Jakub Walczak

ELEMENTY INŻYNIERII OPROGRAMOWANIA W PYTHONIE

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/iopyth
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-9446-9

Copyright © Helion S.A. 2023

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Przedmowa **7**

Wstęp **9**

1. Python — krótka charakterystyka **12**

2. Organizacja środowiska pracy **15**
2.1. Instalacja Pythona 15
2.2. Przygotowanie środowiska wirtualnego 16
2.3. Środowiska pracy w ekosystemie Pythona 20
2.3.1. Powłoka Pythona 20
2.3.2. Edytor tekstu 21
2.3.3. Interaktywny Python 21
2.3.4. Zintegrowane środowisko deweloperskie 23

3. Organizacja projektu **25**
3.1. Moduły, pakiety i przestrzenie nazw 25
3.1.1. Moduły 25
3.1.2. Pakiety 28
3.1.3. Przestrzenie nazw 30
3.2. Importowanie modułów 32
3.2.1. Importowanie pośrednie 34
3.2.2. Importowanie bezpośrednie 35
3.2.3. Importowanie z użyciem symbolu wieloznacznego 36
3.2.4. Importowanie z wykorzystaniem aliasu 37
3.3. Budowanie i publikowanie pakietu 37
3.3.1. Plik `pyproject.toml` 37
3.3.2. Określanie plików składowych biblioteki 40
3.3.3. Dynamiczne określanie metadanych projektu 41
3.3.4. Załączanie plików zasobów 42
3.3.5. Publikowanie projektu 42
3.4. Jawne typowanie zmiennych i funkcji 44

4. Wstęp do programowania zorientowanego obiektowo **47**
4.1. Pojęcie klasy i obiektu 47
4.2. Paradygmat programowania obiektowego 48
4.3. Definiowanie klas i tworzenie obiektów 49

4.4. Destrukcja i finalizowanie obiektu 52
4.5. Pola i metody niepubliczne 54
4.6. Atrybuty i metody klasowe 56
4.7. Metody statyczne 59

5. Dekoratory **61**
5.1. Funkcje zagnieżdżone i zmienne nielokalne 61
5.2. Domknięcie funkcji 62
5.3. Dekoratory nieparametryczne 64
5.4. Dekoratory parametryczne 66
5.5. Atrybuty specjalne funkcji dekorowanych 67

6. Deskryptory **70**
6.1. Protokół deskryptora 70
6.2. Przekazywanie nazwy atrybutu do deskryptora 74
6.3. Rodzaje deskryptorów 74
6.4. Deskryptor własności 76

7. Dziedziczenie **79**
7.1. Dziedziczenie wielorakie 82
7.2. Kolejność dostępu 84
7.3. Przeszukiwanie grafu dziedziczenia 85
7.4. Metaklasy 86
7.5. Klasy szczególne 88
7.5.1. Klasa abstrakcyjna 88
7.5.2. Klasa wyliczająca 90

8. Mechanizm obsługi wyjątków **96**
8.1. Podział wyjątków 96
8.2. Wzbudzanie wyjątku 98
8.3. Obsługa wyjątków 100

9. Metody specjalne klas **103**
9.1. Reprezentacja tekstowa obiektu 103
9.2. Metody porównywania 107
9.3. Wartość skrótu obiektu 109
9.4. Metody specjalne kolekcji 111
9.4.1. Abstrakcyjne metody dla kolekcji sekwencyjnych 112
9.4.2. Abstrakcyjne metody dla kolekcji o charakterze zbiorów 114
9.4.3. Abstrakcyjne metody dla kolekcji mapujących 115
9.4.4. Emulowanie zachowania iteratora 115
9.5. Metody operatorów arytmetycznych 116
9.6. Sprawdzanie wartości logicznej obiektu 118
9.7. Emulowanie zachowania funkcyjnego obiektu 122
9.8. Metody menedżera kontekstu 126

Kup ksi k
Pole ksi k

10. Serializacja i deserializacja **131**
10.1. Prosty format tekstowy 133
10.2. Formaty słownikowe 135
10.3. Piklowanie obiektów 141
10.4. Inne mechanizmy serializacji 142

11. Testy jednostkowe z użyciem biblioteki *pytest* **144**
11.1. Funkcje testujące. Asercje 145
11.2. Parametryzacja testów 147
11.3. Obiekty trwałe w testach 148
11.4. Anotacja testów 151
11.4.1. Oznaczanie testów do pominięcia 152
11.4.2. Anotacja warunkowego pominięcia testów 152
11.4.3. Oznaczanie testów celowo niezaliczonych 152
11.4.4. Własne markery testów 153

12. Wytyczne dotyczące stylu **155**
12.1. Zalecenia ogólne 155
12.2. Zalecenia dotyczące nazewnictwa 156
12.3. Zalecenia dotyczące struktury kodu 156
12.4. Zalecenia dotyczące importowania 156
12.5. Zalecenia dotyczące logiki 157
12.6. Zalecenia dotyczące testów jednostkowych 158
12.7. Zalecenia dotyczące formatowania kodu 158

Bibliografia **160**

Skorowidz **166**

Kup ksi k Pole ksi k

Rozdział 5.

Dekoratory

Przyszła pora, żeby omówić **dekoratory**. Są to charakterystyczne elementy wspierające tzw. metaprogramowanie w Pythonie. Ale żeby dobrze zrozumieć, jak działają, musimy zacząć od wprowadzenia funkcji zagnieżdżonych oraz mechanizmu domknięcia (ang. *closure*).

5.1. Funkcje zagnieżdżone i zmienne nielokalne

Funkcja zagnieżdżona to nic innego jak funkcja umieszczona wewnątrz innej funkcji (listing 5.1).

Listing 5.1. Funkcja zagnieżdżona w innej funkcji

```
1  # definicja funkcji zewnętrznej
2  def make_bold(text):
3      # definicja funkcji zagnieżdżonej
4      def print_text():
5          print(f"<b>{text}</b>")
6      print_text()
```

Ale to jest chyba dość jasne. Nazwa mówi sama za siebie — funkcja zagnieżdżona. Dla nas ważniejsza jest jednak kwestia widoczności poszczególnych zmiennych.

Jeżeli wykonasz powyższą funkcję, zobaczysz, że spowoduje ona wyświetlenie tekstu między znacznikami `<b>...</b>` (listing 5.2).

Listing 5.2. Wykonanie funkcji zagnieżdżonej

```
>>> make_bold("some text")
<b>some text</b>
```

Kup ksi k Pole ksi k

Zatem zmienna `text`, która stanowi argument funkcji zewnętrznej, jest również widoczna w bloku funkcji zagnieżdżonej. Zmienna `text` jest więc dla funkcji `print_text()` zmienną **nielokalną**.

A co się stanie, jeżeli wewnątrz funkcji zagnieżdżonej zdefiniujemy zmienną o takiej samej nazwie? Zmienna nielokalna zostanie **przesłonięta** przez zmienną lokalną, czyli zmienna `text` zostanie zdefiniowana w lokalnej przestrzeni nazw [48]. Poniżej znajdziesz przykład takiego przesłonięcia (listing 5.3).

Listing 5.3. Przesłonięcie zmiennej nielokalnej przez zmienną lokalną

```
1 def make_bold(text):
2     def print_text():
3         text = "inner function call"
4         print(text)
5     print_text()
6 make_bold("outer function call") # OUT: inner function call
```

Zmienne nielokalne w funkcjach zagnieżdżonych mają charakter **tylko do odczytu**! Jeżeli chcesz je modyfikować, konieczna jest deklaracja zmiennej w bloku lokalnym z użyciem słowa `nonlocal` [49]. Patrz: podrozdział 3.1.3.

5.2. Domknięcie funkcji

Domknięcie (ang. *closure*) jest mechanizmem wykorzystującym funkcje zagnieżdżone i zmienne nielokalne. Zmienne w Pythonie charakteryzują się tym, że gdy nie są już potrzebne, np. po wyjściu z bloku funkcji, tracimy do nich dostęp i zostają one usunięte z pamięci (listing 5.4).

Listing 5.4. Przykład zmiennych nielokalnych usuwanych z pamięci po wyjściu z bloku funkcji zewnętrznej

```
1 def make_bold(text):
2     tag_start = "<b>"
3     tag_end = "</b>"
4     def print_text():
5         print(f"{tag_start}{text}{tag_end}")
6     print_text()
```

W powyższym przykładzie tracimy zmienne zdefiniowane w linijkach 2. oraz 3. zaraz po wyjściu z bloku funkcji zewnętrznej. Inaczej jest z domknięciami! Pozwalają one „zapamiętać” zmienne lokalne i nielokalne nawet po wyjściu z danego bloku kodu. A jak zdefiniować domknięcie? Wystarczy, że funkcja zewnętrzna będzie zwracała funkcję zagnieżdżoną (referencję do niej) bez jej wykonywania.

Kup ksi k Pole ksi k

Spójrz na następujący wycinek kodu (listing 5.5).

Listing 5.5. Zmienne nielokalne przechowywane w ramach funkcji domknięcia

```
def wrap_text(tag):
    tag_start = f"<{tag}>"
    tag_end = f"</{tag}>"
    def print_text(text):
        print(f"{tag_start}{text}{tag_end}")
    return print_text
```

Jak widzisz, w linijce 6. zwracana jest funkcja, a nie rezultat jej wykonywania (nie ma nawiasów). Dzięki temu wyrażenie `wrap_text()` zwróci nam funkcję `print_text()`, która uwzględnia zmienne nielokalne `tag_start` oraz `tag_end`.

Teraz możemy w prosty sposób stworzyć kolekcję funkcji dla różnych tagów (listing 5.6).

Listing 5.6. Parametryzowanie funkcji poprzez wykorzystanie domknięcia

```
make_bold = wrap_text("b")
make_italic = wrap_text("i")
make_underscore = wrap_text("u")
```

W ten sposób otrzymaliśmy trzy funkcje. Każda z nich przyjmuje jeden parametr — `text`. Wynika to z tego, że to właśnie funkcja `print_text()` przyjmująca ten argument jest obiektem zwróconym przez funkcję zewnętrzną — `wrap_text()`. Sprawdźmy teraz skuteczność działania naszego domknięcia (listing 5.7).

Listing 5.7. Wykonanie sparametryzowanej funkcji domknięcia

```
>>> make_bold("some text")
<b>some text</b>
```

⚑ Pamiętaj, że domknięcia są leniwie ewaluowane (ang. *lazy evaluated*), tzn. nie są wykonywane, póki nie zostaną jawnie wywołane w kodzie. W związku z tym ewentualne błędy mogą nie być widoczne w momencie definiowania.

Ale nasze domknięcie możemy napisać lepiej! To znaczy: w sposób, który pozwoli nam łączyć kilka operacji naraz. Wystarczy, że linijkę 5. domknięcia (listing 5.5) zastąpimy poleceniem zwracającym ciąg tekstowy (listing 5.8).

Kup ksi k Pole ksi k

Listing 5.8. Alternatywna implementacja domknięcia zwracająca ciąg znaków

```
def wrap_text(tag):
    tag_start = f"<{tag}>"
    tag_end = f"</{tag}>"
    def get_text(text):
        return f"{tag_start}{text}{tag_end}"
    return get_text
```

Teraz złożenie[22] kilku operacji (listing 5.9) będzie działało poprawnie. Musimy tylko redefiniować nasze sparametryzowane domknięcia z listingu 5.6.

Listing 5.9. Złożenie funkcji domknięć

```
>>> print(make_italic(make_bold(text="some_text")))
<i><b>some_text</b></i>
```

5.3. Dekoratory nieparametryczne

No dobrze, teraz już chyba nic nie stoi na przeszkodzie, abyśmy omówili dekoratory.

> Zgodnie z formalną definicją można powiedzieć, że **dekorator** jest funkcją wyższego rzędu, która przyjmuje funkcję jako argument i zwraca również funkcję [18]. Udekorowana funkcja ma taką samą nazwę co funkcja oryginalna — przed udekorowaniem.

Nie jest to więc nic innego jak omówione wcześniej **domknięcie** funkcji wraz z podmianą nazw. Dekoratory wykorzystywane są w formie adnotacji umieszczanej nad funkcją lub metodą.

Wróćmy na chwilę do przykładu z listingu 4.16 zawierającego metodę klasową z pakietu *scikit-learn*. Użyty został tam dekorator `@classmethod` umieszczony nad metodą `compute`. Z uwagi na fakt, że dekorator jest po prostu domknięciem z podmianą nazw, możemy zapisać wcześniejszą definicję w alternatywny (choć mniej przyjazny) sposób (listing 5.10).

[22] Złożenie funkcji w sensie matematycznym, tj. $f_1 \circ f_2 \circ f_3 \circ \ldots$

Kup książkę Poleć książkę

Listing 5.10. Przekształcenie metody w metodę klasową z użyciem dekoratora (po lewej) oraz jawnego przekształcenia (po prawej)

```
...
@classmethod
def compute(
    cls,
    X,
    Y,
    k,
    metric="euclidean",
    chunk_size=None,
    metric_kwargs=None,
    strategy=None,
    return_distance=False,
):
    ...
```

```
...

def compute(
    cls,
    X,
    Y,
    k,
    metric="euclidean",
    chunk_size=None,
    metric_kwargs=None,
    strategy=None,
    return_distance=False,
):
    ...

compute = classmethod(compute)
```

Jak widzisz, dekoratory są stosowane dla wygody i zwiększenia czytelności kodu, czyli jako element tzw. lukru składniowego (ang. *syntactic sugar*).

Możemy powiedzieć w skrócie, że **dekorator** to domknięcie z podmianą nazw, zapisane w formacie `@<nazwa_domkniecia>` i umieszczone nad funkcją lub metodą.

Spróbujmy teraz wykorzystać naszą wcześniejszą funkcję zagnieżdżoną `make_bold()` jako dekorator. Wiemy, że dekorator nie tylko zwraca funkcję, ale musi też ją przyjmować. Mógłby więc wyglądać następująco (listing 5.11).

Listing 5.11. Przykładowy dekorator o nazwie `@make_bold`

```
1 def make_bold(func):
2     def get_text(*args, **kwargs):
3         return f"<b>{func(*args, **kwargs)}</b>"
4     return get_text
```

Linijka 1. to nagłówek funkcji zewnętrznej — nazwa naszego dekoratora. Aktualnie funkcja zewnętrzna ma tylko jeden parametr, którym jest dekorowana funkcja. Linijka 2. to nagłówek funkcji zagnieżdżonej. Przyjmuje ona dowolną liczbę argumentów pozycyjnych i nazwanych[23] (nie wiemy, jaka funkcja będzie dekorowana, jakie przyjmuje ona parametry ani w jaki sposób będzie wywołana). W linijce 3. wykonujemy przekazaną funkcję i zwracamy ciąg tekstowy otoczony tagami pogrubienia HTML.

[23] W tym przykładzie `*args` symbolizuje listę argumentów pozycyjnych, zaś `**kwargs` — argumentów nazwanych. Oczywiście ich nazwy mogą być dowolne — kluczowe jest użycie pojedynczej lub podwójnej gwiazdki.

Kup ksi k Pole ksi k

Takim dekoratorem możemy ozdobić np. funkcję zwracającą tytuł jakiegoś utworu w taki sposób, by był on pogrubiony (listing 5.12).

Listing 5.12. Użycie funkcjonalności `make_bold` jako dekoratora (po lewej) oraz jako jawnego przekształcenia funkcji (po prawej)

```
1  @make_bold
2  def get_title():
3      # np. pobranie z bazy danych
4      return "Władca Pierścieni"
5
6
```

```

def get_title():
    # np. pobranie z bazy danych
    return "Władca Pierścieni"

get_title = make_bold(get_title)
```

Zobaczmy, jaki będzie rezultat wykonania funkcji `get_title()` (listing 5.13).

Listing 5.13. Wykonanie złożenia funkcji

```
>>> print(get_title())
<b>Władca Pierścieni</b>
```

5.4. Dekoratory parametryczne

Dekorator może również przyjmować argumenty, czyli może być parametryczny. Załóżmy, że chcemy użyć naszego wcześniejszego domknięcia funkcji o nazwie `wrap_text()`, by móc skorzystać z dekoratora w takiej formie (listing 5.14):

Listing 5.14. Użycie parametrycznego dekoratora `@wrap_text`

```
1  @wrap_text("b")
2  def get_title():
3      # np. pobranie z bazy danych
4      return "Władca Pierścieni"
5  # lub
6  @wrap_text("u")
7  def get_title():
8      # np. pobranie z bazy danych
9      return "Władca Pierścieni"
```

Być może Twój pierwszy pomysł na implementację wyglądałby następująco (listing 5.15).

Listing 5.15. Błędna implementacja dekoratora parametrycznego

```
1  def wrap_text(func, tag):
2      def get_text(*args, **kwargs):
```

Kup ksi k Pole ksi k

```
        return f"<{tag}>{func(*args, **kwargs)}</{tag}>"
    return get_text
```

Ale uważaj! W przypadku powyższej implementacji (listing 5.15) próba wykonania udekorowanej funkcji (listing 5.14) zakończy się wyświetleniem informacji o błędzie typu `TypeError`[24]. Dlaczego? Ponieważ taki dekorator wymaga następującego zapisu:

```
get_title = wrap_text(get_title, "b")
```

podczas gdy poprawny parametryczny dekorator ma taką alternatywną formę:

```
get_title = wrap_text("b")(get_title)
```

Czy widzisz różnicę? Zamiast funkcji dwuargumentowej mamy tu dwa wywołania jednoargumentowe. Jak możemy to zrealizować? Odpowiedź brzmi: używając kolejnego zagnieżdżenia funkcji! Tak, to może być trochę przytłaczające i na pierwszy rzut oka — nieintuicyjne, ale mam nadzieję, że zaraz wszystko się wyjaśni. Poprawmy teraz zapis naszego parametrycznego dekoratora, żeby wyglądał tak jak na listingu 5.16.

Listing 5.16. Poprawna implementacja dekoratora `@wrap_text`

```
def wrap_text(tag):
    def inner(func):
        def get_text(*args, **kwargs):
            return f"<{tag}>{func(*args, **kwargs)}</{tag}>"
        return get_text
    return inner
```

Przeanalizujmy powyższy kod. Linijka 1. to nagłówek naszego dekoratora. Wskazaliśmy taką nazwę, jakiej chcemy użyć do dekorowania. Funkcja zewnętrzna zwraca obiekt funkcji zagnieżdżonej — `inner()`. Funkcja zagnieżdżona (linijka 2.) przyjmuje jako parametr funkcję, która będzie dekorowana, i zwraca obiekt funkcji zagnieżdżonej głębiej — `get_text()`. Ta najbardziej wewnętrzna funkcja korzysta ze zmiennych nielokalnych zarówno funkcji `inner()`, jak i funkcji zewnętrznej. Zauważ, że na listingu 5.14 w linijkach 1. oraz 6. mamy **wykonanie** funkcji `wrap_header()` (widoczne są nawiasy i argumenty w nich przekazane)!

5.5. Atrybuty specjalne funkcji dekorowanych

Może to być dla Ciebie kolejne zaskoczenie, ale funkcje w Pythonie to również obiekty. W zasadzie w Pythonie wszystko jest obiektem — nie tylko instancja klasy, ale rów-

[24] `TypeError: wrap_text() missing 1 required positional argument: 'func'`.

Kup książkę Poleć książkę

nież klasy jako takie (patrz: podrozdział Metaklasy), funkcje czy typy wbudowane (listing 5.17). Wynika to z implementacji języka.

Listing 5.17. Weryfikacja typów w Pythonie w kontekście obiektowości

```
>>> assert isinstance(None, object)
>>> assert isinstance(RuntimeError, object)
>>> assert isinstance(int, object)
>>> assert isinstance(str, object)
>>> class A: pass
>>> assert isinstance(A, object)
>>> assert isinstance(A(), object)
>>> def add(x,y):...
>>> assert isinstance(add, object)
```

W związku z tym funkcje, klasy, pakiety, typy wbudowane — wszystkie te elementy mają pewne charakterystyczne atrybuty, czyli swego rodzaju metadane. O niektórych z nich wspomnieliśmy już wcześniej przy omawianiu modułów w podrozdziale Moduły. Dla funkcji są to atrybuty: `__name__`, `__module__`, `__qualname__`, `__doc__`, `__annotations__` oraz `__dict__`. Większość z nich już znasz. Atrybut `__name__` przechowuje nazwę funkcji, `__module__` — moduł, w którym funkcja jest zdefiniowana, `__qualname__` — nazwę kwalifikowaną funkcji, `__doc__` zaś — jej dokumentację (*docstring*). Pozostałymi dwoma nie będziemy się na razie zajmować.

Dlaczego o tym mówię w kontekście dekoratorów? Ponieważ z wcześniejszymi implementacjami wiąże się ten problem, że wszystkie wspomniane atrybuty specjalne są **nadpisywane**. Oznacza to, że np. nazwa funkcji (przechowywana w atrybucie `__name__`) zostaje zmieniona na nazwę funkcji zagnieżdżonej — `get_text()` zamiast `get_title()` — co zazwyczaj nie jest pożądanym zachowaniem. Podobnie dzieje się w przypadku innych atrybutów specjalnych. I choć na pierwszy rzut oka nie jest to duży problem, może to prowadzić do sporych komplikacji. W kolejnych listingach znajdziesz dwa alternatywne rozwiązania tego problemu. Pierwsze, chyba dość oczywiste rozwiązanie to ponowne nadpisanie atrybutów (listing 5.18).

Listing 5.18. Ręczne nadpisywanie atrybutów funkcji

```
1  def wrap_text(tag):
2      def inner(func):
3          def get_text(*args, **kwargs):
4              return f"<{tag}>{func(*args, **kwargs)}</{tag}>"
5          print_text.__name__ = func.__name__
6          print_text.__doc__ = func.__doc__
7          # analogicznie nadpisuje się inne atrybuty
8          return get_text
9      return inner
```

Kup ksi k Pole ksi k

Alternatywnym i zdecydowanie wygodniejszym sposobem jest skorzystanie z tzw. fabryki dekoratorów [50] będącej częścią biblioteki standardowej Pythona. Ona nadpisze te atrybuty za nas (listing 5.19).

Listing 5.19. Wykorzystanie dekoratora `@wraps` w celu zdefiniowania własnego dekoratora

```
from functools import wraps


def wrap_text(tag):
    def inner(func):
        @wraps(func)
        def get_text(*args, **kwargs):
            return f"<{tag}>{func(*args, **kwargs)}</{tag}>"

        return get_text
    return inner
```

Jak zapewne widzisz, różnica mieści się w linijce 6. Tak, to nie jest pomyłka. Używamy innego dekoratora, żeby zbudować nasz własny.

★ Dekoratory to funkcje, więc tak jak funkcje mogą być składane:

```
@wrap_text("u")
@wrap_text("b")
def get_title():
    ...
```

co jest równoznaczne z zapisem:

```
get_title = wrap_text("u")(wrap_text("b")(get_title))
```

Teraz bez problemu jesteś w stanie:

1. definiować funkcje nielokalne,
2. wykorzystywać mechanizm zmiennych nielokalnych,
3. definiować dekoratory nieparametryczne i parametryczne,
4. definiować dekoratory z wykorzystaniem fabryki dekoratorów,
5. skutecznie wykorzystywać dekoratory.

Kup ksi k Pole ksi k

Skorowidz

Anaconda
- instalacja, 16

atrybut
- klasowy, 56
- wyszukiwanie, 58

dekorator, 64
- parametryczny, 66

dekorowanie nazw, 55
deserializacja, 131
deskryptor, 70
- danych, 74
- własności, 76

deskryptor:niedanych, 74

funkcja
- argument, 50
- argumenty nazwane, 52
- argumenty pozycyjne, 52
- domknięcie, 62–64
- paramter, 50

garbage collection, 52

instalacja pakietu
- conda, 19
- pip, 18

instrukcja, 20
interaktywny Python, 21
iterator, 93, 115

jawne typowanie, 46
Jupyter, 23

klasa, 48
- abstrakcyjna, 88
- bazowa, 80
- definiowanie nowej klasy, 49
- inicjalizator, 50, 51
- konstruktor, 50

menedżer kontekstu, 127
metaklasa, 87
- implementacja, 88

metoda
- abstrakcyjna, 88
- klasowa, 58
- niepubliczna, 54
- prywatna, 54
- specjalna, 103
- statyczna, 59

moduł, 25
- importowanie, 32

nazwa kwalifikowane, 26

obiekt, 48
- destrukcja, 52
- inicjalizacja, 51
- porównywanie obiektów, 107

pakiet, 25, 28
- przestrzeni nazw, 28
- załączanie pakietu dla publikacji, 40

powłoka Pythona, 20
programowanie zorientowane obiektowo
- kolejnośc dziedziczenia, 84
- abstrakcja, 48
- dziedziczenie, 48, 79
- dziedziczenie wielorakie, 82
- enkapsulacja, 49, 54
- polimorfizm, 49

przestrzeń nazw, 30

Kup ksi k

Pole ksi k

rodzaje, 30
Python
cechy, 12
dystrybucja, 15
historia, 13
interpreter, 15
język programowania, 12, 15

referencja
cykliczna, 53
usuwanie, 53

serializacja, 131, 142
obiektu, 136
skrypt, 21

test jednostkowy, 144
typ wyliczający, 90
nazwa, 92
wartość, 92

wyjątek, 96
blok else, 101
przechwytywanie, 100
wzbudzanie, 98
wyrażenie, 20

zintegrowane środowisko deweloperskie, 23
zalety, 23
zmienne
globalne, 32
nielokalne, 31
określanie typu, 45
znaki zachęty, 20

środowisko wirtualne, 16
tworzenie, 17

PROGRAM PARTNERSKI

— GRUPY HELION —

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

Zdobądź ostrogi programisty w Pythonie!

Python jest językiem, którego powszechnie używa się w wielu obszarach: od programowania gier, przez aplikacje webowe, po systemy analizy danych. Nic więc dziwnego, że cieszy się ogromną popularnością i jest dodatkowo wspierany przez liczną społeczność programistów, projektantów i sympatyków, którzy go rozwijają i rozszerzają.

Do grona osób aktywnie korzystających z tego języka z pewnością należy Jakub Walczak, który w książce poświęconej Pythonowi i inżynierii oprogramowania wprowadza do jego ekosystemu. Dzięki jej lekturze czytelnicy poznają ideę środowisk wirtualnych, sposoby interakcji z interpreterem czy zasady podziału projektu na moduły i pakiety. Adepci sztuki programowania znajdą tu wprowadzenie do mechanizmów pakietu pytest, który zdecydowanie ułatwia pisanie i wykonywanie testów jednostkowych. Opanują również takie zagadnienia jak klasy i obiekty, a także zaawansowane aspekty programowania obiektowego, w tym metaklasy, dziedziczenie i emulowanie szczególnych zachowań obiektów.

Autor zadbał o klarowną strukturę podręcznika, który składa się z dwunastu uporządkowanych rozdziałów. Zawarta w nich treść została uzupełniona dodatkowymi wyjaśnieniami, wskazówkami i podsumowaniami, co ułatwia przyswojenie omawianego materiału.

- **Organizacja środowiska pracy**
- **Organizacja projektu**
- **Wstęp do programowania zorientowanego obiektowo**
- **Deskryptory**
- **Dziedziczenie**
- **Mechanizm obsługi wyjątków**
- **Metody specjalne klas**
- **Dekoratory**
- **Serializacja i deserializacja**
- **Testy jednostkowe z użyciem biblioteki pytest**
- **Wytyczne dotyczące stylu**

Python od A do Z!

Jakub Walczak — rocznik 1994, w grudniu 2022 obronił doktorat z informatyki technicznej i telekomunikacji, od 2019 roku asystent w grupie pracowników badawczo-dydaktycznych na Politechnice Łódzkiej i deweloper oprogramowania naukowego w CMCC Foundation. Od kilku lat entuzjasta Pythona, aktywnie zgłębiający jego tajniki zarówno w pracy zawodowej, jak i poza nią. Miłośnik podróży, kultury hiszpańskiej i słodyczy. W wolnych chwilach mól książkowy i amator sportów. Ciągły poszukiwacz nowych wyzwań, z których ostatnim było przygotowanie tego podręcznika.