# *Python dla Ekspertów*

***Kevin Clarkson***

## Spis treści

# 1. Zaawansowane aspekty składni Pythona

## 1.1. Dekoratory i metaklasy

Dekoratory można łączyć, stosując je jeden po drugim. Kolejność ma znaczenie - dekoratory są aplikowane od dołu do góry.

```
Copydef dekorator1(func):
def wrapper(*args, **kwargs):
print("Dekorator 1")
return func(*args, **kwargs)
return wrapper
def dekorator2(func):
def wrapper(*args, **kwargs):
print("Dekorator 2")
return func(*args, **kwargs)
return wrapper
@dekorator1
@dekorator2
def moja_funkcja():
print("Moja funkcja")
moja_funkcja()
# Wynik:
# Dekorator 1
# Dekorator 2
# Moja funkcja
```

Wbudowane dekoratory

Python oferuje kilka wbudowanych dekoratorów, które są często używane:

1. @property

Dekorator @property zamienia metodę klasy w atrybut tylko do odczytu:

```
Copyclass Osoba:
def __init__(self, imie, nazwisko):
self._imie = imie
self._nazwisko = nazwisko
@property
def pelne_imie(self):
return f"{self._imie} {self._nazwisko}"
osoba = Osoba("Jan", "Kowalski")
print(osoba.pelne_imie)  # Jan Kowalski
```

2. @classmethod

Dekorator @classmethod tworzy metodę klasy, która otrzymuje klasę jako pierwszy argument zamiast instancji:

```
Copyclass Osoba:
licznik = 0
def __init__(self, imie):
self.imie = imie
Osoba.licznik += 1
@classmethod
def liczba_osob(cls):
return cls.licznik
print(Osoba.liczba_osob())  # 0
osoba1 = Osoba("Jan")
osoba2 = Osoba("Anna")
print(Osoba.liczba_osob())  # 2
```

3. @staticmethod

Dekorator @staticmethod tworzy metodę statyczną, która nie otrzymuje ani instancji, ani klasy jako pierwszy argument:

```
Copyclass Matematyka:
@staticmethod
def dodaj(a, b):
return a + b
print(Matematyka.dodaj(3, 5))  # 8
```

Funkcje jako dekoratory klas

Funkcje mogą być również używane jako dekoratory klas. Pozwala to na modyfikację lub rozszerzenie funkcjonalności całej klasy:

```
Copydef dodaj_metode(cls):
def nowa_metoda(self):
return "To jest nowa metoda"
cls.nowa_metoda = nowa_metoda
return cls
@dodaj_metode
class MojaKlasa:
pass
obiekt = MojaKlasa()
print(obiekt.nowa_metoda())  # To jest nowa metoda
```

W tym przykładzie dekorator `dodaj_metode` dodaje nową metodę do dekorowanej klasy. Funkcja dekoratora przyjmuje klasę jako argument, modyfikuje ją i zwraca zmodyfikowaną klasę.

Metaklasy to "klasy klas" w Pythonie. Są one odpowiedzialne za tworzenie i kontrolowanie zachowania klas. Każda klasa w Pythonie jest instancją metaklasy, domyślnie type.

Podstawowa składnia definiowania klasy z użyciem metaklasy wygląda następująco:

```
Copyclass MojaMetaklasa(type):
def __new__(cls, name, bases, attrs):
# Modyfikacja lub dodanie atrybutów
return super().__new__(cls, name, bases, attrs)
class MojaKlasa(metaclass=MojaMetaklasa):
pass
```

Metoda `__new__` metaklasy jest wywoływana podczas tworzenia klasy i może modyfikować jej definicję przed utworzeniem.

Customowe metaklasy

Customowe metaklasy pozwalają na zaawansowaną kontrolę nad procesem tworzenia i zachowaniem klas. Oto przykład metaklasy, która automatycznie dodaje metodę do każdej klasy:

```
Copyclass DodajMetodeMetaklasa(type):
def __new__(cls, name, bases, attrs):
def nowa_metoda(self):
return f"Jestem instancją klasy {name}"
attrs['nowa_metoda'] = nowa_metoda
return super().__new__(cls, name, bases, attrs)
class MojaKlasa(metaclass=DodajMetodeMetaklasa):
pass
obiekt = MojaKlasa()
print(obiekt.nowa_metoda())  # Jestem instancją klasy MojaKlasa
```

Metaklasy mogą również implementować metody `__init__` i `__call__`, które są wywoływane odpowiednio po utworzeniu klasy i podczas tworzenia instancji klasy.

Dziedziczenie metaklas

Metaklasy mogą dziedziczyć po sobie, co pozwala na tworzenie hierarchii metaklas i dzielenie funkcjonalności:

```
Copyclass MetaklasaBazowa(type):
def __new__(cls, name, bases, attrs):
attrs['metoda_bazowa'] = lambda self: "Metoda z metaklasy bazowej"
```

```
    return super().__new__(cls, name, bases, attrs)
    class MetaklasaPochodna(MetaklasaBazowa):
    def __new__(cls, name, bases, attrs):
    attrs['metoda_pochodna'] = lambda self: "Metoda z metaklasy
pochodnej"
    return super().__new__(cls, name, bases, attrs)
    class MojaKlasa(metaclass=MetaklasaPochodna):
    pass
    obiekt = MojaKlasa()
    print(obiekt.metoda_bazowa())    # Metoda z metaklasy
bazowej
    print(obiekt.metoda_pochodna())  # Metoda z metaklasy
pochodnej
```

W tym przykładzie `MetaklasaPochodna` dziedziczy po `MetaklasaBazowej`, co pozwala na dodanie metod z obu metaklas do klasy `MojaKlasa`.

Dziedziczenie metaklas jest szczególnie przydatne, gdy chcemy rozszerzyć funkcjonalność istniejącej metaklasy lub stworzyć hierarchię metaklas z różnymi poziomami modyfikacji klas.

1. Modyfikacja procesu tworzenia klasy

Metaklasy pozwalają na ingerencję w proces tworzenia klasy, umożliwiając modyfikację atrybutów, metod lub dodanie nowych elementów przed finalnym utworzeniem klasy.

```
    Copyclass DodajPrefixMetaklasa(type):
    def __new__(cls, name, bases, attrs):
    prefixed_attrs = {f"my_{key}": value for key, value in attrs.items()
if not key.startswith('__')}
    return super().__new__(cls, name, bases, prefixed_attrs)
    class MojaKlasa(metaclass=DodajPrefixMetaklasa):
    x = 1
    def metoda(self):
    pass
    print(MojaKlasa.my_x)  # 1
    print(hasattr(MojaKlasa, 'my_metoda'))  # True
```

2. Automatyczne rejestrowanie klas

Metaklasy mogą być używane do automatycznego rejestrowania klas w centralnym rejestrze, co jest przydatne w przypadku systemów wtyczek lub mapowania klas.

```
Copyclass RejestrKlas:
klasy = {}
@classmethod
def zarejestruj(cls, nazwa, klasa):
cls.klasy[nazwa] = klasa
class AutoRejestrMetaklasa(type):
def __new__(cls, name, bases, attrs):
nowa_klasa = super().__new__(cls, name, bases, attrs)
RejestrKlas.zarejestruj(name, nowa_klasa)
return nowa_klasa
class KlasaA(metaclass=AutoRejestrMetaklasa):
pass
class KlasaB(metaclass=AutoRejestrMetaklasa):
pass
print(RejestrKlas.klasy)  # {'KlasaA': <class '__main__.KlasaA'>,
'KlasaB': <class '__main__.KlasaB'>}
```

3. Implementacja wzorców projektowych

Metaklasy mogą być wykorzystane do implementacji wzorców projektowych, takich jak Singleton.

```
Copyclass SingletonMetaklasa(type):
_instances = {}
def __call__(cls, *args, **kwargs):
if cls not in cls._instances:
cls._instances[cls] = super().__call__(*args, **kwargs)
return cls._instances[cls]
class Singleton(metaclass=SingletonMetaklasa):
pass
s1 = Singleton()
s2 = Singleton()
print(s1 is s2)  # True
```

Metaklasy w kontekście dziedziczenia wielokrotnego

W przypadku dziedziczenia wielokrotnego, gdzie klasy bazowe mają różne metaklasy, Python stosuje złożone reguły rozwiązywania konfliktów metaklas. Ogólna zasada jest taka, że metaklasa musi być podklasą wszystkich metaklas klas bazowych.

```
Copyclass MetaklasaA(type):
pass
class MetaklasaB(type):
pass
```

```
class MetaklasaC(MetaklasaA, MetaklasaB):
pass
class A(metaclass=MetaklasaA):
pass
class B(metaclass=MetaklasaB):
pass
class C(A, B, metaclass=MetaklasaC):
pass
```

W powyższym przykładzie, `MetaklasaC` dziedziczy po `MetaklasaA` i `MetaklasaB`, co pozwala na użycie jej jako metaklasy dla klasy `C`, która dziedziczy po `A` i `B`.

Jeśli nie zostanie dostarczona odpowiednia metaklasa, Python zgłosi błąd, informując o niemożliwości rozwiązania konfliktu metaklas.

Metaklasy w kontekście dziedziczenia wielokrotnego wymagają starannego planowania i zrozumienia hierarchii klas oraz metaklas, aby uniknąć konfliktów i zapewnić poprawne działanie kodu.

## 1.2. Generatory i korutyny

Generatory w Pythonie mogą nie tylko produkować wartości, ale także je przyjmować za pomocą metody `send()`. Pozwala to na dwukierunkową komunikację między kodem wywołującym a generatorem.

```
Copydef echo_generator():
while True:
value = yield
print(f"Otrzymano: {value}")
gen = echo_generator()
next(gen)  # Inicjalizacja generatora
gen.send("Hello")  # Wysyłanie wartości do generatora
gen.send("World")
# Wynik:
# Otrzymano: Hello
# Otrzymano: World
```

Metoda `send()` wznawia wykonanie generatora i wysyła wartość do ostatniego wyrażenia `yield`. Wartość ta staje się wynikiem wyrażenia `yield`.

Zamykanie generatorów (metody close() i throw())

Generatory można jawnie zamknąć za pomocą metody `close()`. Wywołanie tej metody powoduje zgłoszenie wyjątku `GeneratorExit` w miejscu, gdzie generator jest wstrzymany.

```
Copydef closable_generator():
try:
while True:
yield "Jestem aktywny"
finally:
print("Generator został zamknięty")
gen = closable_generator()
print(next(gen))
gen.close()
# Wynik:
# Jestem aktywny
# Generator został zamknięty
```

Metoda `throw()` pozwala na zgłoszenie dowolnego wyjątku w miejscu, gdzie generator jest wstrzymany:

```
Copydef exception_handler_generator():
try:
yield "Normalnie"
yield "Nadal normalnie"
except ValueError:
yield "Złapano ValueError"
gen = exception_handler_generator()
print(next(gen))
print(gen.throw(ValueError))
# Wynik:
# Normalnie
# Złapano ValueError
```

Delegowanie generatorów (yield from)

Wyrażenie `yield from` pozwala na delegowanie części lub całości generacji do innego generatora lub iterowalnego obiektu. Jest to szczególnie przydatne przy tworzeniu złożonych generatorów lub implementacji korutyn.

```
Copydef subgenerator():
```

```
yield 1
yield 2
yield 3
def main_generator():
yield "Start"
yield from subgenerator()
yield "Koniec"
for item in main_generator():
print(item)
# Wynik:
# Start
# 1
# 2
# 3
# Koniec
```

`yield from` nie tylko deleguje generację wartości, ale także obsługuje przesyłanie wartości i wyjątków między wywołującym a subdelegatem:

```
Copydef delegating_generator():
x = yield from subgenerator()
print(f"Subgenerator zwrócił: {x}")
def subgenerator():
while True:
x = yield
if x is None:
return "Zakończono"
print(f"Subgenerator otrzymał: {x}")
g = delegating_generator()
next(g)  # Inicjalizacja
g.send("Hello")
g.send("World")
g.send(None)  # Zakończenie subgeneratora
# Wynik:
# Subgenerator otrzymał: Hello
# Subgenerator otrzymał: World
# Subgenerator zwrócił: Zakończono
```

`yield from` upraszcza tworzenie złożonych generatorów i umożliwia bardziej modularne projektowanie kodu wykorzystującego generatory.

Korutyny to specjalne funkcje w Pythonie, które mogą być wstrzymywane i wznawiane w określonych punktach wykonania. Są one kluczowym elementem programowania asynchronicznego w Pythonie, umożliwiając efektywne zarządzanie współbieżnymi zadaniami bez konieczności używania wielu wątków.

Korutyny definiuje się za pomocą słowa kluczowego `async def`:

```
Copyasync def prosta_korutyna():
print("Rozpoczynam korutynę")
await asyncio.sleep(1)
print("Korutyna zakończona")
```

Słowo kluczowe `await` jest używane wewnątrz korutyny do wstrzymania jej wykonania do czasu zakończenia innej operacji asynchronicznej.

Asynchroniczne generatory

Asynchroniczne generatory łączą koncepcje generatorów i korutyn. Pozwalają na tworzenie asynchronicznych iteratorów, które mogą być używane w pętlach `async for`.

Składnia asynchronicznego generatora:

```
Copyasync def asynchroniczny_generator():
for i in range(3):
await asyncio.sleep(1)
yield i
async def main():
async for item in asynchroniczny_generator():
print(item)
asyncio.run(main())
# Wynik (z 1-sekundowymi przerwami):
# 0
# 1
# 2
```

Asynchroniczne generatory używają `async def` do definicji i `yield` do generowania wartości, ale mogą również używać `await` wewnątrz swojego ciała.

Słowa kluczowe async i await

`async` i `await` to kluczowe słowa wprowadzone w Pythonie 3.5 dla obsługi programowania asynchronicznego.

1. `async`:

* Używane do definiowania korutyn (`async def`)

* Używane do definiowania asynchronicznych menedżerów kontekstu (`async with`)
* Używane w asynchronicznych pętlach (`async for`)

2. `await`:
* Używane wewnątrz korutyn do wstrzymania wykonania i oczekiwania na zakończenie innej operacji asynchronicznej
* Może być używane tylko wewnątrz funkcji zdefiniowanych z `async def`

Przykład użycia `async` i `await`:

```
Copyimport asyncio
async def fetch_data():
print("Rozpoczynam pobieranie danych")
await asyncio.sleep(2)  # Symulacja operacji I/O
print("Dane pobrane")
return "Ważne dane"
async def process_data(data):
print("Rozpoczynam przetwarzanie")
await asyncio.sleep(1)  # Symulacja przetwarzania
print(f"Przetworzono: {data.upper()}")
async def main():
data = await fetch_data()
await process_data(data)
asyncio.run(main())
# Wynik (z odpowiednimi przerwami):
# Rozpoczynam pobieranie danych
# Dane pobrane
# Rozpoczynam przetwarzanie
# Przetworzono: WAŻNE DANE
```

W tym przykładzie, `async def` definiuje korutyny, a `await` jest używane do wstrzymania wykonania korutyny do czasu zakończenia asynchronicznej operacji. `asyncio.run()` jest używane do uruchomienia głównej korutyny i zarządzania pętlą zdarzeń.

Programowanie asynchroniczne z użyciem `async` i `await` pozwala na efektywne zarządzanie wieloma współbieżnymi zadaniami, szczególnie w aplikacjach intensywnie korzystających z operacji I/O.

1. Składnia:
* Generatory: definiowane za pomocą `def` i używają `yield`
* Korutyny: definiowane za pomocą `async def` i używają `await`

2. Cel:
* Generatory: głównie do tworzenia iteratorów i lazy evaluation
* Korutyny: do obsługi asynchronicznych operacji i współbieżności
3. Działanie:
* Generatory: wstrzymują się przy `yield`, zwracając wartość
* Korutyny: wstrzymują się przy `await`, oczekując na zakończenie innej operacji asynchronicznej
4. Wywołanie:
* Generatory: używane w pętlach `for` lub przez `next()`
* Korutyny: uruchamiane przez pętlę zdarzeń lub `await` w innej korutynie

Zastosowania korutyn w programowaniu asynchronicznym

1. Obsługa wielu połączeń sieciowych:

```
Copyasync def handle_connection(connection):
data = await connection.receive()
result = await process_data(data)
await connection.send(result)
```

2. Równoległe zapytania do bazy danych:

```
Copyasync def fetch_user_data(user_ids):
tasks = [db.fetch_user(user_id) for user_id in user_ids]
return await asyncio.gather(*tasks)
```

3. Asynchroniczne operacje na plikach:

```
Copyasync def process_large_file(filename):
async with aiofiles.open(filename, mode='r') as file:
content = await file.read()
return await process_content(content)
```

4. Obsługa długotrwałych zadań w tle:

```
Copyasync def background_task():
while True:
await perform_maintenance()
await asyncio.sleep(3600)  # co godzinę
```

Integracja korutyn z biblioteką asyncio

1. Uruchamianie korutyn:

```
Copyimport asyncio
async def main():
await asyncio.sleep(1)
print("Hello, World!")
asyncio.run(main())
```

2. Równoległe wykonywanie korutyn:

```
Copyasync def main():
task1 = asyncio.create_task(coroutine1())
task2 = asyncio.create_task(coroutine2())
await task1
await task2
```

3. Oczekiwanie na wiele korutyn:

```
Copyasync def main():
results = await asyncio.gather(
coroutine1(),
coroutine2(),
coroutine3()
)
```

4. Obsługa timeoutów:

```
Copyasync def main():
try:
result = await asyncio.wait_for(long_running_task(),
timeout=10.0)
except asyncio.TimeoutError:
print("Operacja przekroczyła limit czasu")
```

5. Synchronizacja asynchroniczna:

```
Copylock = asyncio.Lock()
async def protected_resource():
async with lock:
# dostęp do współdzielonego zasobu
```

6. Asynchroniczne kolejki:

```
Copyqueue = asyncio.Queue()
async def producer():
await queue.put(item)
async def consumer():
item = await queue.get()
```

7. Asynchroniczne konteksty menedżera:

```
Copyasync with aiohttp.ClientSession() as session:
async with session.get('http://example.com') as response:
html = await response.text()
```

Korutyny i asyncio tworzą potężny ekosystem do programowania asynchronicznego w Pythonie, umożliwiając efektywne zarządzanie współbieżnymi zadaniami, szczególnie w aplikacjach intensywnie korzystających z I/O, takich jak serwery sieciowe, aplikacje webowe czy systemy przetwarzające duże ilości danych.

## 1.3. Konteksty wykonania i menedżery kontekstu

Menedżery kontekstu oferują elegancki sposób obsługi wyjątków poprzez metodę `__exit__`. Gdy wyjątek wystąpi wewnątrz bloku `with`, zostaje on przekazany do `__exit__` wraz z typem wyjątku, jego wartością i traceback.

```
Copyclass ExceptionHandler:
def __enter__(self):
print("Wejście do kontekstu")
def __exit__(self, exc_type, exc_value, traceback):
if exc_type is not None:
print(f"Złapano wyjątek: {exc_type.__name__}: {exc_value}")
return True  # Tłumi wyjątek
print("Wyjście z kontekstu")
return False  # Przepuszcza wyjątek (domyślne zachowanie)
with ExceptionHandler():
raise ValueError("Przykładowy błąd")
print("Kontynuacja programu")
```

Tworzenie własnych menedżerów kontekstu

Aby stworzyć własny menedżer kontekstu, należy zaimplementować metody `__enter__` i `__exit__`:

```
Copyclass DatabaseConnection:
def __init__(self, db_name):
self.db_name = db_name
def __enter__(self):
print(f"Łączenie z bazą danych {self.db_name}")
# Tu w rzeczywistości nastąpiłoby nawiązanie połączenia
return self
def __exit__(self, exc_type, exc_value, traceback):
print(f"Zamykanie połączenia z bazą danych {self.db_name}")
# Tu w rzeczywistości nastąpiłoby zamknięcie połączenia
if exc_type:
print(f"Wystąpił błąd: {exc_type.__name__}: {exc_value}")
return False
with DatabaseConnection("MyDB") as db:
print("Wykonywanie operacji na bazie danych")
# Symulacja błędu
raise Exception("Błąd bazy danych")
```

Funkcja contextlib.contextmanager

Moduł `contextlib` dostarcza dekorator `@contextmanager`, który pozwala na tworzenie menedżerów kontekstu za pomocą generatorów, upraszczając proces implementacji:

```
Copyfrom contextlib import contextmanager
@contextmanager
def file_manager(filename, mode):
try:
f = open(filename, mode)
yield f
finally:
f.close()
with file_manager('example.txt', 'w') as file:
file.write('Hello, World!')
```

W tym przykładzie:

1. Kod przed `yield` odpowiada metodzie `__enter__`.
2. `yield` zwraca obiekt, który będzie przypisany do zmiennej po `as`.
3. Kod po `yield` odpowiada metodzie `__exit__`.

`contextlib.contextmanager` automatycznie obsługuje wyjątki:

```
Copy@contextmanager
def error_handler():
try:
yield
except Exception as e:
print(f"Złapano wyjątek: {type(e).__name__}: {e}")
with error_handler():
raise ValueError("Przykładowy błąd")
```

Użycie `@contextmanager` znacznie upraszcza tworzenie menedżerów kontekstu, szczególnie gdy nie potrzebujemy pełnej kontroli nad procesem wejścia i wyjścia z kontekstu. Jest to szczególnie przydatne dla prostych przypadków użycia, gdzie głównym celem jest zapewnienie poprawnego zarządzania zasobami lub obsługa wyjątków.

1. Pliki: Wbudowany menedżer kontekstu dla plików zapewnia automatyczne zamknięcie pliku po zakończeniu operacji:

```
Copywith open('plik.txt', 'w') as f:
f.write('Przykładowy tekst')
# Plik zostaje automatycznie zamknięty po wyjściu z bloku with
```

2. Połączenia sieciowe: Dla połączeń sieciowych, np. z użyciem biblioteki `socket`:

```
Copyimport socket
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
s.connect(('example.com', 80))
s.sendall(b'GET / HTTP/1.1\r\nHost: example.com\r\n\r\n')
data = s.recv(1024)
# Połączenie zostaje automatycznie zamknięte
```

3. Blokady: Menedżery kontekstu są przydatne do zarządzania blokadami, np. z modułu `threading`:

```
Copyimport threading
lock = threading.Lock()
with lock:
# Kod chroniony blokadą
pass
# Blokada zostaje automatycznie zwolniona
```

Zagnieżdżone menedżery kontekstu

Python pozwala na zagnieżdżanie menedżerów kontekstu, co jest przydatne, gdy potrzebujemy zarządzać wieloma zasobami jednocześnie:

```
Copywith open('plik1.txt', 'r') as f1, open('plik2.txt', 'w') as f2:
content = f1.read()
f2.write(content.upper())
```

Można również używać wielu linii dla lepszej czytelności:

```
Copywith (
open('plik1.txt', 'r') as f1,
open('plik2.txt', 'w') as f2,
open('plik3.txt', 'w') as f3
):
content = f1.read()
f2.write(content.upper())
f3.write(content.lower())
```

Aliasy w with (as)

Słowo kluczowe `as` w instrukcji `with` pozwala na przypisanie wyniku metody `__enter__` do zmiennej, która może być używana wewnątrz bloku kontekstowego:

```
Copyclass ContextExample:
def __enter__(self):
return "Wartość zwrócona przez __enter__"
```

```
    def __exit__(self, exc_type, exc_value, traceback):
    pass
    with ContextExample() as ce:
    print(ce)  # Wypisze: Wartość zwrócona przez __enter__
```

Można również używać wielu aliasów w jednej instrukcji `with`:

```
    Copywith open('input.txt', 'r') as input_file, open('output.txt', 'w') as output_file:
    for line in input_file:
    output_file.write(line.upper())
```

W przypadku menedżerów kontekstu, które nie zwracają użytecznej wartości w `__enter__`, można pominąć alias:

```
    Copywith open('log.txt', 'a'):
    # Plik zostanie otwarty i zamknięty, ale nie potrzebujemy do niego odwoływać się
    pass
```

Aliasy są szczególnie przydatne, gdy menedżer kontekstu zwraca obiekt, który ma być używany wewnątrz bloku `with`. Pozwalają na czytelne i zwięzłe zarządzanie zasobami, jednocześnie zapewniając dostęp do tych zasobów w kontrolowany sposób.

Asynchroniczne menedżery kontekstu są używane w programowaniu asynchronicznym i są definiowane za pomocą metod `__aenter__` i `__aexit__`. Używa się ich z instrukcją `async with`:

```
    Copyimport asyncio
    class AsyncContextManager:
    async def __aenter__(self):
    await asyncio.sleep(1)  # Symulacja asynchronicznej operacji
    print("Wejście do kontekstu asynchronicznego")
    return self
    async def __aexit__(self, exc_type, exc_value, traceback):
    await asyncio.sleep(1)  # Symulacja asynchronicznej operacji
    print("Wyjście z kontekstu asynchronicznego")
    async def main():
    async with AsyncContextManager() as manager:
    print("Wewnątrz kontekstu asynchronicznego")
    asyncio.run(main())
```

Asynchroniczne menedżery kontekstu są szczególnie przydatne przy pracy z asynchronicznymi zasobami, takimi jak połączenia do baz danych czy sesje HTTP:

```
Copyimport aiohttp
import asyncio
async def fetch_data(url):
async with aiohttp.ClientSession() as session:
async with session.get(url) as response:
return await response.text()
asyncio.run(fetch_data('http://example.com'))
```

Konteksty wykonania w wielowątkowości

Menedżery kontekstu są przydatne w programowaniu wielowątkowym do zarządzania współdzielonymi zasobami i synchronizacją:

1. Blokady (Locks):

```
Copyimport threading
lock = threading.Lock()
def worker():
with lock:
# Kod chroniony blokadą
pass
threads = [threading.Thread(target=worker) for _ in range(5)]
for thread in threads:
thread.start()
```

2. Semafory:

```
Copyimport threading
semaphore = threading.Semaphore(2)  # Maksymalnie 2 wątki jednocześnie
def worker():
with semaphore:
# Kod ograniczony do wykonywania przez maksymalnie 2 wątki
pass
```

3. Lokalne dane wątku:

```
Copyimport threading
thread_local = threading.local()
def set_context():
thread_local.value = threading.current_thread().name
def worker():
with threading.local() as local:
set_context()
print(f"Wątek {local.value} wykonuje pracę")
threads = [threading.Thread(target=worker) for _ in range(3)]
```

```
for thread in threads:
thread.start()
```

Zastosowania menedżerów kontekstu w testowaniu

Menedżery kontekstu są niezwykle przydatne w testowaniu, szczególnie do:

1. Mockowania:

```
Copyfrom unittest.mock import patch
def test_function():
with patch('module.function') as mock_function:
mock_function.return_value = 'mocked value'
# Wykonaj test z podmienioną funkcją
```

2. Tymczasowych zmian środowiska:

```
Copyimport os
class EnvironmentVariable:
def __init__(self, key, value):
self.key = key
self.value = value
self.original = None
def __enter__(self):
self.original = os.environ.get(self.key)
os.environ[self.key] = self.value
def __exit__(self, exc_type, exc_value, traceback):
if self.original is None:
del os.environ[self.key]
else:
os.environ[self.key] = self.original
def test_environment_dependent():
with EnvironmentVariable('TEST_VAR', 'test_value'):
# Wykonaj test z ustawioną zmienną środowiskową
```

3. Zarządzanie zasobami testowymi:

```
Copyimport tempfile
import os
class TempDirectory:
def __enter__(self):
self.temp_dir = tempfile.mkdtemp()
return self.temp_dir
def __exit__(self, exc_type, exc_value, traceback):
os.rmdir(self.temp_dir)
def test_file_operations():
```

```
    with TempDirectory() as temp_dir:
    # Wykonaj testy operacji na plikach w tymczasowym katalogu
```

Menedżery kontekstu w testowaniu pomagają w izolowaniu testów, zarządzaniu zasobami testowymi i symulowaniu różnych warunków środowiskowych, co prowadzi do bardziej niezawodnych i powtarzalnych testów.

## 1.4. Zaawansowane aspekty dziedziczenia

Mieszaniny (mixins) to klasy zaprojektowane do rozszerzania funkcjonalności innych klas poprzez wielodziedziczenie. Nie są przeznaczone do samodzielnego użytku, ale do łączenia z innymi klasami.

Cechy mieszanin:

1. Implementują konkretne metody lub zestaw metod.
2. Nie mają własnego stanu (zazwyczaj nie mają **init**).
3. Służą do dzielenia funkcjonalności między niezwiązanymi klasami.

Przykład:

```
    Copyclass JSONSerializableMixin:
    def to_json(self):
    import json
    return json.dumps(self.__dict__)
    class User(JSONSerializableMixin):
    def __init__(self, name, email):
    self.name = name
    self.email = email
    user = User("Jan", "jan@example.com")
    print(user.to_json())  # {"name": "Jan", "email":
"jan@example.com"}
```

Kompozycja vs dziedziczenie

Kompozycja i dziedziczenie to dwa różne podejścia do projektowania relacji między klasami.

Dziedziczenie:

* Tworzy relację "jest" (is-a).
* Pozwala na ponowne użycie kodu.
* Może prowadzić do sztywnych hierarchii.

Kompozycja:
* Tworzy relację "ma" (has-a).
* Oferuje większą elastyczność.
* Ułatwia zmianę zachowania w czasie wykonania.
Przykład kompozycji:

```
Copyclass Engine:
def start(self):
return "Engine started"
class Car:
def __init__(self):
self.engine = Engine()
def start(self):
return self.engine.start()
car = Car()
print(car.start())  # Engine started
```

Zaleca się preferowanie kompozycji nad dziedziczeniem, gdy to możliwe ("Composition over inheritance").
Dziedziczenie wielokrotne a wzorce projektowe
Dziedziczenie wielokrotne w Pythonie umożliwia implementację kilku ważnych wzorców projektowych:
1. Adapter:

```
Copyclass OldSystem:
def old_method(self):
return "Old method"
class NewInterface:
def new_method(self):
pass
class Adapter(OldSystem, NewInterface):
def new_method(self):
return self.old_method()
```

2. Dekorator:

```
Copyclass Component:
def operation(self):
return "Basic operation"
class Decorator(Component):
def __init__(self, component):
self.component = component
def operation(self):
return f"Decorated {self.component.operation()}"
```

```
# Użycie
basic = Component()
decorated = Decorator(basic)
print(decorated.operation())  # Decorated Basic operation
```

3. Strategia:

```
Copyclass DefaultStrategy:
def execute(self):
return "Default strategy"
class AlternativeStrategy:
def execute(self):
return "Alternative strategy"
class Context(DefaultStrategy, AlternativeStrategy):
def __init__(self, strategy=None):
self.strategy = strategy or self.execute
def execute_strategy(self):
return self.strategy()
# Użycie
context = Context(AlternativeStrategy().execute)
print(context.execute_strategy())  # Alternative strategy
```

4. Template Method:

```
Copyclass AbstractClass:
def template_method(self):
self.step1()
self.step2()
def step1(self):
pass
def step2(self):
pass
class ConcreteClass(AbstractClass):
def step1(self):
print("Implementacja kroku 1")
def step2(self):
print("Implementacja kroku 2")
```

Dziedziczenie wielokrotne w Pythonie pozwala na elastyczne implementacje tych wzorców, ale wymaga ostrożności, aby uniknąć konfliktów nazw i zapewnić czytelność kodu.

Metody klasowe i statyczne są dziedziczone w podobny sposób jak zwykłe metody instancji, ale z pewnymi różnicami:

Metody klasowe:

* Dziedziczone przez klasy pochodne.

* Pierwszy argument (cls) odnosi się do klasy, która wywołuje metodę, a nie do klasy, w której metoda jest zdefiniowana.

```
Copyclass Bazowa:
@classmethod
def metoda_klasowa(cls):
print(f"Metoda klasowa wywołana dla {cls.__name__}")
class Pochodna(Bazowa):
pass
Pochodna.metoda_klasowa()  # Wypisze: Metoda klasowa
wywołana dla Pochodna
```

Metody statyczne:

* Również dziedziczone.

* Zachowują się tak samo w klasie bazowej i pochodnej, ponieważ nie mają dostępu do stanu klasy ani instancji.

```
Copyclass Bazowa:
@staticmethod
def metoda_statyczna():
print("Metoda statyczna")
class Pochodna(Bazowa):
pass
Pochodna.metoda_statyczna()  # Wypisze: Metoda statyczna
```

Abstrakcyjne klasy bazowe (abc module)

Moduł abc (Abstract Base Classes) w Pythonie umożliwia tworzenie abstrakcyjnych klas bazowych, które definiują interfejs dla klas pochodnych.

Główne cechy:

* Nie można tworzyć instancji abstrakcyjnej klasy bazowej.

* Klasy pochodne muszą implementować wszystkie abstrakcyjne metody.

Przykład:

```
Copyfrom abc import ABC, abstractmethod
class Ksztalt(ABC):
@abstractmethod
def oblicz_pole(self):
pass
class Kwadrat(Ksztalt):
def __init__(self, bok):
```

```
    self.bok = bok
    def oblicz_pole(self):
    return self.bok ** 2
    # Kwadrat().oblicz_pole()  # OK
    # Ksztalt()  # Błąd: nie można utworzyć instancji klasy
abstrakcyjnej
```

Interfejsy w Pythonie

Python nie ma wbudowanego konceptu interfejsów jak w niektórych innych językach, ale można je symulować za pomocą abstrakcyjnych klas bazowych:

```
    Copyfrom abc import ABC, abstractmethod
    class InterfejsBazy(ABC):
    @abstractmethod
    def polacz(self):
    pass
    @abstractmethod
    def rozlacz(self):
    pass
    class BazaMySQL(InterfejsBazy):
    def polacz(self):
    print("Połączenie z MySQL")
    def rozlacz(self):
    print("Rozłączenie z MySQL")
    class BazaPostgreSQL(InterfejsBazy):
    def polacz(self):
    print("Połączenie z PostgreSQL")
    def rozlacz(self):
    print("Rozłączenie z PostgreSQL")
```

Protokoły (Python 3.8+): Python 3.8 wprowadził koncepcję protokołów, które są bliższe tradycyjnym interfejsom:

```
    Copyfrom typing import Protocol
    class Drawable(Protocol):
    def draw(self) -> None:
    ...
    class Circle:
    def draw(self) -> None:
    print("Rysowanie koła")
    def draw_shape(shape: Drawable):
    shape.draw()
```

```
    circle = Circle()
    draw_shape(circle)  # OK, Circle implementuje protokół
Drawable
```

Protokoły zapewniają statyczne sprawdzanie typów bez konieczności jawnego dziedziczenia.

Abstrakcyjne klasy bazowe i protokoły w Pythonie umożliwiają definiowanie kontraktów, które muszą być spełnione przez klasy implementujące, co prowadzi do bardziej przewidywalnego i łatwiejszego w utrzymaniu kodu.

W Pythonie konwencje nazewnicze definiują atrybuty prywatne i chronione:

1. Atrybuty prywatne (z przedrostkiem __):

* Nie są bezpośrednio dostępne w klasach pochodnych.

* Python stosuje name mangling, zmieniając nazwę na _NazwaKlasy__atrybut.

```
    Copyclass Bazowa:
    def __init__(self):
    self.__prywatny = "Prywatny"
    self._chroniony = "Chroniony"
    class Pochodna(Bazowa):
    def __init__(self):
    super().__init__()
    print(self._chroniony)  # Dostęp do chronionego - OK
    # print(self.__prywatny)  # Błąd - AttributeError
    print(self._Bazowa__prywatny)  # Dostęp do prywatnego przez
name mangling - OK
```

2. Atrybuty chronione (z przedrostkiem _):

* Są dziedziczone i dostępne w klasach pochodnych.

* Konwencja sugeruje, że nie powinny być używane poza klasą i jej potomkami.

Dynamiczne dziedziczenie (zmiana **bases**)

Python pozwala na dynamiczną zmianę klas bazowych w czasie wykonania poprzez modyfikację atrybutu **bases**:

```
    Copyclass A:
    def metoda(self):
    return "A"
    class B:
    def metoda(self):
```

```
return "B"
class C(A):
pass
print(C().metoda())  # Wypisze: A
# Zmiana klasy bazowej
C.__bases__ = (B,)
print(C().metoda())  # Wypisze: B
```

Należy używać tej techniki ostrożnie, ponieważ może prowadzić do trudnych do wykrycia błędów i nieoczekiwanych zachowań.

Rozwiązywanie konfliktów nazw w wielodziedziczeniu

Gdy klasa dziedziczy po wielu klasach bazowych, mogą pojawić się konflikty nazw. Python rozwiązuje je zgodnie z Method Resolution Order (MRO):

1. Kolejność dziedziczenia ma znaczenie:

```
Copyclass A:
def metoda(self):
return "A"
class B:
def metoda(self):
return "B"
class C(A, B):
pass
class D(B, A):
pass
print(C().metoda())  # Wypisze: A
print(D().metoda())  # Wypisze: B
```

2. Użycie super() do jawnego wywołania metod z klas bazowych:

```
Copyclass A:
def metoda(self):
return "A"
class B:
def metoda(self):
return "B"
class C(A, B):
def metoda(self):
return super().metoda() + " i C"
print(C().metoda())  # Wypisze: A i C
```

3. Jawne wywoływanie metod z konkretnych klas bazowych:

```
Copyclass A:
```

```
def metoda(self):
return "A"
class B:
def metoda(self):
return "B"
class C(A, B):
def metoda(self):
return A.metoda(self) + " i " + B.metoda(self)
print(C().metoda())  # Wypisze: A i B
```

4. Używanie mixin-ów do rozdzielenia funkcjonalności:

```
Copyclass LoggingMixin:
def log(self, message):
print(f"Log: {message}")
class A:
def metoda(self):
return "A"
class B(LoggingMixin, A):
def metoda(self):
result = super().metoda()
self.log(result)
return result
B().metoda()  # Wypisze: Log: A
```

Przy rozwiązywaniu konfliktów nazw kluczowe jest zrozumienie MRO i świadome projektowanie hierarchii klas. W złożonych przypadkach warto rozważyć użycie kompozycji zamiast wielodziedziczenia.

## 1.5. Protokoły i abstrakcyjne klasy bazowe

Moduł collections.abc (Abstract Base Classes) zawiera zestaw abstrakcyjnych klas bazowych, które definiują interfejsy dla różnych typów kolekcji i iteratorów w Pythonie. Główne klasy w tym module to:

1. Iterable: dla obiektów, które można iterować.
2. Container: dla obiektów, które obsługują operator in.
3. Sized: dla obiektów z metodą **len**.
4. Sequence: dla uporządkowanych, indeksowalnych kolekcji.

5. MutableSequence: dla modyfikowalnych sekwencji.
6. Mapping: dla obiektów typu słownikowego.
7. MutableMapping: dla modyfikowalnych mapowań.
8. Set: dla zbiorów.
9. MutableSet: dla modyfikowalnych zbiorów.

Przykład użycia:

```
Copyfrom collections.abc import Sequence
class CustomList(Sequence):
def __init__(self, data):
self._data = data
def __len__(self):
return len(self._data)
def __getitem__(self, index):
return self._data[index]
custom_list = CustomList([1, 2, 3])
print(isinstance(custom_list, Sequence))  # True
```

Tworzenie własnych abstrakcyjnych klas bazowych

Aby stworzyć własną abstrakcyjną klasę bazową, należy:

1. Zaimportować moduł abc.
2. Dziedziczyć po abc.ABC.
3. Użyć dekoratora @abstractmethod dla metod abstrakcyjnych.

Przykład:

```
Copyfrom abc import ABC, abstractmethod
class AbstractVehicle(ABC):
@abstractmethod
def start_engine(self):
pass
@abstractmethod
def stop_engine(self):
pass
def honk(self):
print("Beep beep!")
class Car(AbstractVehicle):
def start_engine(self):
print("Car engine started")
def stop_engine(self):
print("Car engine stopped")
# car = Car()  # OK
```

```
# vehicle = AbstractVehicle()  # Błąd: nie można utworzyć instancji klasy abstrakcyjnej
```

Dekorator @abstractmethod

Dekorator @abstractmethod służy do oznaczania metod jako abstrakcyjnych w klasach abstrakcyjnych. Główne cechy:

1. Metoda oznaczona jako @abstractmethod musi być zaimplementowana w klasach pochodnych.
2. Klasa zawierająca co najmniej jedną metodę abstrakcyjną staje się klasą abstrakcyjną.
3. Nie można tworzyć instancji klas abstrakcyjnych.
4. Klasy pochodne muszą implementować wszystkie metody abstrakcyjne, aby można było tworzyć ich instancje.

Przykład użycia:

```
Copyfrom abc import ABC, abstractmethod
class Shape(ABC):
@abstractmethod
def area(self):
pass
@abstractmethod
def perimeter(self):
pass
class Rectangle(Shape):
def __init__(self, width, height):
self.width = width
self.height = height
def area(self):
return self.width * self.height
def perimeter(self):
return 2 * (self.width + self.height)
# shape = Shape()  # Błąd: nie można utworzyć instancji klasy abstrakcyjnej
rect = Rectangle(5, 3)
print(rect.area())  # 15
print(rect.perimeter())  # 16
```

Abstrakcyjne klasy bazowe i dekorator @abstractmethod są narzędziami do definiowania interfejsów i wymuszania ich implementacji w klasach pochodnych. Pomagają w tworzeniu spójnego i przewidywalnego API, ułatwiając tym samym projektowanie i utrzymanie kodu.

Protokoły strukturalne, wprowadzone w PEP 544, to sposób definiowania interfejsów w Pythonie, który opiera się na strukturze obiektu, a nie na jawnym dziedziczeniu. Główne cechy:

1. Definicja poprzez klasę Protocol z modułu typing.

2. Obiekty są zgodne z protokołem, jeśli mają wymagane metody i atrybuty.

3. Brak konieczności jawnej deklaracji implementacji.

Przykład:

```
Copyfrom typing import Protocol
class Printable(Protocol):
def print(self) -> None:
...
class Document:
def print(self) -> None:
print("Drukuję dokument")
def print_object(obj: Printable) -> None:
obj.print()
doc = Document()
print_object(doc)  # Działa, mimo że Document nie dziedziczy po Printable
```

Moduł typing i protokoły w typowaniu statycznym

Moduł typing w Pythonie dostarcza narzędzi do statycznego typowania, w tym wsparcie dla protokołów:

1. Protocol: klasa bazowa dla definiowania protokołów.

2. runtime_checkable: dekorator umożliwiający sprawdzanie zgodności z protokołem w czasie wykonania.

Przykład:

```
Copyfrom typing import Protocol, runtime_checkable
@runtime_checkable
class Sized(Protocol):
def __len__(self) -> int:
...
class CustomContainer:
def __len__(self) -> int:
return 42
container = CustomContainer()
print(isinstance(container, Sized))  # True
```

Protokoły w typowaniu statycznym pozwalają na:

1. Definiowanie interfejsów bez konieczności dziedziczenia.

2. Sprawdzanie typów w czasie kompilacji przez narzędzia takie jak mypy.

3. Opcjonalne sprawdzanie w czasie wykonania przy użyciu @runtime_checkable.

Różnice między protokołami a abstrakcyjnymi klasami bazowymi

1. Definicja:

* Protokoły: definiowane jako klasy dziedziczące po typing.Protocol.

* ABC: definiowane jako klasy dziedziczące po abc.ABC.

2. Implementacja:

* Protokoły: implicitna, bazująca na strukturze obiektu.

* ABC: explicytna, wymaga jawnego dziedziczenia.

3. Sprawdzanie zgodności:

* Protokoły: głównie w czasie kompilacji, opcjonalnie w czasie wykonania.

* ABC: w czasie wykonania.

4. Elastyczność:

* Protokoły: bardziej elastyczne, umożliwiają częściową implementację.

* ABC: mniej elastyczne, wymagają pełnej implementacji metod abstrakcyjnych.

5. Użycie:

* Protokoły: idealne do definiowania interfejsów dla typowania statycznego.

* ABC: dobre do wymuszania implementacji konkretnych metod i tworzenia szkieletów klas.

Przykład różnicy:

```
Copyfrom typing import Protocol
from abc import ABC, abstractmethod
class PrintableProtocol(Protocol):
def print(self) -> None:
...
class PrintableABC(ABC):
@abstractmethod
def print(self) -> None:
pass
# Dla protokołu
class DocumentP:
def print(self) -> None:
```

```
    print("Drukuję dokument")
    # Dla ABC
    class DocumentA(PrintableABC):
    def print(self) -> None:
    print("Drukuję dokument")
    # Użycie
    doc_p = DocumentP()  # OK
    doc_a = DocumentA()  # OK
    print(isinstance(doc_p, PrintableProtocol))  # True, jeśli użyto
@runtime_checkable
    print(isinstance(doc_a, PrintableABC))  # True
```

Protokoły oferują większą elastyczność i są bardziej zgodne z filozofią duck typing Pythona, podczas gdy abstrakcyjne klasy bazowe zapewniają silniejsze gwarancje dotyczące implementacji i są bardziej odpowiednie do definiowania szkieletów klas.

Protokoły można implementować dla istniejących klas bez modyfikacji ich kodu źródłowego. Można to osiągnąć poprzez:

1. Rejestrację klasy jako implementującej protokół:

```
    Copyfrom typing import Protocol, runtime_checkable
    @runtime_checkable
    class Printable(Protocol):
    def print(self) -> None:
    ...
    class ExistingClass:
    def print(self) -> None:
    print("Drukuję")
    # Rejestracja
    Printable.register(ExistingClass)
    obj = ExistingClass()
    print(isinstance(obj, Printable))  # True
```

2. Użycie adaptera:

```
    Copyclass LegacyPrinter:
    def do_print(self):
    print("Drukuję starą metodą")
    class PrinterAdapter(Printable):
    def __init__(self, legacy_printer):
    self.legacy_printer = legacy_printer
    def print(self) -> None:
```

```
    self.legacy_printer.do_print()
    legacy = LegacyPrinter()
    adapter = PrinterAdapter(legacy)
```

Zastosowanie protokołów i abstrakcyjnych klas bazowych w projektowaniu API

1. Definiowanie kontraktów:

* Protokoły: dla luźno powiązanych interfejsów.
* ABC: dla ścisłych kontraktów z wymuszeniem implementacji.

```
    Copyfrom typing import Protocol
    from abc import ABC, abstractmethod
    class DataSource(Protocol):
    def get_data(self) -> str:
    ...
    class DataProcessor(ABC):
    @abstractmethod
    def process(self, data: str) -> str:
    pass
    def process_data(source: DataSource, processor: DataProcessor)
-> str:
    data = source.get_data()
    return processor.process(data)
```

2. Tworzenie rozszerzalnych systemów:

```
    Copyclass Plugin(Protocol):
    def execute(self) -> None:
    ...
    class PluginManager:
    def __init__(self):
    self.plugins: list[Plugin] = []
    def add_plugin(self, plugin: Plugin) -> None:
    self.plugins.append(plugin)
    def execute_all(self) -> None:
    for plugin in self.plugins:
    plugin.execute()
```

3. Definiowanie interfejsów dla bibliotek:

```
    Copyclass Serializable(Protocol):
    def to_json(self) -> str:
    ...
    def from_json(self, json_str: str) -> None:
    ...
```

```
def save_to_file(obj: Serializable, filename: str) -> None:
with open(filename, 'w') as f:
f.write(obj.to_json())
```

Wpływ protokołów na wydajność i elastyczność kodu

Wydajność:

1. Protokoły nie wprowadzają dodatkowego narzutu w czasie wykonania, w przeciwieństwie do sprawdzania dziedziczenia ABC.

2. Statyczne sprawdzanie typów z użyciem protokołów może pomóc w wykryciu błędów na etapie kompilacji, co poprawia ogólną wydajność i niezawodność kodu.

```
Copyfrom typing import Protocol
class Drawable(Protocol):
def draw(self) -> None:
...
def render(objects: list[Drawable]) -> None:
for obj in objects:
obj.draw()  # Brak sprawdzania w runtime
```

Elastyczność:

1. Protokoły umożliwiają tworzenie bardziej modularnego kodu, ułatwiając wymianę komponentów.

2. Pozwalają na częściową implementację interfejsu, co jest przydatne w przypadku mock objects w testach.

```
Copyclass MockDrawable:
def draw(self) -> None:
pass  # Minimalna implementacja dla testów
def test_render():
mock = MockDrawable()
render([mock])  # Działa bez pełnej implementacji
```

3. Ułatwiają adaptację istniejącego kodu do nowych interfejsów bez konieczności modyfikacji oryginalnych klas.

```
Copyclass LegacyComponent:
def old_method(self):
print("Stara metoda")
class ComponentAdapter(Protocol):
def new_method(self):
...
def adapt(component: LegacyComponent) ->
ComponentAdapter:
class Adapter:
```

```
    def new_method(self):
    component.old_method()
    return Adapter()
    adapted = adapt(LegacyComponent())
    adapted.new_method()  # Wywołuje "Stara metoda"
```

Protokoły i abstrakcyjne klasy bazowe oferują różne podejścia do projektowania API, z protokołami faworyzującymi elastyczność i zgodność z duck typing, a ABC zapewniającymi silniejsze gwarancje implementacji. Wybór między nimi zależy od konkretnych wymagań projektu i preferowanego stylu programowania.

# 2. Metaprogramowanie i magia Pythona