych.
sklearn.model_selection	Podział danych, walidacja krzyżowa.
sklearn.linear_model	Algorytmy liniowe (regresja, klasyfikacja).
sklearn.ensemble	Algorytmy zespołowe (Random Forest, Gradient Boosting).
sklearn.metrics	Ocena modeli (accuracy, precision, recall).

Aplikacje naukowe w Pythonie

TensorFlow to otwartoźródłowa biblioteka opracowana przez Google, która służy do uczenia maszynowego i głębokiego uczenia. Obsługuje szeroki zakres zastosowań, od prostych algorytmów uczenia maszynowego po złożone modele głębokich sieci neuronowych.

Aplikacje naukowe w Pythonie

Kluczowe cechy TensorFlow:

- Operacje matematyczne: TensorFlow umożliwia efektywne obliczenia na tensorach, które są wielowymiarowymi tablicami danych.
- Automatyczne różniczkowanie: TensorFlow śledzi operacje matematyczne, aby automatycznie obliczać gradienty potrzebne do optymalizacji modeli.
- Wsparcie GPU/TPU: Przyspieszenie obliczeń dzięki wykorzystaniu procesorów graficznych (GPU) lub tensorowych (TPU).

Aplikacje naukowe w Pythonie

Kluczowe cechy TensorFlow:

- Wszechstronność: TensorFlow może być używany zarówno do uczenia maszynowego, jak i głębokiego uczenia w aplikacjach mobilnych, serwerowych i przeglądarkowych.
- TensorFlow Lite i TensorFlow.js: Umożliwia wdrażanie modeli na urządzeniach mobilnych lub w przeglądarkach internetowych.

Aplikacje naukowe w Pythonie

Główne komponenty TensorFlow:

- Tensor: Podstawowa jednostka danych w TensorFlow (podobna do tablic NumPy, ale z obsługą GPU).
- Model API: TensorFlow oferuje zarówno API niskiego poziomu, jak i wysokopoziomowe API Keras.
- **Keras** jest biblioteką wysokiego poziomu, wchodzącą w skład **TensorFlow** (od wersji TensorFlow 2.0). Upraszcza budowę, trenowanie i wdrażanie modeli głębokiego uczenia.

ZAAWANSOWANE TECHNIKI PROGRAMOWANIA

DR INŻ. MARCIN KULIK
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI
KATEDRA AUTOMATYZACJI NAPĘDÓW I ROBOTYKI

Wizualizacja danych

Wizualizacja danych

Matplotlib to wszechstronna biblioteka do tworzenia wykresów w Pythonie. Umożliwia generowanie różnorodnych wizualizacji, od prostych wykresów liniowych po zaawansowane wykresy 3D i interaktywne wizualizacje. Jest to jedno z najpopularniejszych narzędzi do wizualizacji danych w środowisku naukowym i inżynierskim.

Wizualizacja danych

Główne cechy Matplotlib

1. Wszechstronność:

- Obsługuje różne rodzaje wykresów, takie jak wykresy liniowe, punktowe, histogramy, wykresy kołowe, 3D i inne.
- Wysoka konfigurowalność – możliwość pełnej personalizacji wykresów.

2. Integracja:

- Działa z bibliotekami NumPy, Pandas, SciPy oraz Jupyter Notebook.
- Możliwość tworzenia grafik wektorowych i rastrowych (np. SVG, PNG, PDF).

Wizualizacja danych

Główne cechy Matplotlib

3. Kompatybilność:

- Możliwość tworzenia wykresów zarówno statycznych, jak i interaktywnych (np. za pomocą interfejsu matplotlib.pyplot).

4. Rozszerzalność:

- Umożliwia tworzenie animacji i dynamicznych wizualizacji za pomocą modułu animation.

5. Integracja z LaTeX:

- Możliwość dodawania formuł matematycznych na wykresach.

Wizualizacja danych

Moduły w Matplotlib

1. pyplot:

•Najczęściej używany moduł, który oferuje prosty interfejs podobny do MATLAB-a.

•Umożliwia szybkie tworzenie wykresów z użyciem funkcji takich jak plot(), bar(), scatter().

2. matplotlib.figure i matplotlib.axes:

•Moduły dla bardziej zaawansowanej kontroli nad wykresami i osiami.

3. matplotlib.animation:

•Tworzenie animacji.

4. matplotlib.style:

•Zestaw predefiniowanych stylów wykresów (np. ggplot, seaborn, dark_background).

Wizualizacja danych

Seaborn to biblioteka Python służąca do tworzenia atrakcyjnych i informacyjnych wizualizacji danych. Jest oparta na Matplotlib, ale oferuje bardziej intuicyjny interfejs i domyślne style, które pozwalają tworzyć estetyczne wykresy przy minimalnym nakładzie pracy. Seaborn integruje się z Pandas, co czyni go doskonałym narzędziem do wizualizacji danych tabelarycznych.

Wizualizacja danych

Główne cechy Seaborn

1. Estetyka wykresów:

- Domyślne style Seaborn są zaprojektowane z myślą o czytelności i atrakcyjności wizualnej.
- Możliwość łatwej personalizacji wyglądu wykresów.

2. Wykresy statystyczne:

- Obsługuje różne typy wykresów statystycznych, takie jak wykresy regresji, histogramy, mapy cieplne (heatmapy) i wykresy pudełkowe.

Wizualizacja danych

Główne cechy Seaborn

3. Integracja z Pandas:

- Automatycznie obsługuje dane w formacie Pandas DataFrame, co ułatwia pracę z danymi tabelarycznymi.

4. Złożone wizualizacje w prosty sposób:

- Tworzenie wykresów z podziałem na kategorie (facets) i wykresów wielowymiarowych jest szybkie i intuicyjne.

Wizualizacja danych

Folium to biblioteka Python służąca do tworzenia interaktywnych map. Jest oparta na bibliotece Leaflet.js, co umożliwia generowanie map w prosty sposób bez konieczności znajomości zaawansowanych technologii frontendowych.

Wizualizacja danych

Główne cechy Folium

1. Tworzenie interaktywnych map:

- Możliwość generowania map, które użytkownicy mogą przesuwać, przybliżać i oddalać.
- Obsługa różnych stylów map, takich jak OpenStreetMap, Stamen czy Mapbox.

2. Dodawanie elementów na mapę:

- Markery, wielokąty, linie, kształty i inne elementy geoprzestrzenne.
- Obsługa wyskakujących okienek (pop-up) i podpowiedzi (tooltip).

3. Obsługa danych geoprzestrzennych:

- Łatwe integrowanie danych w formatach takich jak GeoJSON, Shapefiles, czy dane tabelaryczne Pandas.

Wizualizacja danych

Główne cechy Folium

4. Interaktywne funkcje:

•Obsługa warstw (np. warstw ciepła).

•Dynamiczne mapy z animacjami i efektami.

5. Ekosystem w Pythonie:

•Bezproblemowa integracja z bibliotekami takimi jak Pandas, NumPy czy GeoPandas.

6. Łatwość użycia:

•Prosty interfejs umożliwia tworzenie map przy użyciu niewielkiej ilości kodu.

Łączenie Pythona z innymi językami programowania

Łączenie Pythona z innymi językami programowania

C/C++:

- Można używać fragmentów kodu napisanego w C/C++ w programie Python poprzez wykorzystanie odpowiednich narzędzi takich jak Cython.
- Można również używać bibliotek napisanych w C/C++ jako funkcje w Pythonie poprzez korzystanie z rozszerzeń CPython.

Rust:

- Rust jest językiem o bezpiecznym systemie typów i jest znacznie szybszy niż Python.
- Istnieją narzędzia, takie jak rust-cpython, które umożliwiają integrację Rusta z Pythonem, co pozwala na korzystanie z funkcji napisanych w Rust jako modułów Pythona.

Łączenie Pythona z innymi językami programowania

Fortran:

- Fortran jest językiem znacznie starszym niż Python, ale w wielu przypadkach nadal używanym, zwłaszcza w dziedzinie obliczeń naukowych i inżynierii.
- Można korzystać z kodu Fortran w Pythonie poprzez użycie narzędzi takich jak f2py.

Java:

- Chociaż Java jest maszyną wirtualną, a nie językiem kompilowanym do kodu maszynowego, to w niektórych zastosowaniach może być szybsza niż Python.
- Można korzystać z kodu Java w Pythonie za pomocą biblioteki JPype lub poprzez użycie interfejsu Java Native Interface (JNI).

Łączenie Pythona z innymi językami programowania

Julia:

- Julia jest nowoczesnym językiem programowania, zaprojektowanym specjalnie do obliczeń naukowych. Jest znacznie szybsza niż Python.
- Julia może być używana wraz z Pythonem dzięki bibliotece PyJulia do współpracy między oboma językami.

Cython:

- Chociaż Cython jest bardziej rozbudowanym dialektem Pythona niż odrębnym językiem, umożliwia kompilację kodu Pythona do kodu C, co może poprawić jego wydajność.
- Można również używać kodu C jako rozszerzeń w Cythonie.

Łączenie Pythona z innymi językami programowania

Biblioteki DLL (Dynamic Link Libraries) są popularne w środowisku Windows i mogą być napisane w różnych językach programowania. Przykłady języków, w których można utworzyć biblioteki DLL:

C:

- C jest jednym z najczęstszych języków używanych do tworzenia bibliotek DLL ze względu na swoją niskopoziomową naturę i bliskie powiązanie z systemem operacyjnym.

C++:

- C++ dziedziczy możliwość tworzenia bibliotek DLL z języka C i wprowadza dodatkowe funkcje zorientowane obiektowo.

Łączenie Pythona z innymi językami programowania

C# (przy użyciu .NET):

- Biblioteki DLL w C# często są kompatybilne z platformą .NET. Można korzystać z narzędzi takich jak Visual Studio do tworzenia bibliotek DLL z projektów C#.

Rust:

- Rust również umożliwia tworzenie bibliotek DLL, a dzięki integracji z językiem Python (np. za pomocą rust-cpython), można je używać w kodzie Pythona.

Fortran:

- Fortran może być używany do tworzenia bibliotek DLL, które później można zintegrować z kodem Pythona przy użyciu narzędzi takich jak f2py.

Łączenie Pythona z innymi językami programowania

Używanie DLL w Pythonie:

- Moduł **ctypes** w Pythonie umożliwia dynamiczne ładowanie bibliotek DLL i wywoływanie ich funkcji z poziomu Pythona.
- **CFFI** (C Foreign Function Interface) to moduł w Pythonie, który umożliwia wywoływanie funkcji z bibliotek C bezpośrednio z poziomu Pythona.
- **SWIG** (Simplified Wrapper and Interface Generator) to narzędzie, które automatyzuje proces tworzenia wrapperów dla kodu napisanego w różnych językach, umożliwiając korzystanie z nich w Pythonie.

Zastosowania bibliotek DLL w Pythonie obejmują:

Optymalizacja wydajności:

- Używanie funkcji napisanych w językach kompilowanych do kodu maszynowego, co może poprawić wydajność w porównaniu do kodu interpretowanego w Pythonie.

Integracja z istniejącym kodem:

- Wykorzystywanie istniejących bibliotek i modułów napisanych w innych językach w celu ponownego użycia już istniejącej funkcjonalności.

Przyspieszanie algorytmów:

- Implementowanie algorytmów o dużej wydajności w językach kompilowanych i korzystanie z nich w Pythonie w celu przyspieszenia obliczeń.

Dlaczego Rust?

- Bezpieczeństwo pamięci: Rust został zaprojektowany z myślą o eliminacji typowych błędów związanych z zarządzaniem pamięcią
- Wydajność: Rust pozwala na pisanie kodu niskopoziomowego, który jest bliski wydajnością do kodu napisanego w C i C++.
- Wielowątkowość: Rust posiada wbudowane mechanizmy, które ułatwiają pisanie bezpiecznego kodu wielowątkowego.

Podstawy języka Rust

- Struktura programu

```
fn main() {
    println!("Hello, world!");
}
```

- Zmienne i stałe

```
let x = 5;
// x = 6; // Błąd kompilacji
```

Podstawy języka Rust

- Aby zmienna była modyfikowalna, należy użyć słowa kluczowego mut:

```
let mut y = 10;
y = 15; // OK
```

- Stałe są deklarowane przy użyciu const:

```
const MAX_POINTS: u32 = 100_000;
```

Podstawy języka Rust

- Typy danych: Rust jest językiem statycznie typowanym, co oznacza, że typy wszystkich zmiennych są znane w czasie kompilacji:

```
let a: i32 = 10;
let b: f64 = 20.5;
let c: bool = true;
let d: char = 'R';
```

- Funkcje: Definiowanie funkcji w Rust jest proste i przypomina inne języki programowania:

```
fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

Podstawy języka Rust

- Własność i pożyczanie: Jedną z najważniejszych cech Rust jest system własności (ownership), który pomaga zarządzać pamięcią:
- Własność: Każdy zasób ma jednego właściciela w danym momencie.
- Pożyczanie: Rust pozwala na pożyczanie referencji do danych, ale zarządza nimi za pomocą zasad borrow checker.

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1; // s1 jest przeniesione do s2
    // println!("{}", s1); // Błąd: s1 nie jest już dostępne
}
```

Podstawy języka Rust

- Kontrola przepływu: Rust oferuje standardowe konstrukcje kontroli przepływu, takie jak if, loop, while, oraz for:

```
let number = 3;
if number < 5 {
    println!("condition was true");
} else {
    println!("condition was false");
}
let mut count = 0;
loop {
    count += 1;
    if count == 5 {
        break;
    }
}
```

```
while count != 0 {
    println!("{}", count);
    count -= 1;
}

let array = [10, 20, 30, 40, 50];

for element in array.iter() {

    println!("value is: {}", element);

}
```

Program Obliczający Liczbę Fibonacciego

```
fn main() {
    let n = 10;
    println!("Fibonacci number at position {} is: {}", n, fibonacci(n));
}
fn fibonacci(n: u32) -> u32 {
    if n <= 1 {
        n
    } else {
        fibonacci(n - 1) + fibonacci(n - 2)
    }
}
```

Program Obliczający Silnię

```
fn main() {
    let num = 5;
    println!("Factorial of {} is: {}", num, factorial(num));
}
fn factorial(n: u32) -> u32 {
    let mut result = 1;
    for i in 1..=n {
        result *= i;
    }
    result
}
```

Uruchomienie programu napisanego w Rust w VS Code

1) Instalacja Rust ze strony internetowej

2) Instalacja rozszerzenia Rust do VS Code – rust-analyzer

3) Utworzenie nowego projektu poprzez wpisanie komendy „cargo new my_project” w terminalu

4) Utworzenie pliku z kodem w folderze src

```
myrustproject/
│
├── Cargo.toml
└── src/
    └── main.rs
```

5) Kompilacja i uruchomienie programu komendą „cargo run” w terminalu

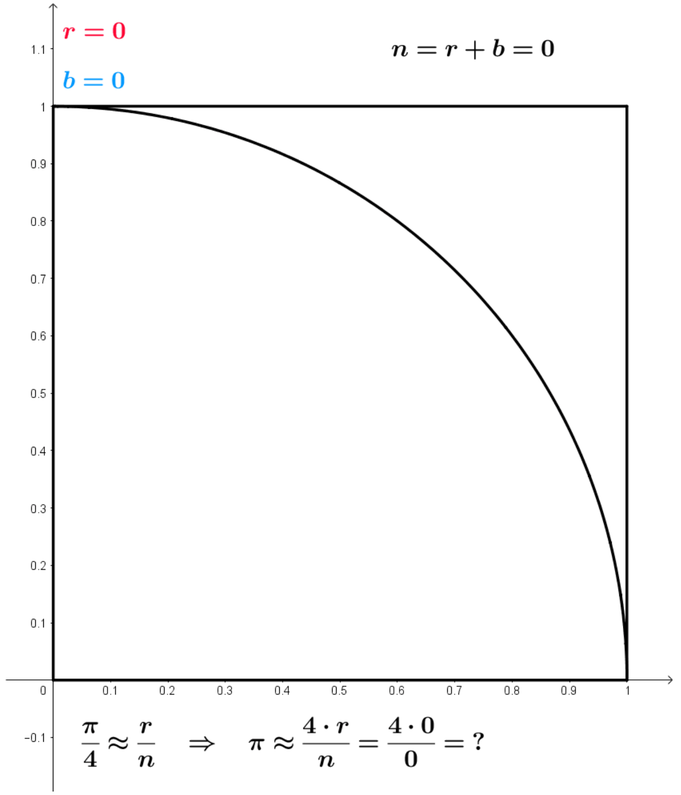
Python vs Rust

Cecha	Python	Rust
Przeznaczenie i Filozofia	Wysokopoziomowy, interpretowany, czytelność i prostota	Niskopoziomowy, kompilowany, wydajność i bezpieczeństwo
Zarządzanie Pamięcią	Automatyczne, garbage collector	Ręczne, system własności i pożyczania (ownership and borrowing)
Wydajność	Wolniejszy, interpretowany	Bardzo wysoka, kompilowany do kodu maszynowego
Ekosystem i Biblioteki	Bogaty ekosystem, szczególnie w analizie danych, AI	Młodszy ekosystem, szybki rozwój, silne wsparcie dla systemowego oprogramowania
Składnia i Język	Prosta, czytelna, wcięcia definiują bloki kodu	Złożona, zaawansowane abstrakcje, zarządzanie pamięcią i wątkowość
Paradygmaty Programowania	Obiektowe, funkcyjne, proceduralne	Wieloparadygmatowy, silne wsparcie dla systemowego i współbieżnego programowania
Typowanie	Dynamiczne	Statyczne
Popularne Zastosowania	Szybkie prototypowanie, analiza danych, web, AI	Systemy operacyjne, oprogramowanie wbudowane, gry, przetwarzanie danych w czasie rzeczywistym
Społeczność	Duża, aktywna	Rośnie, aktywna
Narzędzia i IDE	Wszechstronne wsparcie (VS Code, PyCharm, etc.)	Wsparcie rozwijające się (VS Code, CLion, etc.)

Łączenie Pythona z innymi językami programowania

Przykład: Obliczanie pola powierzchni koła metodą Monte Carlo – porównanie

- Python
- Python - numpy
- C++
- Rust
- Julia



ZAAWANSOWANE TECHNIKI PROGRAMOWANIA

DR INŻ. MARCIN KULIK
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI
KATEDRA AUTOMATYZACJI NAPĘDÓW I ROBOTYKI

Sprawy organizacyjne

- 15 godzin wykładu – 15 x 1h
- Forma zaliczenia: zaliczenie
  – test na ostatnim wykładzie
- Karta opisu przedmiotu – strona kursu oraz Wydziału

- Konsultacje – poniedziałek 12:45 - 13:30

pokój P1-6 lub zdalnie (email -> etele)

m.kulik@po.edu.pl

Literatura podstawowa (sylabus):

1. Eric Matthes, Python. Instrukcje dla programisty. Wydanie II, Helion 2020, e-book.
2. Adam Freeman, ASP.NET MVC 5. Zaawansowane programowanie (ebook), Helion 2015, e-book.
3. Steven F. Lott, Python. Programowanie funkcyjne, Helion 2019, e-book.

Literatura uzupełniająca (sylabus)

1. Al Sweigart, Automate the Boring Stuff with Python, 2019
2. Dane Hillard, Practices of the Python Pro, 2019

Sprawy organizacyjne

Tematyka zajęć:

1) Platformy programistyczne - MS Visual Studio
2) Środowiska programistyczne
3) Programowanie funkcyjne
4) Programowanie obiektowe
5) Dokumentacja kodu i wirtualne środowiska
6) Wzorce projektowe
7) Multitasking
8) Łączenie Pythona z innymi językami programowania i bibliotekami dynamicznymi
9) Przykłady współczesnych aplikacji i projektów z obszaru data science i sztucznej inteligencji.

Sprawy organizacyjne

Nazwa kursu na Moodle:

ZaaTecPR(2) # Zaawansowane techniki programowania

Sprawy organizacyjne

Python:

https://www.python.org/

https://www.programiz.com/python-programming/online-compiler/

https://www.online-python.com/

Visual Studio Code:

https://code.visualstudio.com/

PyCharm:

https://www.jetbrains.com/pycharm/

Anaconda:

https://www.anaconda.com/

Powtórzenie wiadomości

Python:

- Python to popularny język programowania, który jest znany ze swojej czytelności i prostoty.

- Jest interpretowany, co oznacza, że kod źródłowy jest wykonywany linia po linii przez interpreter Pythona.

Powtórzenie wiadomości

Typy danych w Pythonie:

- **Integer (int):** Liczby całkowite, np. 5, -3, 100.
- **Float (float):** Liczby zmiennoprzecinkowe, np. 3.14, -0.5, 2.0.
- **String (str):** Ciągi znaków, np. "Hello, World!".
- **Boolean (bool):** Wartości logiczne True lub False.
- **List (list):** Zmienna kolekcja elementów, np. [1, 2, 3].
- **Tuple (tuple):** Niezmienne kolekcje elementów, np. (1, 2, 3).
- **Dictionary (dict):** Kolekcja klucz-wartość, np. {"klucz": "wartość"}.

```
age = 30           # Integer
price = 19.99      # Float
name = "John"      # String
is_student = True  # Boolean
my_list = [1, 2, 3]  # List
my_tuple = (1, 2, 3) # Tuple
my_dict = {"key": "value"}  # Dictionary
```

Powtórzenie wiadomości

Zmienne i przypisywanie wartości:

- W Pythonie zmiennej przypisuje się wartość za pomocą operatora "=".
- Zmienne są dynamicznie typowane, co oznacza, że nie trzeba deklarować ich typu.

```
x = 5
print(type(x))
x = "Hello"
print(type(x))
```

```
<class 'int'>
<class 'str'>
```

Powtórzenie wiadomości

Instrukcje warunkowe:

- Instrukcje warunkowe pozwalają na wykonywanie różnych bloków kodu w zależności od spełnienia pewnych warunków.
- Przykładowe instrukcje warunkowe to **if**, **elif** (skrót od "else if") i **else**.

```
age = 100
if age >= 100:
    print("Jesteś stary")
else:
    print("Jesteś młody")
```

Powtórzenie wiadomości

Pętle:

- Pętle pozwalają na wykonywanie pewnych fragmentów kodu wielokrotnie.
- W Pythonie istnieją dwie główne pętle: **for** do iterowania przez kolekcje i **while** do wykonywania kodu w oparciu o warunki.

```
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    print(number,end = ', ')
print()
count = 0
while count < 5:
    print(count,end = ', ')
    count += 1
print()
```

```
1, 2, 3, 4, 5,
0, 1, 2, 3, 4,
```

Powtórzenie wiadomości

Funkcje:

- Funkcje to bloki kodu, które można wielokrotnie wywoływać, aby wykonywać określone zadania.
- Funkcje są definiowane za pomocą słowa kluczowego **def**.

```
def greet(name):
    return "Hello, " + name

message = greet("John")
print(message)
```

```
Hello, John
```

Powtórzenie wiadomości

Importowanie modułów:

- Python pozwala na importowanie modułów i bibliotek, aby rozszerzyć funkcjonalność programu.
- Importowanie odbywa się za pomocą instrukcji **import**.

```
import math
print(math.sqrt(25))
```

Powtórzenie wiadomości

Operatory:

- Python obsługuje różne operatory matematyczne (+, -, *, /) oraz operatory porównania (==, !=, <, >), które są używane w wyrażeniach.

```
x = 10
y = 5
result1 = x + y
result2 = x * y
```

Powtórzenie wiadomości

Kontrola przepływu programu:

- Kontrola przepływu programu obejmuje instrukcje warunkowe, pętle i wyjątki, które pozwalają na sterowanie, jak kod jest wykonywany.
- Instrukcje **break** i **continue** pozwalają na manipulację pętlami.

```
for i in range(5):
    if i == 3:
        continue  # Pomija iterację, gdy i = 3
    print(i)
```

```
0
1
2
4
```

```
for i in range(5):
    if i == 3:
        break  # Przerywa pętlę, gdy i = 3
    print(i)
```

```
0
1
2
```

Powtórzenie wiadomości

Obsługa wyjątków:

- Wyjątki są sytuacjami błędów, które mogą wystąpić podczas wykonywania programu.
- Instrukcje **try**, **except** i **finally** (, else) pozwalają na obsługę i zarządzanie wyjątkami.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Dzielenie przez zero jest niedozwolone")
finally:
    print("Wykonuję się zawsze")
```

```
Dzielenie przez zero jest niedozwolone
Wykonuję się zawsze
```

Programowanie obiektowe w Pythonie

Obiekty i klasy:

- W Pythonie wszystko jest obiektem, co oznacza, że każda zmienna (obiekt) przechowuje dane (właściwości) i metody z nimi związane.
- Klasa to szablon lub blueprint, który definiuje właściwości i zachowanie obiektów. Obiekty są instancjami klas.

Programowanie obiektowe w Pythonie

Definicja klasy:

- Klasa jest tworzona za pomocą słowa kluczowego **class**, a konstruktor klasy (metoda **__init__**) jest używany do inicjalizacji obiektów.

```
class Osoba:
    def __init__(self, imie, nazwisko):
        self.imie = imie
        self.nazwisko = nazwisko

nowa_osoba = Osoba("Jan", "Kowalski")
```

Programowanie obiektowe w Pythonie

```
class Silnik:
    def start(self):
        print("Silnik rusza")

class Samochod:
    def __init__(self):
        self.silnik = Silnik()

    def uruchom_samochod(self):
        print("Uruchamiam samochód")
        self.silnik.start()

moj_samochod = Samochod()
moj_samochod.uruchom_samochod()  # Wywołuje metodę Silnik.start()
```

Kompozycja:

- Kompozycja polega na tworzeniu bardziej złożonych obiektów poprzez składanie ich z innych obiektów.

Programowanie obiektowe w Pythonie

Moduły i importowanie:

- Kody związane z klasami można organizować w różnych modułach i importować je w innych częściach projektu za pomocą instrukcji **import**.

```
# Moduł "moja_klasa.py"
class MojaKlasa:
    def __init__(self, x):
        self.x = x

# Inny plik
from moja_klasa import MojaKlasa

obiekt = MojaKlasa(42)
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- Metody specjalne (nazywane również magicznymi metodami lub metodami podwójnego podkreślenia) to specjalnie zdefiniowane metody w języku Python, które mają określone znaczenie i są wywoływane automatycznie w określonych sytuacjach.
- Metody specjalne pozwalają na dostosowanie zachowania obiektów w Pythonie do różnych sytuacji i operacji. Poprawnie zdefiniowane metody specjalne umożliwiają bardziej intuicyjne i elastyczne korzystanie z obiektów w ramach języka Python.

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__init__(self, …)**
- Metoda konstruktora, inicjalizująca nowe instancje klasy.

```
class Osoba:
    def __init__(self, imie, nazwisko):
        self.imie = imie
        self.nazwisko = nazwisko

osoba = Osoba("Jan", "Kowalski")
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__str__(self)**
- Metoda zwracająca reprezentację tekstową obiektu. Wywoływana przez funkcję **str()** lub print(obiekt).

```
class Ksiazka:
    def __init__(self, tytul, autor):
        self.tytul = tytul
        self.autor = autor

    def __str__(self):
        return f"{self.tytul} by {self.autor}"

ksiazka = Ksiazka("Władca Pierścieni", "J.R.R. Tolkien")
print(str(ksiazka))  # Wywołuje metodę __str__
```

```
Władca Pierścieni by J.R.R. Tolkien
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__len__(self)**
- Metoda zwracająca długość (liczbę elementów) obiektu. Wywoływana przez funkcję **len()**.

```
class ListaLiczb:
    def __init__(self, liczby):
        self.liczby = liczby

    def __len__(self):
        return len(self.liczby)

lista = ListaLiczb([1, 2, 3, 4, 5])
print(len(lista))  # Wywołuje metodę __len__
```

```
5
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__getitem__(self, key)**
- Metoda pozwalająca na dostęp do elementów obiektu za pomocą indeksu. Wywoływana, gdy używamy nawiasów kwadratowych ([]) do odczytu wartości.

```
class ListaLiczb:
    def __init__(self, liczby):
        self.liczby = liczby

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

lista = ListaLiczb([1, 2, 3, 4, 5])
print(lista[2])  # Wywołuje metodę __getitem__ (zwróci 3)
```

```
3
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__setitem__(self, key, value)**
- Metoda pozwalająca na ustawianie wartości elementów obiektu za pomocą indeksu. Wywoływana, gdy używamy nawiasów kwadratowych ([]) do przypisania wartości.

```
class ListaLiczb:
    def __init__(self, liczby):
        self.liczby = liczby

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

lista = ListaLiczb([1, 2, 3, 4, 5])
lista[2] = 10  # Wywołuje metodę __setitem__, zmienia 3 na 10
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__delitem__(self, key)**
- Metoda pozwalająca na usunięcie elementu obiektu za pomocą indeksu. Wywoływana, gdy używamy instrukcji **del**.

```
class ListaLiczb:
    def __init__(self, liczby):
        self.liczby = liczby

    def __delitem__(self, index):
        del self.liczby[index]

lista = ListaLiczb([1, 2, 3, 4, 5])
del lista[2]  # Wywołuje metodę __delitem__, usuwa 3
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__iter__(self)**
- Metoda zwracająca iterator, który umożliwia iterację po obiekcie. Wywoływana przez funkcję **iter()**.

```
class LiczbyParzyste:
    def __init__(self, limit):
        self.limit = limit

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n <= self.limit:
            wynik = self.n
            self.n += 2
            return wynik
        else:
            raise StopIteration

parzyste = LiczbyParzyste(10)
for liczba in parzyste:
    print(liczba)  # Iteruje po liczbach parzystych
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__eq__(self, other)**
- Metoda porównująca obiekt z innym obiektem. Wywoływana przez operator ==.

```
class Osoba:
    def __init__(self, imie, nazwisko):
        self.imie = imie
        self.nazwisko = nazwisko

    def __eq__(self, other):
        return self.imie == other.imie and self.nazwisko == other.nazwisko

osoba1 = Osoba("Jan", "Kowalski")
osoba2 = Osoba("Jan", "Kowalski")
print(osoba1 == osoba2)  # Wywołuje metodę __eq__, zwróci True
```

__lt__(self, other), __le__(self, other), __ne__(self, other), __gt__(self, other), __ge__(self, other)

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__add__(self, other)**
- Metoda pozwalająca na dodawanie dwóch obiektów za pomocą operatora +.

```
class Liczba:
    def __init__(self, wartosc):
        self.wartosc = wartosc

    def __add__(self, other):
        return self.wartosc + other.wartosc

liczba1 = Liczba(5)
liczba2 = Liczba(3)
wynik = liczba1 + liczba2  # Wywołuje metodę __add__, zwróci 8
```

__sub__(self, other), __mul__(self, other), __truediv__(self, other)

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__repr__(self)**
- Metoda zwracająca reprezentację tekstową obiektu, która jest używana przez funkcję **repr()**. Jest to zazwyczaj bardziej techniczna reprezentacja obiektu niż **__str__**.

```
class Ksiazka:
    def __init__(self, tytul, autor):
        self.tytul = tytul
        self.autor = autor

    def __repr__(self):
        return f"Ksiazka('{self.tytul}', '{self.autor}')"

ksiazka = Ksiazka("Władca Pierścieni", "J.R.R. Tolkien")
print(repr(ksiazka))  # Wywołuje metodę __repr__
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__call__(self, ...)**
- Metoda umożliwiająca wywoływanie obiektu jak funkcji. Wywoływana, gdy nawiasy okrągłe () są używane z obiektem.

```
class Licznik:
    def __init__(self):
        self.wartosc = 0

    def __call__(self, x):
        self.wartosc += x
        return self.wartosc

licznik = Licznik()
print(licznik(5))  # Wywołuje metodę __call__, zwróci 5
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__contains__(self, item)**
- Metoda sprawdzająca, czy obiekt zawiera określony element. Wywoływana przez operator **in**.

```
class ListaLiczb:
    def __init__(self, liczby):
        self.liczby = liczby

    def __contains__(self, item):
        return item in self.liczby

lista = ListaLiczb([1, 2, 3, 4, 5])
print(3 in lista)  # Wywołuje metodę __contains__, zwróci True
```

Dziedziczenie (Inheritance):

- Dziedziczenie umożliwia tworzenie nowych klas na podstawie istniejących klas.
- Klasa dziedzicząca (potomna) może dziedziczyć metody i atrybuty od klasy bazowej (nadrzędnej).
- Dziedziczenie pomaga w tworzeniu hierarchii klas, co pozwala na ponowne wykorzystanie kodu i organizację struktury programu.

Programowanie obiektowe w Pythonie

```
class Zwierze:
    def __init__(self, imie):
        self.imie = imie
    def jaki_glos(self):
        pass
class Kot(Zwierze):
    def jaki_glos(self):
        return "Miau!"
class Pies(Zwierze):
    def jaki_glos(self):
        return "Hau!"

kot = Kot("Filemon")
pies = Pies("Burek")
print(kot.imie, "wydaje dźwięk:", kot.jaki_glos())  # Wynik: Filemon wydaje dźwięk: Miau!
print(pies.imie, "wydaje dźwięk:", pies.jaki_glos())  # Wynik: Burek wydaje dźwięk: Hau!
```

Dziedziczenie (Inheritance):

- W tym przykładzie **Kot** i **Pies** dziedziczą po klasie **Zwierze**, a każda z tych klas nadpisuje metodę **jaki_glos**, aby dostarczyć własną implementację. Dzięki dziedziczeniu możemy tworzyć konkretne klasy na bazie ogólnych klas nadrzędnych.

Programowanie obiektowe w Pythonie

Polimorfizm (Polymorphism):

- Polimorfizm umożliwia wywoływanie tych samych metod na różnych obiektach, niezależnie od ich konkretnej implementacji.
- Istnieje dwa rodzaje polimorfizmu w Pythonie: polimorfizm oparty na funkcjach (funkcje mogą działać na różnych typach danych) i polimorfizm oparty na klasach (klasy potomne mogą nadpisywać metody klasy bazowej).
- Polimorfizm pomaga tworzyć bardziej elastyczny i rozszerzalny kod.

Programowanie obiektowe w Pythonie

Polimorfizm (Polymorphism):

- Funkcja **przywitaj** przyjmuje obiekt zwierzęcia i wywołuje jego metodę **jaki_glos**, niezależnie od tego, czy to kot, czy pies. To jest przykład polimorfizmu, gdzie ta sama funkcja działa na różnych obiektach.

```
def przywitaj(zwierze):
    print("Cześć,", zwierze.imie, "wydaje dźwięk:", zwierze.jaki_glos())

kot = Kot("Filemon")
pies = Pies("Burek")

przywitaj(kot)  # Wynik: Cześć, Filemon wydaje dźwięk: Miau!
przywitaj(pies)  # Wynik: Cześć, Burek wydaje dźwięk: Hau!
```

Abstrakcja (Abstraction):

- Abstrakcja polega na ukrywaniu szczegółów implementacji i prezentowaniu tylko istotnych informacji na zewnątrz.
- W Pythonie możemy tworzyć klasy abstrakcyjne, które definiują abstrakcyjne metody, które muszą być zaimplementowane przez klasy dziedziczące.
- Abstrakcja pomaga w tworzeniu interfejsów i definiowaniu ogólnych zachowań bez potrzeby precyzyjnej implementacji.

Programowanie obiektowe w Pythonie

```
from abc import ABC, abstractmethod

class Pojazd(ABC):
    def __init__(self, marka):
        self.marka = marka
    @abstractmethod
    def jazda(self):
        pass
class Samochod(Pojazd):
    def jazda(self):
        return "Jadę samochodem marki " + self.marka
class Motocykl(Pojazd):
    def jazda(self):
        return "Jadę motocyklem marki " + self.marka

samochod = Samochod("Toyota")
motocykl = Motocykl("Honda")
print(samochod.jazda())  # Wynik: Jadę samochodem marki Toyota
print(motocykl.jazda())  # Wynik: Jadę motocyklem marki Honda
```

Abstrakcja (Abstraction):

- W tym przykładzie **Pojazd** jest klasą abstrakcyjną z abstrakcyjną metodą **jazda**. Klasy **Samochod** i **Motocykl** dziedziczą po **Pojazd** i implementują tę abstrakcyjną metodę. Dzięki temu możemy stworzyć ogólny interfejs dla pojazdów i wymuszać implementację metody **jazda** w klasach potomnych.

ZAAWANSOWANE TECHNIKI PROGRAMOWANIA

DR INŻ. MARCIN KULIK
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI
KATEDRA AUTOMATYZACJI NAPĘDÓW I ROBOTYKI

Programowanie obiektowe w Pythonie

Metody specjalne:

- Metody specjalne (nazywane również magicznymi metodami lub metodami podwójnego podkreślenia) to specjalnie zdefiniowane metody w języku Python, które mają określone znaczenie i są wywoływane automatycznie w określonych sytuacjach.
- Metody specjalne pozwalają na dostosowanie zachowania obiektów w Pythonie do różnych sytuacji i operacji. Poprawnie zdefiniowane metody specjalne umożliwiają bardziej intuicyjne i elastyczne korzystanie z obiektów w ramach języka Python.

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__init__(self, …)**
- Metoda konstruktora, inicjalizująca nowe instancje klasy.

```
class Osoba:
    def __init__(self, imie, nazwisko):
        self.imie = imie
        self.nazwisko = nazwisko

osoba = Osoba("Jan", "Kowalski")
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__str__(self)**
- Metoda zwracająca reprezentację tekstową obiektu. Wywoływana przez funkcję **str()** lub print(obiekt).

```
class Ksiazka:
    def __init__(self, tytul, autor):
        self.tytul = tytul
        self.autor = autor

    def __str__(self):
        return f"{self.tytul} by {self.autor}"

ksiazka = Ksiazka("Władca Pierścieni", "J.R.R. Tolkien")
print(str(ksiazka))  # Wywołuje metodę __str__
```

```
Władca Pierścieni by J.R.R. Tolkien
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__len__(self)**
- Metoda zwracająca długość (liczbę elementów) obiektu. Wywoływana przez funkcję **len()**.

```
class ListaLiczb:
    def __init__(self, liczby):
        self.liczby = liczby

    def __len__(self):
        return len(self.liczby)

lista = ListaLiczb([1, 2, 3, 4, 5])
print(len(lista))  # Wywołuje metodę __len__
```

```
5
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__getitem__(self, key)**
- Metoda pozwalająca na dostęp do elementów obiektu za pomocą indeksu. Wywoływana, gdy używamy nawiasów kwadratowych ([]) do odczytu wartości.

```
class ListaLiczb:
    def __init__(self, liczby):
        self.liczby = liczby

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

lista = ListaLiczb([1, 2, 3, 4, 5])
print(lista[2])  # Wywołuje metodę __getitem__ (zwróci 3)
```

```
3
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__setitem__(self, key, value)**
- Metoda pozwalająca na ustawianie wartości elementów obiektu za pomocą indeksu. Wywoływana, gdy używamy nawiasów kwadratowych ([]) do przypisania wartości.

```
class ListaLiczb:
    def __init__(self, liczby):
        self.liczby = liczby

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

lista = ListaLiczb([1, 2, 3, 4, 5])
lista[2] = 10  # Wywołuje metodę __setitem__, zmienia 3 na 10
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__delitem__(self, key)**
- Metoda pozwalająca na usunięcie elementu obiektu za pomocą indeksu. Wywoływana, gdy używamy instrukcji **del**.

```
class ListaLiczb:
    def __init__(self, liczby):
        self.liczby = liczby

    def __delitem__(self, index):
        del self.liczby[index]

lista = ListaLiczb([1, 2, 3, 4, 5])
del lista[2]  # Wywołuje metodę __delitem__, usuwa 3
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__iter__(self)**
- Metoda zwracająca iterator, który umożliwia iterację po obiekcie. Wywoływana przez funkcję **iter()**.

```
class LiczbyParzyste:
    def __init__(self, limit):
        self.limit = limit

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n <= self.limit:
            wynik = self.n
            self.n += 2
            return wynik
        else:
            raise StopIteration

parzyste = LiczbyParzyste(10)
for liczba in parzyste:
    print(liczba)  # Iteruje po liczbach parzystych
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__eq__(self, other)**
- Metoda porównująca obiekt z innym obiektem. Wywoływana przez operator ==.

```
class Osoba:
    def __init__(self, imie, nazwisko):
        self.imie = imie
        self.nazwisko = nazwisko

    def __eq__(self, other):
        return self.imie == other.imie and self.nazwisko == other.nazwisko

osoba1 = Osoba("Jan", "Kowalski")
osoba2 = Osoba("Jan", "Kowalski")
print(osoba1 == osoba2)  # Wywołuje metodę __eq__, zwróci True
```

__lt__(self, other), __le__(self, other), __ne__(self, other), __gt__(self, other), __ge__(self, other)

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__add__(self, other)**
- Metoda pozwalająca na dodawanie dwóch obiektów za pomocą operatora +.

```
class Liczba:
    def __init__(self, wartosc):
        self.wartosc = wartosc

    def __add__(self, other):
        return self.wartosc + other.wartosc

liczba1 = Liczba(5)
liczba2 = Liczba(3)
wynik = liczba1 + liczba2  # Wywołuje metodę __add__, zwróci 8
```

__sub__(self, other), __mul__(self, other), __truediv__(self, other)

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__repr__(self)**
- Metoda zwracająca reprezentację tekstową obiektu, która jest używana przez funkcję **repr()**. Jest to zazwyczaj bardziej techniczna reprezentacja obiektu niż **__str__**.

```
class Ksiazka:
    def __init__(self, tytul, autor):
        self.tytul = tytul
        self.autor = autor

    def __repr__(self):
        return f"Ksiazka('{self.tytul}', '{self.autor}')"

ksiazka = Ksiazka("Władca Pierścieni", "J.R.R. Tolkien")
print(repr(ksiazka))  # Wywołuje metodę __repr__
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__call__(self, ...)**
- Metoda umożliwiająca wywoływanie obiektu jak funkcji. Wywoływana, gdy nawiasy okrągłe () są używane z obiektem.

```
class Licznik:
    def __init__(self):
        self.wartosc = 0

    def __call__(self, x):
        self.wartosc += x
        return self.wartosc

licznik = Licznik()
print(licznik(5))  # Wywołuje metodę __call__, zwróci 5
```

Programowanie obiektowe w Pythonie

Metody specjalne:

- **__contains__(self, item)**
- Metoda sprawdzająca, czy obiekt zawiera określony element. Wywoływana przez operator **in**.

```
class ListaLiczb:
    def __init__(self, liczby):
        self.liczby = liczby

    def __contains__(self, item):
        return item in self.liczby

lista = ListaLiczb([1, 2, 3, 4, 5])
print(3 in lista)  # Wywołuje metodę __contains__, zwróci True
```

Programowanie obiektowe w Pythonie

Dziedziczenie (Inheritance):

- Dziedziczenie umożliwia tworzenie nowych klas na podstawie istniejących klas.
- Klasa dziedzicząca (potomna) może dziedziczyć metody i atrybuty od klasy bazowej (nadrzędnej).
- Dziedziczenie pomaga w tworzeniu hierarchii klas, co pozwala na ponowne wykorzystanie kodu i organizację struktury programu.

Programowanie obiektowe w Pythonie

```
class Zwierze:
    def __init__(self, imie):
        self.imie = imie
    def jaki_glos(self):
        pass
class Kot(Zwierze):
    def jaki_glos(self):
        return "Miau!"
class Pies(Zwierze):
    def jaki_glos(self):
        return "Hau!"

kot = Kot("Filemon")
pies = Pies("Burek")
print(kot.imie, "wydaje dźwięk:", kot.jaki_glos())  # Wynik: Filemon wydaje dźwięk: Miau!
print(pies.imie, "wydaje dźwięk:", pies.jaki_glos())  # Wynik: Burek wydaje dźwięk: Hau!
```

Dziedziczenie (Inheritance):

- W tym przykładzie **Kot** i **Pies** dziedziczą po klasie **Zwierze**, a każda z tych klas nadpisuje metodę **jaki_glos**, aby dostarczyć własną implementację. Dzięki dziedziczeniu możemy tworzyć konkretne klasy na bazie ogólnych klas nadrzędnych.

Programowanie obiektowe w Pythonie

Polimorfizm (Polymorphism):

- Polimorfizm umożliwia wywoływanie tych samych metod na różnych obiektach, niezależnie od ich konkretnej implementacji.
- Istnieje dwa rodzaje polimorfizmu w Pythonie: polimorfizm oparty na funkcjach (funkcje mogą działać na różnych typach danych) i polimorfizm oparty na klasach (klasy potomne mogą nadpisywać metody klasy bazowej).
- Polimorfizm pomaga tworzyć bardziej elastyczny i rozszerzalny kod.

Programowanie obiektowe w Pythonie

Polimorfizm (Polymorphism):

- Funkcja **przywitaj** przyjmuje obiekt zwierzęcia i wywołuje jego metodę **jaki_glos**, niezależnie od tego, czy to kot, czy pies. To jest przykład polimorfizmu, gdzie ta sama funkcja działa na różnych obiektach.

```
def przywitaj(zwierze):
    print("Cześć,", zwierze.imie, "wydaje dźwięk:", zwierze.jaki_glos())

kot = Kot("Filemon")
pies = Pies("Burek")

przywitaj(kot)  # Wynik: Cześć, Filemon wydaje dźwięk: Miau!
przywitaj(pies)  # Wynik: Cześć, Burek wydaje dźwięk: Hau!
```

Abstrakcja (Abstraction):

- Abstrakcja polega na ukrywaniu szczegółów implementacji i prezentowaniu tylko istotnych informacji na zewnątrz.
- W Pythonie możemy tworzyć klasy abstrakcyjne, które definiują abstrakcyjne metody, które muszą być zaimplementowane przez klasy dziedziczące.
- Abstrakcja pomaga w tworzeniu interfejsów i definiowaniu ogólnych zachowań bez potrzeby precyzyjnej implementacji.

Programowanie obiektowe w Pythonie

```
from abc import ABC, abstractmethod

class Pojazd(ABC):
    def __init__(self, marka):
        self.marka = marka
    @abstractmethod
    def jazda(self):
        pass
class Samochod(Pojazd):
    def jazda(self):
        return "Jadę samochodem marki " + self.marka
class Motocykl(Pojazd):
    def jazda(self):
        return "Jadę motocyklem marki " + self.marka

samochod = Samochod("Toyota")
motocykl = Motocykl("Honda")
print(samochod.jazda())  # Wynik: Jadę samochodem marki Toyota
print(motocykl.jazda())  # Wynik: Jadę motocyklem marki Honda
```

Abstrakcja (Abstraction):

- W tym przykładzie **Pojazd** jest klasą abstrakcyjną z abstrakcyjną metodą **jazda**. Klasy **Samochod** i **Motocykl** dziedziczą po **Pojazd** i implementują tę abstrakcyjną metodę. Dzięki temu możemy stworzyć ogólny interfejs dla pojazdów i wymuszać implementację metody **jazda** w klasach potomnych.

Programowanie funkcyjne w Pythonie

Programowanie funkcyjne w Pythonie to podejście do tworzenia oprogramowania, które opiera się głównie na funkcjach. Główne cechy programowania funkcyjnego to:

- Brak efektów ubocznych: Funkcje nie powinny zmieniać stanu programu ani dokonywać zmian w zmiennych poza nimi samymi. To znaczy, że funkcja zawsze zwraca ten sam wynik dla tych samych argumentów, bez wpływania na jakiekolwiek globalne zmienne.
- Funkcje jako obiekty pierwszego rzędu: W języku Python funkcje są obiektami pierwszego rzędu, co oznacza, że można je przypisywać do zmiennych, przekazywać jako argumenty do innych funkcji i zwracać jako wartości z innych funkcji.
- Wysoki poziom abstrakcji: Programowanie funkcyjne często opiera się na abstrakcjach takich jak mapowanie, filtracja i redukcja. Dzięki nim można operować na kolekcjach danych w sposób bardziej zwięzły i zrozumiały.

Programowanie funkcyjne w Pythonie

Funkcje anonimowe (lambda):

- Funkcje lambda są używane do definiowania krótkich i jednorazowych funkcji w miejscach, gdzie potrzebna jest taka funkcja, ale nie ma potrzeby tworzenia pełnej funkcji za pomocą instrukcji def. (często wykorzystuje się wraz z funkcjami wyższego rzędu, takimi jak map, filter i sorted)
- Funkcje lambda są zwykle bardzo krótkie i przeznaczone do wykonywania prostych operacji. Wielokrotne instrukcje są nieodpowiednie w funkcjach lambda.

```
square = lambda x: x * x
add = lambda x, y: x + y

print(square(5))  # Wynik: 25
print(add(3, 4))  # Wynik: 7
```

Programowanie funkcyjne w Pythonie

Funkcja map:

- **map** to wbudowana funkcja w Pythonie, która służy do stosowania określonej funkcji do każdego elementu w liście, krotce lub innej iterowalnej strukturze danych.
- Po zastosowaniu funkcji **map** otrzymujemy nową sekwencję, która zawiera wyniki wywołania funkcji na każdym elemencie oryginalnej sekwencji.
- Jest to przydatne narzędzie do przekształcania danych w jednym kroku, na przykład przekształcania listy liczb w listę ich kwadratów.

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x * x, numbers))
# Wynik: [1, 4, 9, 16, 25]
```

Programowanie funkcyjne w Pythonie

Funkcja filter:

- **filter** to wbudowana funkcja w Pythonie, która służy do filtrowania elementów z danej sekwencji na podstawie warunku zdefiniowanego w funkcji.
- Po zastosowaniu funkcji **filter** otrzymujemy nową sekwencję, która zawiera tylko te elementy, które spełniają podany warunek.
- Jest to przydatne narzędzie do wybierania elementów z sekwencji na podstawie określonych kryteriów.

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

Programowanie funkcyjne w Pythonie

Funkcja reduce:

- **reduce** to funkcja znajdująca się w module functools, która służy do łączenia elementów w sekwencji przy użyciu określonej funkcji.
- Funkcja **reduce** iteruje przez sekwencję i stosuje podaną funkcję do pary elementów, a następnie stosuje ją ponownie do wyniku i następnego elementu, tworząc pojedynczy wynik.
- Jest to przydatne narzędzie do wykonywania operacji redukujących, takich jak obliczanie iloczynu elementów w liście.

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Wynik: 120 (1 * 2 * 3 * 4 * 5)
```

Programowanie funkcyjne w Pythonie

partial - Częściowe stosowanie funkcji:

- Funkcja **partial** pozwala na tworzenie nowych funkcji poprzez częściowe stosowanie istniejących funkcji, co jest przydatne, gdy chcemy zdefiniować funkcje z ustalonymi argumentami.

```
from functools import partial

def multiply(x, y):
    return x * y

double = partial(multiply, 2)
triple = partial(multiply, 3)
print(double(5))  # Wynik: 10
print(triple(5))  # Wynik: 15
```

Programowanie funkcyjne w Pythonie

Zastosowanie funkcji lambda z innymi funkcjami

```
# Przykład 1: Funkcja lambda jako argument funkcji map
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
# Wynik: [1, 4, 9, 16, 25]

# Przykład 2: Funkcja lambda jako argument funkcji filter
even = list(filter(lambda x: x % 2 == 0, numbers))
# Wynik: [2, 4]

# Przykład 3: Funkcja lambda do sortowania listy słów według ich długości
words = ["jabłko", "banan", "wiśnia", "truskawka", "ananas"]
sorted_words = sorted(words, key=lambda word: len(word))
# Wynik: ['banan', 'jabłko', 'wiśnia', 'ananas', 'truskawka']
```

ZAAWANSOWANE TECHNIKI PROGRAMOWANIA

DR INŻ. MARCIN KULIK
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI
KATEDRA AUTOMATYZACJI NAPĘDÓW I ROBOTYKI

Programowanie funkcyjne w Pythonie

List comprehensions:

- wyrażenia w języku Python, które pozwalają na tworzenie nowych list na podstawie istniejących list, sekwencji lub iterowalnych obiektów w bardziej zwięzły i czytelny sposób. Są często używane w Pythonie do transformacji, filtrowania i generowania nowych list.

```
nowa_lista = [wyrażenie for element in sekwencja if warunek]
```

gdzie:

nowa_lista to lista, którą chcemy utworzyć.

wyrażenie to wyrażenie, które ma być obliczane dla każdego elementu.

element to zmienna, która przyjmuje wartość każdego elementu w sekwencji.

sekwencja to oryginalna sekwencja lub iterowalny obiekt, z którego pobierane są elementy.

warunek (opcjonalny) to warunek, którym musi spełniać element, aby został uwzględniony w nowej liście.

Programowanie funkcyjne w Pythonie

List comprehensions:

```
numbers = [1, 2, 3, 4, 5]
squares = [x**2 for x in numbers]
# Wynik: [1, 4, 9, 16, 25]
```

```
numbers = [1, 2, 3, 4, 5]
even_numbers = [x for x in numbers if x % 2 == 0]
# Wynik: [2, 4]
```

```
words = ["jabłko", "banan", "wiśnia", "truskawka", "ananas"]
long_words = [word for word in words if len(word) > 5]
# Wynik: ['jabłko', 'wiśnia', 'truskawka', 'ananas']
```

```
list1 = [1, 2, 3]
list2 = [10, 20, 30]
pairs = [(x, y) for x in list1 for y in list2]
# Wynik: [(1, 10), (1, 20), (1, 30), (2, 10), (2, 20), (2, 30), (3, 10), (3, 20), (3, 30)]
```

Programowanie funkcyjne w Pythonie

Przekazywanie argumentów do metod:

- przez pozycję

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

wynik = dodaj(3, 5)  # Przekazywane są wartości 3 i 5 do argumentów a i b.
```

Programowanie funkcyjne w Pythonie

Przekazywanie argumentów do metod:

- przez nazwę

```
def odejmij(a, b):
    return a - b

wynik = odejmij(b=5, a=3)  # Przekazywane są argumenty a=3 i b=5.
```

Programowanie funkcyjne w Pythonie

Przekazywanie argumentów do metod:

- przekazywanie argumentów domyślnych

```
def powitanie(imie="Anonim"):
    return f"Witaj, {imie}!"

wynik = powitanie()      # Wynik: "Witaj, Anonim!"
wynik2 = powitanie("Jan")  # Wynik: "Witaj, Jan!"
```

Programowanie funkcyjne w Pythonie

Przekazywanie argumentów do metod:

rozpakowywanie argumentów:

- można użyć operatora * lub ** do rozpakowania argumentów z listy lub słownika i przekazania ich do funkcji.

```
def dodaj(a, b, c):
    return a + b + c

argumenty = [1, 2, 3]
wynik = dodaj(*argumenty)  # Rozpakowanie
#listy argumentów.
# Wynik: 6
```

```
def informacje(imie, wiek):
    return f"Imie: {imie}, Wiek: {wiek}"

dane = {"imie": "Jan", "wiek": 30}
wynik = informacje(**dane)  # Rozpakowanie
#słownika argumentów.
# Wynik: "Imie: Jan, Wiek: 30"
```

Programowanie funkcyjne w Pythonie

Przekazywanie argumentów do metod:

*args (gwiazdka przed "args"):

- *args pozwala na przekazywanie dowolnej liczby argumentów pozycyjnych do funkcji. Argumenty te są zbierane w krotkę, co oznacza, że są indeksowane od 0. Jest to przydatne, gdy nie wiadomo, ile argumentów zostanie przekazanych do funkcji.

```
def suma(*args):
    total = 0
    for arg in args:
        total += arg
    return total

wynik = suma(1, 2, 3, 4, 5)
# Wynik: 15
```

Programowanie funkcyjne w Pythonie

Przekazywanie argumentów do metod:

**kwargs:

- **kwargs pozwala na przekazywanie dowolnej liczby argumentów nazwanych w formie słownika (klucz-wartość) do funkcji. Jest to przydatne, gdy funkcja ma obsługiwać różne argumenty, z których część może być opcjonalna.

```
def informacje(**kwargs):
    for klucz, wartosc in kwargs.items():
        print(f"{klucz}: {wartosc}")

dane = {"imie": "Jan", "wiek": 30, "miasto": "Warszawa"}
informacje(**dane)
# Wynik:
# imie: Jan
# wiek: 30
# miasto: Warszawa
```

Programowanie funkcyjne w Pythonie

Przekazywanie argumentów do metod:

łączenie różnych metod

```
def kompleksowa_funkcja(a, b, *args, c=0, **kwargs):
    wynik = a + b + c
    for arg in args:
        wynik += arg
    for klucz, wartosc in kwargs.items():
        wynik += wartosc
    return wynik

# Wywołanie funkcji z różnymi rodzajami argumentów
print(kompleksowa_funkcja(1, 2, 3, 4, c=5, x=6, y=7))
# Wynik: 28
```

Dekoratory w Pythonie

Dekoratory:

- Dekoratory w języku Python to specjalne funkcje, które pozwalają na modyfikowanie innych funkcji lub metod w deklaratywny sposób. Są one szeroko używane w Pythonie do rozszerzania funkcjonalności istniejącego kodu, np. dodawania logiki, walidacji danych, pomiaru czasu wykonywania funkcji itp.

Dekoratory w Pythonie

Tworzenie dekoratorów:

- Dekoratory w języku Python to specjalne funkcje, które pozwalają na modyfikowanie innych funkcji lub metod w deklaratywny sposób. Są one szeroko używane w Pythonie do rozszerzania funkcjonalności istniejącego kodu, np. dodawania logiki, walidacji danych, pomiaru czasu wykonywania funkcji itp.

```
def dekorator(funkcja_do_udekorowania):
    def nowa_funkcja(*args, **kwargs):
        # Dodaj logikę przed wywołaniem funkcji_do_udekorowania
        wynik = funkcja_do_udekorowania(*args, **kwargs)
        # Dodaj logikę po wywołaniu funkcji_do_udekorowania
        return wynik
    return nowa_funkcja
```

Dekoratory w Pythonie

Przykłady zastosowania dekoratorów:

Dekorator logujący:

- Pozwala na zapisywanie logów przed i po wywołaniu funkcji.

```
def logowanie(funkcja):
    def zaloguj(*args, **kwargs):
        print(f"Wywoływanie funkcji: {funkcja.__name__}")
        wynik = funkcja(*args, **kwargs)
        print(f"Zakończono funkcję: {funkcja.__name__}")
        return wynik
    return zaloguj

@logowanie
def dodaj(a, b):
    return a + b
```

Dekoratory w Pythonie

Przykłady zastosowania dekoratorów:

Dekorator pomiaru czasu:

- Pomaga w mierzeniu czasu wykonywania funkcji.

```
import time
def pomiar_czasu(funkcja):
    def zmierz_czas(*args, **kwargs):
        start = time.time()
        wynik = funkcja(*args, **kwargs)
        koniec = time.time()
        print(f"Czas wykonania {funkcja.__name__}: {koniec - start} sekund")
        return wynik
    return zmierz_czas

@pomiar_czasu
def silnia(n):
    if n == 0:
        return 1
    else:
        return n * silnia(n - 1)
```

Dekoratory w Pythonie

Popularne dekoratory:

W środowisku Python istnieje wiele popularnych bibliotek, które dostarczają gotowe dekoratory. Niektóre z nich to:

- @classmethod i @staticmethod: Dekoratory wbudowane w Pythona, które służą do oznaczania metod w klasach jako metody klasy lub metody statyczne.
- @property: Pozwala na dostęp do atrybutów obiektów jak do atrybutów klasy, a jednocześnie pozwala na dodanie dodatkowej logiki (getter) podczas odczytywania wartości atrybutu.
- @abstractmethod: Z dekoratorem @abstractmethod można tworzyć abstrakcyjne klasy i wymagać od dziedziczących klas implementacji określonych metod.

Dekoratory w Pythonie

Popularne dekoratory:

W środowisku Python istnieje wiele popularnych bibliotek, które dostarczają gotowe dekoratory. Niektóre z nich to:

- @login_required (np. w Django): W przypadku aplikacji internetowych, ten dekorator wymaga, aby użytkownik był zalogowany, zanim będzie mógł uzyskać dostęp do danej strony lub funkcji.
- @app.route (np. w Flask): Pozwala na mapowanie funkcji na konkretne adresy URL, co jest używane w mikroframeworku Flask do obsługi różnych stron internetowych.
- @caching: Dekoratory do buforowania (caching) wyników funkcji, co pozwala na przyspieszenie wywołań funkcji oznaczonej jako kosztowna operacja.

ZAAWANSOWANE TECHNIKI PROGRAMOWANIA

DR INŻ. MARCIN KULIK
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI
KATEDRA AUTOMATYZACJI NAPĘDÓW I ROBOTYKI

Dokumentowanie kodu w Pythonie

Dokumentowanie kodu w Pythonie

Komentarze w kodzie

- Komentarze do wyjaśnienia trudnych fragmentów kodu, trudnych decyzji projektowych lub innych szczegółów.
- Komentarze powinny być krótkie, klarowne i zrozumiałe.

```
# Przykład komentarza:
x = 10  # Inicjalizacja zmiennej x
```

Dokumentowanie kodu w Pythonie

Docstringi:

- Dla modułów, klas i funkcji używa się docstringów, czyli wielolinijkowych komentarzy umieszczonych pod definicją.
- Docstringi pozwalają na generowanie dokumentacji automatycznie.

```
def dodaj(a, b):
    """
    Dodaje dwie liczby.

    :param a: Pierwsza liczba
    :param b: Druga liczba
    :return: Wynik dodawania
    """
    return a + b
```

Dokumentowanie kodu w Pythonie

Nazwy zmiennych i funkcji:

- Nadawaj zmiennym, funkcjom i klasom znaczące i opisowe nazwy.
- Unikaj nazw takich jak a, x, foo, które są mało opisowe.

```
# Zła nazwa zmiennej:
a = 5

# Dobra nazwa zmiennej:
ilosc_elementow = 5
```

Dokumentowanie kodu w Pythonie

Tworzenie pliku readme.md:

- readme.md to plik Markdown, który służy do opisania projektu.
- Opisz krótko, o co chodzi w projekcie, jak go zainstalować i uruchomić, oraz jakie są główne funkcje i zależności.

```
# Nazwa Projektu
Krótki opis projektu.
## Instalacja
1. Sklonuj repozytorium: `git clone https://github.com/twoj-repozytorium.git`
2. Wejdź do katalogu projektu: `cd twoj-repozytorium`
3. Zainstaluj zależności: `pip install -r requirements.txt`
## Użycie
Opisz, jak uruchomić projekt i jak go używać.
## Dokumentacja
Link do dokumentacji projektu lub opis funkcji w projekcie.
## Autor
Twój nazwisko lub pseudonim
## Licencja
To jest miejsce na informacje o licencji projektu.
```

Dokumentowanie kodu w Pythonie

Unikaj Zbędnego Komentowania:

- Nie komentuj oczywistych lub samodokumentujących się fragmentów kodu.

```
# Zły komentarz:
i = i + 1  # Zwiększamy zmienną i o 1

# Dobry kod:
i += 1
```

Dokumentowanie kodu w Pythonie

Pydoc:

- Pydoc to narzędzie do generowania dokumentacji Pythona dostępne w standardowej bibliotece Pythona. Pozwala ono na łatwe uzyskanie informacji na temat modułów, klas, funkcji i metod Pythona, włączając w to docstringi, które są częścią kodu źródłowego. Pydoc generuje dokumentację w formie tekstu lub jako stronę HTML.

Dokumentowanie kodu w Pythonie

Pydoc w konsoli:

- W terminalu lub wierszu poleceń wpisz polecenie **pydoc** razem z nazwą modułu, klasy, funkcji lub metody, której dokumentację chcesz przeglądać.
- Przykłady:

  Aby uzyskać dokumentację wbudowanej funkcji **print**, wpisz **pydoc print**.

  Aby uzyskać dokumentację modułu **math**, wpisz **pydoc math**.

  Pydoc otworzy nowe okno konsoli lub przeglądarki, w którym wyświetli dokumentację.

Dokumentowanie kodu w Pythonie

Moduł Pydoc w skrypcie Pythona:

```
import pydoc

# Wygeneruj dokumentację dla funkcji `print`
pydoc.doc(print)
```

Dokumentowanie kodu w Pythonie

Dokumentacja własnych modułów i funkcji:

Aby korzystać z Pydoc do dokumentowania własnych modułów, klas, funkcji i metod, należy umieścić docstringi w odpowiednich miejscach w kodzie. Docstringi powinny znajdować się na początku modułu, klasy, funkcji lub metody.

```
pydoc nazwa_modulu

pydoc -b nazwa_modulu

(Python -m pydoc)
```

```
def dodaj(a, b):
    """
    Dodaje dwie liczby.

    :param a: Pierwsza liczba
    :param b: Druga liczba
    :return: Wynik dodawania
    """
    return a + b
```

Dokumentowanie kodu w Pythonie

readme.md w formacie Markdown

- Ważna część projektowania i dokumentacji projektu na platformie GitHub i wielu innych miejscach. Markdown jest prostym językiem znaczników, który umożliwia formatowanie tekstu w czytelny sposób.

Dokumentowanie kodu w Pythonie

readme.md w formacie Markdown

Nagłówki:

- Nagłówki w Markdown oznacza się za pomocą znaku #. Im więcej znaków #, tym mniejszy nagłówek.

```
# Nagłówek Poziomu 1
## Nagłówek Poziomu 2
### Nagłówek Poziomu 3
```

Dokumentowanie kodu w Pythonie

readme.md w formacie Markdown

Tekst Pogrubiony i Kursywa::

- Tekst pogrubiony oznacza się dwoma gwiazdkami **tekst** lub dwoma podkreśleniami __tekst__.
- Tekst kursywą oznacza się jedną gwiazdką *tekst* lub jednym podkreśleniem _tekst_.

Dokumentowanie kodu w Pythonie

readme.md w formacie Markdown

Lista punktowana i lista numerowana:

- Listę punktowaną oznacza się znakiem -, + lub *, a każdy element zaczyna się od spacji.
- Listę numerowaną oznacza się cyframi z kropką i spacją.

```
- Element 1
- Element 2
```

```
1. Element 1
2. Element 2
```

Dokumentowanie kodu w Pythonie

readme.md w formacie Markdown

Linki i obrazy:

- Linki można tworzyć za pomocą [tekst_linka](adres_url).
- Wstawianie obrazów jest podobne do linków,

ale z wykorzystaniem wykrzykników ![tekst_alternatywny](adres_obrazu).

```
[GitHub](https://github.com)
```

```
![Logo GitHub](https:///images/GitHub-Mark.png)
```

Dokumentowanie kodu w Pythonie

readme.md w formacie Markdown

Kod źródłowy:

- Aby wyróżnić kod źródłowy, użyj trzech backticków ``` przed i po kodzie.

````
```python
def funkcja():
    return "Hello, World!"
```
````

Dokumentowanie kodu w Pythonie

readme.md w formacie Markdown

Hiperłącza:

- Aby utworzyć hiperłącza wewnętrzne w dokumencie, użyj # i nazwy nagłówka.

```
[Link do nagłówka 1](#nagłówek-poziomu-1)
```

Dokumentowanie kodu w Pythonie

readme.md w formacie Markdown

Linie Oddzielające:

- Aby utworzyć poziomą linię oddzielającą, użyj trzech myślników --- lub trzech gwiazdek ***.

Dokumentowanie kodu w Pythonie

readme.md w formacie Markdown

README.md > # Nazwa Projektu > ## Licencja

```
# Nazwa Projektu
Krótki opis projektu.
## Instalacja
1. Sklonuj repozytorium: `git clone https://github.com/twoje_repozytorium.git`
2. Wejdź do katalogu projektu: `cd twoje-repozytorium`
3. Zainstaluj zależności: `pip install -r requirements.txt`
## Użycie
Opisz, jak uruchomić projekt i jak go używać.
- 1 punkt
- 2 punkt
3. punkt
4. punkt

```python
def funkcja():
    return "Hello, World!"
```
Dokumentacja
Link do dokumentacji projektu lub opis funkcji w projekcie.
Autor
Twoje nazwisko lub pseudonim
Licencja
To jest miejsce na informacje o licencji projektu
```

Kontrola wersji - GitHub
Projekt w Pythonie

Dobre praktyki tworzenia repozytorium na GitHub
Projekt w Pythonie

- **Opis i dokumentacja**: Dodaj opis repozytorium, który jasno opisuje cel projektu. Twórz czytelną dokumentację projektu w pliku README.md. Opisz, jak zainstalować, skonfigurować i korzystać z projektu. Dodaj plik requirements.txt
- **.gitignore**: Dodaj odpowiedni plik .gitignore, aby wykluczyć pliki tymczasowe, pliki kompilowane i inne pliki, które nie powinny być śledzone przez system kontroli wersji Git. Gotowe szablony plików .gitignore dla różnych języków, frameworków i narzędzi programistycznych można znaleźć na stronie GitHub Gitignore Repository (https://github.com/github/gitignore). To repozytorium zawiera wiele gotowych plików .gitignore dostosowanych do konkretnych środowisk programistycznych.

Dobre praktyki tworzenia repozytorium na GitHub
Projekt w Pythonie

- **Branches**: Używaj różnych gałęzi (branches) do rozwoju i eksperymentowania z kodem. Zachowuj główną gałąź (zazwyczaj nazywaną "main" lub "master") stabilną.
- **Commit messages**: Pisz jasne i opisowe komunikaty commitów, które wyjaśniają, co zostało zmienione. Używaj j. angielskiego.
- **Regularne aktualizacje**: Aktualizuj repozytorium regularnie.
- **Issues** i **Pull Requests**: Korzystaj z systemu zgłaszania problemów (Issues) i próśb o przypisanie zmian (Pull Requests) do dyskusji i współpracy z innymi programistami.

Tworzenie wirtualnego środowiska

Środowisko wirtualne w kontekście programowania Python to izolowane i niezależne od siebie środowisko, w którym można instalować i zarządzać pakietami oraz bibliotekami Pythona. Głównym celem jest izolacja projektów od siebie, co pozwala na uniknięcie konfliktów wersji pakietów i zapewnia spójność zależności w projekcie. Oto kilka głównych zalet korzystania z wirtualnych środowisk:

- Izolacja projektów: Każdy projekt może korzystać z własnego, niezależnego środowiska wirtualnego, co oznacza, że pakiety i biblioteki używane w jednym projekcie nie będą wpływać na inne projekty.
- Zarządzanie wersjami: Można kontrolować wersje pakietów i bibliotek używanych w danym projekcie, co pozwala na zachowanie spójności i uniknięcie konfliktów wersji.

Tworzenie wirtualnego środowiska

- Czystość i porządek: Dzięki wykorzystaniu wirtualnych środowisk, projekt nie zostawia "śmieci" w systemie lub globalnym interpreterze Pythona.
- Wymienność: Można udostępnić informacje o środowisku w postaci pliku requirements.txt, dzięki czemu inni programiści mogą łatwo odtworzyć środowisko projektu na swoich komputerach.
- Testowanie i rozwijanie oprogramowania: ułatwia testowanie kodu na różnych wersjach interpretera Pythona i różnych konfiguracjach środowisk.
- Elastyczność: Można korzystać z różnych interpreterów Pythona (np. Python 3.7, 3.8, 3.9 itp.) w różnych wirtualnych środowiskach.

Tworzenie wirtualnego środowiska

venv:

- **venv** jest modułem zawartym w standardowej instalacji Pythona
- Aby utworzyć wirtualne środowisko, należy w folderze z projektem otworzyć wiersz poleceń i zastosować komendę:

  python -m venv myenv

  gdzie myenv to nazwa stworzonego wirtualnego środowiska
- To polecenie utworzy katalog o nazwie "myenv" w bieżącym katalogu i zainstaluje w nim osobną kopię Pythona oraz narzędzia pip i setuptools.

Tworzenie wirtualnego środowiska

venv:

- aktywacja środowiska komendą: myenv\Scripts\activate
  (macOS i Linux: source myenv/bin/activate)
- po aktywacji wirtualnego środowiska, można uruchamiać w nim projekty oraz instalować dodatkowe moduły z wykorzystaniem narzędzia **pip**
- dezaktywacja wirtualnego środowiska: aby zakończyć pracę w wirtualnym środowisku, wystarczy wpisać polecenie **deactivate**

ZAAWANSOWANE TECHNIKI PROGRAMOWANIA

DR INŻ. MARCIN KULIK
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI I INFORMATYKI
KATEDRA AUTOMATYZACJI NAPĘDÓW I ROBOTYKI

Obsługa wyjątków

Obsługa wyjątków w Pythonie

Obsługa wyjątków w języku Python pozwala na reagowanie na błędy i nieprzewidziane sytuacje w trakcie wykonywania programu.

- Blok **try**: wewnątrz bloku try umieszczamy kod, który może generować wyjątki.
- Blok **except**: pozwala obsługiwać wyjątki, które mogą zostać wygenerowane w bloku try. Możemy określić konkretne rodzaje wyjątków, które chcemy obsłużyć, lub użyć ogólnego bloku except bez określenia konkretnego typu wyjątku.
- Blok **else** (opcjonalny): jest wykonywany, jeśli w bloku try nie zostanie wygenerowany wyjątek.
- Blok **finally** (opcjonalny): zawiera kod, który zostanie zawsze wykonany, niezależnie od tego, czy wystąpił wyjątek czy nie.

Obsługa wyjątków w Pythonie

Przykłady obsługi wyjątków:

```
# Przykład 1: Obsługa ogólnego wyjątku
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Dzielenie przez zero!")
except:
    print("Inny wyjątek")
```

```
# Przykład 2: Obsługa konkretnego wyjątku
try:
    num = int("abc")
except ValueError:
    print("Nie można przekonwertować na liczbę")
```

Obsługa wyjątków w Pythonie

Przykłady obsługi wyjątków:

```
# Przykład 3: Użycie bloku else
try:
    file = open("plik.txt", "r")
except FileNotFoundError:
    print("Plik nie istnieje")
else:
    print("Plik został otwarty")
```

```
# Przykład 4: Blok finally
try:
    f = open("plik.txt", "r")
    # Tu można pracować z plikiem
except FileNotFoundError:
    print("Plik nie istnieje")
finally:
    f.close()  # Zawsze zamykamy plik, nawet jeśli wystąpi błąd
```

Obsługa wyjątków w Pythonie

Przykłady obsługi wyjątków:

```
# Przykład 4: Obsługa błędu braku dostępu do zasobu sieciowego, np. API.
import requests

try:
    response = requests.get("https://example.com/api/data")
    response.raise_for_status() #Wywołanie tej metody zgłosi wyjątek, jeśli otrzymamy kod błędu
HTTP
except requests.exceptions.RequestException as e:
    print(f"Błąd podczas próby pobrania danych: {e}")
else:
    data = response.json()
    print(f"Pobrane dane: {data}")
```

Obsługa wyjątków w Pythonie

Przykłady obsługi wyjątków:

```
# Przykład 5: Obsługa błędu podczas konwersji danych.
try:
    user_input = input("Podaj liczbę całkowitą: ")
    number = int(user_input)
except ValueError:
    print("Nieprawidłowy format liczby")
else:
    result = number * 2
    print(f"Wynik: {result}")
```

Obsługa wyjątków w Pythonie

Przykłady obsługi wyjątków:

```
try:
    # Próba otwarcia pliku do odczytu
    file = open("plik.txt", "r")
except FileNotFoundError:
    print("Plik nie istnieje")
else:
    try:
        # Odczytanie zawartości pliku
        data = file.read()
    except Exception as e:
        print(f"Błąd odczytu pliku: {e}")
    else:
        print("Zawartość pliku:")
        print(data)
    finally:
        file.close()  # Zawsze zamykamy plik, niezależnie od tego, czy wystąpił błąd odczytu
finally:
    print("Koniec programu")
```

Obsługa plików w Pythonie

Obsługa plików w Pythonie

W Pythonie obsługa plików jest bardzo prosta i intuicyjna. Możemy odczytywać, zapisywać oraz modyfikować np. pliki tekstowe, binarne. Python oferuje wbudowane funkcje, które ułatwiają zarządzanie plikami. W Pythonie pliki otwieramy za pomocą funkcji open(), która umożliwia otwarcie pliku w różnych trybach:

'r' Odczyt (domyślny tryb). Plik musi istnieć.

'w' Zapis. Tworzy nowy plik lub nadpisuje istniejący.

'a' Dopisywanie. Dodaje nowe dane na końcu pliku, nie usuwa istniejących.

'x' Tworzy nowy plik. Jeśli plik już istnieje, zgłasza błąd.

'b' Tryb binarny.

't' Tryb tekstowy (domyślny).

'+' Odczyt i zapis jednocześnie.

Obsługa plików w Pythonie

Zawsze zamykaj pliki po zakończeniu operacji, używając close(), lub lepiej, używaj bloku **with**, który automatycznie zamyka plik:

```
with open('plik.txt', 'r') as plik:
    zawartosc = plik.read()
```

Czytanie i zapisywanie plików:

read(): Odczytuje cały plik.

readline(): Odczytuje jedną linię z pliku.

readlines(): Zwraca listę wszystkich linii w pliku.

write(): Zapisuje tekst do pliku.

writelines(): Zapisuje listę linii do pliku.

Obsługa plików w Pythonie

Używaj bloku try-except do obsługi wyjątków związanych z plikami, takich jak FileNotFoundError:

```
try:
    with open('nieistniejacy_plik.txt', 'r') as plik:
        zawartosc = plik.read()
except FileNotFoundError:
    print("Plik nie istnieje")
```

Obsługa plików w Pythonie

Najlepsze biblioteki do obsługi różnych typów plików

Obsługa plików tekstowych:

- Wbudowane funkcje Pythona: open(), read(), write(), writelines().

Pliki CSV i Excel:

- pandas: Najbardziej popularna biblioteka do pracy z plikami tabelarycznymi (CSV, Excel).

  Odczyt pliku CSV: pd.read_csv('plik.csv').

  Zapis pliku Excel: df.to_excel('plik.xlsx').

- openpyxl: Obsługuje pliki Excel w formacie .xlsx.

Pliki JSON:

- json (moduł wbudowany w Pythona): Służy do odczytu i zapisu danych w formacie JSON.

Obsługa plików w Pythonie

Pliki PDF:

- PyPDF2: Do odczytu, modyfikacji i łączenia plików PDF.

  Odczyt PDF: pdf_reader = PyPDF2.PdfReader(open('plik.pdf', 'rb')).

- ReportLab: Do generowania plików PDF.

Pliki graficzne:

- Pillow (PIL): Biblioteka do obsługi i przetwarzania obrazów (JPEG, PNG, GIF, BMP, TIFF).

  Otwieranie obrazu: Image.open('obraz.jpg').

  Zapis obrazu: obraz.save('nowy_obraz.png').

Obsługa plików w Pythonie

Pliki audio:

- pydub: Obsługa plików audio w formatach MP3, WAV, OGG.

  Otwieranie pliku audio: audio = AudioSegment.from_file('plik.mp3').

  Zapis pliku audio: audio.export('plik.wav', format='wav').

Pliki wideo:

- OpenCV: Obsługa i przetwarzanie wideo oraz obrazów.

  Odczyt wideo: video = cv2.VideoCapture('plik.mp4').

Archiwa ZIP:

- zipfile (moduł wbudowany w Pythona): Do pracy z plikami ZIP.

Obsługa plików w Pythonie

Podsumowanie:

Python oferuje szeroką gamę narzędzi i bibliotek do obsługi różnorodnych formatów plików. Wybór odpowiednich narzędzi zależy od typu pliku, z jakim się pracuje. Moduły wbudowane, takie jak open(), json, csv, oraz biblioteki zewnętrzne, takie jak pandas, PyPDF2, Pillow, OpenCV, zapewniają prostą i efektywną obsługę plików w różnych formatach.