Wydanie IV

Profesjonalne programowanie w Pythonie

Poznaj najlepsze praktyki kodowania i zaawansowane koncepcje programowania

Michał Jaworski
Tarek Ziadé

Helion

Packt

Tytuł oryginału: Expert Python Programming: Master Python by learning the best coding practices and advanced programming concepts, 4th Edition

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-8743-0

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/prprp4.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/prprp4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

3

ROZDZIAŁ

Nowości w Pythonie

Jednym z najistotniejszych etapów w historii Pythona było prawdopodobnie udostępnienie wersji 3.0. Oto najważniejsze zmiany, jakie zostały w niej wprowadzone:

- rozwiązanie wielu problemów dotyczących obsługi tekstu, danych i kodowania Unicode,
- rezygnacja z klas w dawnym stylu,
- rozpoczęcie reorganizacji biblioteki standardowej,
- wprowadzenie adnotacji funkcji,
- wprowadzenie nowej składni obsługi wyjątków.

Z rozdziału 1., „Aktualny stan Pythona", wiesz już, że Python 3 nie jest zgodny z Pythonem 2. Jest to główny powód, dla którego pełna akceptacja Pythona 3 przez społeczność zajęła tak dużo czasu. Była to bolesna, choć konieczna lekcja dla głównych programistów Pythona i społeczności skupionej wokół tego języka.

Na szczęście problemy związane z przyjmowaniem Pythona 3 nie zatrzymały procesu ewolucji języka. Od 3 grudnia 2008 roku (jest to data oficjalnego udostępnienia Pythona 3.0) konsekwentnie wprowadzane są nowe poważne aktualizacje Pythona. Każda nowa wersja dodaje nowe usprawnienia do języka, biblioteki standardowej i interpretera. Ponadto od wersji 3.9 obowiązuje roczny cykl udostępniania nowych edycji. To oznacza, że co rok otrzymamy nowe funkcje i usprawnienia.

> Jeśli chcesz się dowiedzieć czegoś więcej o cyklu udostępniania wersji Pythona, zapoznaj się z dokumentem *PEP 602 — Annual Release Cycle for Python* (*https://www.python.org/dev/peps/pep-0602/*).

W tym rozdziale przyjrzymy się bliżej ewolucji Pythona w ostatnich latach. Omówimy kilka ważnych dodatków z ostatnich wersji. Spekulacyjnie wybiegniemy też w przyszłość i zaprezentujemy kilka funkcji, które zostały zaakceptowane w procesie przygotowywania dokumentów PEP i już niedługo staną się oficjalną częścią Pythona. Oto zagadnienia, którymi się zajmiemy:

- niedawne dodatki do języka,
- nie tak nowe, ale jeszcze błyszczące,
- co może nas czekać w przyszłości.

Jednak przed omówieniem nowych funkcji warto zacząć od analizy wymagań technicznych.

Wymagania techniczne

Oto omawiane w tym rozdziale pakiety Pythona, które możesz pobrać z repozytorium PyPI:

- `mypy`,
- `pyright`.

Informacje o instalowaniu pakietów znajdziesz w rozdziale 2., „Nowoczesne środowiska programistyczne Pythona".

Pliki z kodem do tego rozdziału są dostępne na stronie *https://github.com/PacktPublishing/Expert-Python-Programming-Fourth-Edition/tree/main/Chapter%203* i w witrynie wydawnictwa Helion.

Niedawne dodatki do języka

W każdej wersji Pythona wprowadzane są liczne zmiany różnego rodzaju. W prawie wszystkich edycjach Pythona pojawiają się nowe elementy składni. Jednak większość zmian związana jest z biblioteką standardową Pythona, interpreterem CPython, API Pythona i API C w interpreterze CPython. Z powodu ograniczonego miejsca nie da się ich wszystkich omówić w tej książce. Dlatego skupiamy się tylko na nowych elementach składni i nowościach w bibliotece standardowej.

W dwóch najnowszych wersjach Pythona można wskazać cztery główne zmiany w składni:

- operatory scalania i aktualizacji słownika (dodane w Pythonie 3.9),
- wyrażenia przypisania (dodane w Pythonie 3.8),
- wskazówki dotyczące typów w typach generycznych (dodane w Pythonie 3.9),
- argumenty czysto pozycyjne (dodane w Pythonie 3.9).

Te cztery mechanizmy można zaliczyć do grupy zmian poprawiających jakość życia. Nie wprowadzają one żadnych nowych paradygmatów programowania. Nie zmieniają też radykalnie sposobu pisania kodu. Umożliwiają jedynie stosowanie lepszych wzorców programistycznych lub precyzyjniejsze definiowanie API.

W ostatnich latach główni programiści Pythona koncentrowali się przede wszystkim na usuwaniu martwych lub nadmiarowych modułów z biblioteki standardowej, a nie na dodawaniu czegokolwiek nowego. Jednak od czasu do czasu wprowadzane są jakieś dodatki do biblioteki standardowej. W ostatnich dwóch wersjach zyskaliśmy dwa zupełnie nowe moduły. Są to:

- moduł `zoneinfo` do korzystania z bazy danych ze strefami czasowymi organizacji **Internet Assigned Numbers Authority** (**IANA**; dodany w Pythonie 3.9),
- moduł `graphlib` do operowania strukturami grafowymi (dodany w Pythonie 3.8).

Oba moduły mają dość małe API. Dalej omawiamy kilka obszarów, w których można jest zastosować. Najpierw jednak przyjrzyjmy się zmianom w składni wprowadzonym w Pythonie 3.8 i 3.9.

Operatory scalania i aktualizacji słownika

Python umożliwia stosowanie zestawu operatorów arytmetycznych do operowania wbudowanymi typami kontenerowymi, w tym listami, krotkami, zbiorami i słownikami.

Dla list i krotek można używać operatora dodawania + , aby złączyć dwie zmienne, pod warunkiem jednak, że są to zmienne tego samego typu. Dostępny jest też operator +=, który umożliwia modyfikowanie istniejących zmiennych. Poniższy fragment kodu zawiera przykłady złączania list i krotek w interaktywnej sesji:

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> value = [1, 2, 3]
>>> value += [4, 5, 6]
>>> value
[1, 2, 3, 4, 5, 6]
>>> value = (1, 2, 3)
>>> value += (4, 5, 6)
>>> value
(1, 2, 3, 4, 5, 6)
```

Dla zbiorów dostępne są cztery operatory dwuargumentowe, które dają nowy zbiór:

- **Operator części wspólnej** — reprezentowany przez symbol & (bitowy operator I). Tworzy zbiór z elementami występującymi i w jednym, i w drugim zbiorze jednocześnie.
- **Operator sumy** — reprezentowany przez symbol | (bitowy operator LUB). Tworzy zbiór z wszystkimi elementami z obu zbiorów.
- **Operator różnicy** — reprezentowany przez symbol - (odejmowanie). Tworzy zbiór z elementami lewego zbioru, które nie występują w prawym zbiorze.
- **Operator XOR** — reprezentowany przez symbol ^ (bitowy operator XOR). Tworzy zbiór z elementami, które występują albo w jednym zbiorze, albo w drugim, ale nie w obu jednocześnie.

Poniższy fragment kodu zawiera przykłady zastosowania operatorów części wspólnej i sumy do zbiorów w interaktywnej sesji:

```
>>> {1, 2, 3} & {1, 4}
{1}
```

```
>>> {1, 2, 3} | {1, 4}
{1, 2, 3, 4}
>>> {1, 2, 3} - {1, 4}
{2, 3}
>>> {1, 2, 3} ^ {1, 4}
{2, 3, 4}
```

W Pythonie przez bardzo długi czas nie istniał specjalny operator dwuargumentowy, który umożliwiałby uzyskanie nowego słownika na podstawie dwóch istniejących słowników. Od wersji 3.9 można używać operatorów | (bitowy operator LUB) i |= (złożony bitowy operator LUB), które pozwalają scalać i aktualizować słowniki. Należy je stosować jako idiomatyczny sposób tworzenia sumy dwóch słowników. Powody dodania tych operatorów są opisane w dokumencie *PEP 584 — Add Union Operators To Dict*.

> **Idiom programistyczny** jest typowym i zalecanym sposobem wykonywania określonego zadania w danym języku programowania. Pisanie idiomatycznego kodu jest ważną częścią kultury użytkowników Pythona. W dokumencie *Zen of Python* można przeczytać, że „powinien istnieć jeden — i najlepiej tylko jeden — oczywisty sposób na wykonanie danego zadania".
>
> Więcej idiomów omawiamy w rozdziale 4., „Porównanie Pythona z innymi językami".

Aby scalić dwa słowniki w nowy, zastosuj następujące wyrażenie:

```
słownik_1 | słownik_2
```

Wynikowy słownik to zupełnie nowy obiekt z wszystkimi kluczami z obu słowników źródłowych. Jeśli w obu słownikach występują pokrywające się klucze, w wynikowym obiekcie zachowywane są wartości z prawego obiektu.

Poniżej pokazujemy, jak zastosować tę składnię do dwóch literałów słownikowych. Lewy słownik jest modyfikowany wartościami ze słownika podanego po prawej stronie:

```
>>> {'a': 1} | {'a': 3, 'b': 2}
{'a': 3, 'b': 2}
```

Jeśli chcesz zmodyfikować zmienną reprezentującą słownik kluczami z innego słownika, możesz zastosować następujący operator złożony:

```
istniejący_słownik |= inny_słownik
```

Oto przykład zastosowania tego operatora do rzeczywistej zmiennej:

```
>>> mydict = {'a': 1}
>>> mydict |= {'a': 3, 'b': 2}
>>> mydict
{'a': 3, 'b': 2}
```

W starszych wersjach Pythona najprostszym sposobem na aktualizację istniejącego słownika zawartością innego słownika było użycie metody `update()`, tak jak w poniższym przykładzie:

```
istniejący_słownik.update(inny_słownik)
```

Ta metoda modyfikuje `istniejący_słownik` w miejscu i nie zwraca wartości. To oznacza, że nie jest możliwe proste utworzenie scalonego słownika w ramach wyrażenia; metoda ta zawsze jest stosowana jako instrukcja.

> Różnica między słowami „wyrażenie" i „instrukcja" jest wyjaśniona w podrozdziale „Wyrażenia przypisania".

Inna możliwość: rozpakowywanie słownika

Mało znanym faktem jest to, że Python już przed wersją 3.9 udostępniał dość zwięzły sposób scalania dwóch słowników. Służył do tego mechanizm **rozpakowywania słownika**. Obsługę rozpakowywania słowników w literałach typu `dict` dodano w Pythonie 3.5. Technika ta jest opisana w dokumencie *PEP 448 Additional Unpacking Generalizations*. Składnia pozwalająca rozpakować dwa słowniki (lub większą ich liczbę) do nowego obiektu wygląda tak:

```
{**słownik_1, **słownik_2}
```

Oto przykład z rzeczywistymi literałami:

```
>>> a = {'a': 1}; b = {'a':3, 'b': 2}
>>> {**a, **b}
{'a': 3, 'b': 2}
```

Ten mechanizm, podobnie jak rozpakowywanie list (składnia `*wartość`), może wyglądać znajomo dla osób, które mają doświadczenie w pisaniu tak zwanych **funkcji wariadycznych**, przyjmujących niezdefiniowany zestaw argumentów i argumenty nazwane. Składnia ta jest przydatna przede wszystkim przy pisaniu dekoratorów.

> Funkcje wariadyczne i dekoratory omawiamy szczegółowo w rozdziale 4., „Porównanie Pythona z innymi językami".

Należy zapamiętać, że rozpakowywanie słownika, choć niezwykle popularne w definicjach funkcji, jest wyjątkowo rzadko stosowaną techniką scalania słowników. Może być ona niezrozumiała dla mniej doświadczonych programistów czytających Twój kod. Dlatego w kodzie używającym Pythona 3.9 i nowszych wersji języka należy preferować nowy operator scalania. W starszych wersjach Pythona czasem lepiej jest posłużyć się tymczasowym słownikiem i prostą metodą `update()`.

Inna możliwość: ChainMap z modułu collections

Jeszcze inny sposób tworzenia obiektu, który technicznie jest połączeniem dwóch słowników, polega na użyciu klasy `ChainMap` z modułu `collections`. Jest to klasa nakładkowa, która przyjmuje kilka obiektów odwzorowań (tu są nimi słowniki) i działa tak, jakby był to jeden obiekt odwzorowań.

Składnia scalania dwóch słowników za pomocą klasy `ChainMap` wygląda tak:

```
nowe_odwzorowanie = ChainMap(słownik_2, słownik_1)
```

Zauważ, że kolejność słowników jest tu odwrotna niż w operatorze |. To oznacza, że jeśli pobierzesz określony klucz z obiektu `nowe_odwzorowanie`, opakowane obiekty będą przeszukiwane w kolejności od lewej do prawej. Przyjrzyj się poniższemu kodowi, który ilustruje operacje z użyciem klasy `ChainMap`:

```
>>> from collections import ChainMap
>>> user_account = {"iban": "GB71BARC20031885581746", "type":
"konto"}
>>> user_profile = {"display_name": "Jan Kowalski", "type": "profil"}
>>> user = ChainMap(user_account, user_profile)
>>> user["iban"]
'GB71BARC20031885581746'
>>> user["display_name"]
'Jan Kowalski'
>>> user["type"]
'konto'
```

W tym przykładzie widać, że wynikowy obiekt `user` w obiekcie typu `ChainMap` zawiera klucze z obu słowników — `user_account` i `user_profile`. Jeśli któreś z kluczy się powtarzają, obiekt typu `ChainMap` zwróci wartość z pierwszego od lewej odwzorowania, które zawiera dany klucz. Operator scalania słowników działa więc na odwrót.

`ChainMap` jest obiektem nakładkowym. To oznacza, że nie kopiuje zawartości przekazanych mu źródłowych odwzorowań, tylko zapisuje je jako referencje. Dlatego jeśli podstawowy obiekt się zmieni, `ChainMap` może zwrócić zmodyfikowane dane. Przyjrzyj się poniższej kontynuacji wcześniejszej sesji interaktywnej:

```
>>> user["display_name"]
'Jan Kowalski'
>>> user_profile["display_name"] = "Adam Mickiewicz"
>>> user["display_name"]
'Adam Mickiewicz'
```

Ponadto obiekty klasy `ChainMap` umożliwiają zapis i zachowują zmiany w powiązanym odwzorowaniu. Należy jednak pamiętać, że operacje zapisu, aktualizacji i usuwania dotyczą tylko lewego odwzorowania. Jeśli nie zachowasz należytej ostrożności, mogą wystąpić zaskakujące sytuacje, tak jak w poniższej kontynuacji poprzedniej sesji:

```
>>> user["display_name"] = "Jan Kowalski"
>>> user["age"] = 33
>>> user["type"] = "rozszerzenie"
>>> user_profile
{'display_name': 'Adam Mickiewicz', 'type': 'profil'}
>>> user_account
{'iban': 'GB71BARC20031885581746', 'type': 'rozszerzenie', 'display_name':
'Jan Kowalski', 'age': 33}
```

W tym przykładzie widać, że klucz `'display_name'` jest zapisywany w słowniku `user_account`, a początkowym słownikiem źródłowym tego klucza był `user_profile`. W wielu kontekstach takie zapisywanie danych w źródłowych słownikach w obiekcie klasy `ChainMap` jest niepożądane.

Dlatego często stosowany idiom związany z używaniem tych obiektów do scalania dwóch słowników obejmuje jawną konwersję na nowy słownik. Oto przykład dotyczący wcześniej zdefiniowanych słowników wejściowych:

```
>>> dict(ChainMap(user_account, user_profile))
{'display_name': 'Jan Kowalski', 'type': 'konto', 'iban':
'GB71BARC20031885581746'}
```

Jeśli chcesz tylko scalić dwa słowniki, wybierz nowy operator scalania zamiast klasy `ChainMap`. Nie oznacza to jednak, że takie obiekty stały się bezużyteczne. Jeżeli potrzebne jest wprowadzanie zmian w obie strony, należy posłużyć się klasą `ChainMap`. Ponadto działa ona dla odwzorowań dowolnego typu. Dlatego jeśli chcesz zapewnić ujednolicony dostęp do wielu obiektów działających tak jak słowniki, `ChainMap` umożliwia utworzenie jednostki działającej jak efekt ich scalenia.

> Jeśli masz niestandardową klasę reprezentującą dzienniki, zawsze możesz rozszerzyć ją o specjalną metodę `__or__()`, aby zapewnić zgodność z operatorem `|`, zamiast używać klasy `ChainMap`. Przesłanianie metod specjalnych omawiamy w rozdziale 4., „Porównanie Pythona z innymi językami". Jednak użycie klasy `ChainMap` jest zwykle łatwiejsze niż pisanie niestandardowej metody `__or__()` i umożliwia pracę z istniejącymi instancjami klas, których nie możesz zmodyfikować.

Zazwyczaj najważniejszym powodem stosowania klasy `ChainMap` zamiast rozpakowywania słowników lub operatora sumy jest zachowanie zgodności wstecz. W wersjach Pythona starszych niż 3.9 nie można stosować nowego operatora scalania słowników. Dlatego jeśli musisz pisać kod za pomocą starszych wersji Pythona, użyj klasy `ChainMap`. W nowych wersjach języka lepiej jest korzystać z operatora scalania.

Inną zmianą składniową, która ma duży wpływ na zgodność wstecz, są wyrażenia przypisania.

Wyrażenia przypisania

Jest to ciekawy mechanizm, ponieważ jego wprowadzenie wpływa na jedną z podstawowych cech składni Pythona — rozróżnienie na wyrażenia i instrukcje. Wyrażenia i instrukcje są podstawowymi cegiełkami prawie każdego języka programowania. Różnica między nimi jest prosta: wyrażenia mają wartość, a instrukcje nie.

Instrukcje możesz traktować jak kolejne operacje, które wykonuje program. Tak więc przypisania wartości, klauzule `if` lub pętle `for` i `while` są instrukcjami. Także definicje funkcji i klas są instrukcjami.

Wyrażenie to cokolwiek, co można umieścić w klauzuli `if`. Typowymi przykładami wyrażeń są literały, wartości zwracane przez operatory (w tym operatory złożone) oraz wyrażenia listowe, słownikowe i zbiorowe. Także wywołania funkcji i metod są wyrażeniami.

Niektóre elementy wielu języków programowania są prawie zawsze instrukcjami. Często są to:

- definicje funkcji i klas,
- pętle,
- klauzule `if…else`,
- przypisania wartości.

W Pythonie udało się wyeliminować opisane rozróżnienie dzięki udostępnieniu składni, która pozwala stosować odpowiedniki takich elementów języka mające postać wyrażeń:

- Wyrażenia lambda do tworzenia funkcji anonimowych są odpowiednikiem definicji funkcji:

```
lambda x: x**2
```

- Tworzenie instancji typu `type` jest odpowiednikiem definicji klasy:

```
type("MyClass", (), {})
```

- Różne wyrażenia składania kolekcji (ang. *comprehensions*) są odpowiednikiem pętli:

```
squares_of_2 = [x**2 for x in range(10)]
```

- Wyrażenia złożone są odpowiednikiem instrukcji `if … else`:

```
"nieparzysta" if number % 2 else "parzysta"
```

Jednak przez wiele lat niedostępna była składnia, która pozwalałaby przypisywać wartość do zmiennej w formie wyrażenia. Była to bez wątpienia świadoma decyzja projektowa twórców Pythona. W językach takich jak C, gdzie przypisanie do zmiennej może mieć postać zarówno wyrażenia, jak i instrukcji, często się zdarza, że operator przypisania jest mylony z operatorem równości. Każdy, kto programował w C, może potwierdzić, że jest to irytujące źródło błędów. Przyjrzyj się przykładowemu kodowi w C:

```
int err = 0;
if (err = 1) {
    printf("Wystąpił błąd");
}
```

Porównaj go z poniższym fragmentem:

```
int err = 0;
if (err == 1) {
    printf("Wystąpił błąd");
}
```

Obie wersje są składniowo poprawne w C, ponieważ `err = 1` jest w tym języku wyrażeniem o wartości 1. Porównaj to z Pythonem, gdzie poniższy kod spowoduje błąd składniowy:

```
err = 0
if err = 1:
    printf("Wystąpił błąd")
```

Jednak w rzadkich sytuacjach wygodne może być używanie operatora przypisania do zmiennej, który zwraca wartość. Na szczęście w Pythonie 3.8 wprowadzono specjalny operator `:=`, który przypisuje wartość do zmiennej, ale działa jak wyrażenie, a nie jak instrukcja. Z powodu wyglądu został on szybko nazwany operatorem **morsa**.

Trzeba przy tym zauważyć, że zastosowania tego operatora są dość ograniczone. Pomaga on pisać bardziej zwięzły kod, a taki kod często jest łatwiejszy do zrozumienia, ponieważ ma lepszy stosunek sygnału do szumu. Najczęściej stosuje się go wtedy, kiedy trzeba wykonać skomplikowane obliczenia wartości, a następnie natychmiast posłużyć się nią w dalszych instrukcjach.

Często podawanym przykładem jest używanie wyrażeń regularnych. Wyobraź sobie prostą aplikację, która wczytuje kod źródłowy napisany w Pythonie i sprawdza go za pomocą wyrażeń regularnych, szukając zaimportowanych modułów.

Bez wyrażeń przypisania kod może wyglądać tak:

```
import os
import re
import sys

import_re = re.compile(
    r"^\s*import\s+\.{0,2}((\w+\.)*(\w+))\s*$"
)
import_from_re = re.compile(
    r"^\s*from\s+\.{0,2}((\w+\.)*(\w+))\s+import\s+(\w+|\*)+\s*$"
)

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print(f"użytkowanie: {os.path.basename(__file__)} file-name")
        sys.exit(1)

    with open(sys.argv[1]) as file:
        for line in file:
            match = import_re.search(line)
            if match:
                print(match.groups()[0])

            match = import_from_re.search(line)
            if match:
                print(match.groups()[0])
```

Widać tu, że trzeba dwukrotnie powtórzyć kod, który sprawdza dopasowanie złożonych wyrażeń, a następnie pobiera pogrupowane tokeny. Za pomocą wyrażeń przypisania ten blok kodu można zmodyfikować w następujący sposób:

```
if match := import_re.match(line):
    print(match.groups()[0])

if match := import_from_re.match(line):
    print(match.groups()[0])
```

Widać więc niewielką poprawę czytelności, ale nie jest ona istotna. Tego rodzaju zmiany są przydatne w sytuacjach, gdy ten sam wzorzec trzeba powtórzyć wielokrotnie. Ciągłe przypisywanie tymczasowych wyników do tej samej zmiennej może sprawić, że kod będzie niepotrzebnie rozbudowany.

Innym zastosowaniem wyrażeń przypisania jest używanie tych samych danych w wielu miejscach w większych wyrażeniach. Przyjrzyj się literałowi słownikowemu, który reprezentuje dane na temat fikcyjnego użytkownika:

```
first_name = "Jan"
last_name = "Kowalski"
height = 168
weight = 70

user = {
    "first_name": first_name,
    "last_name": last_name,
    "display_name": f"{first_name} {last_name}",
    "height": height,
    "weight": weight,
    "bmi": weight / (height / 100) ** 2,
}
```

Załóżmy, że ważne jest zachowanie spójności wszystkich elementów. Dlatego wyświetlane nazwisko zawsze powinno obejmować imię i nazwisko, a wskaźnik BMI należy obliczać na podstawie wagi i wzrostu. Aby zapobiec pomyłkom w trakcie edycji poszczególnych komponentów danych, trzeba zdefiniować je jako odrębne zmienne. Jednak po utworzeniu słownika nie są one potrzebne. Wyrażenia przypisania umożliwiają zapisanie tego słownika w bardziej zwięzły sposób:

```
user = {
    "first_name": (first_name := "Jan"),
    "last_name": (last_name := "Kowalski"),
    "display_name": f"{first_name} {last_name}",
    "height": (height := 168),
    "weight": (weight := 70),
    "bmi": weight / (height / 100) ** 2,
}
```

Widać tu, że wyrażenia przypisania zostały umieszczone w nawiasach. Niestety składnia `:=` powoduje konflikt ze znakiem `:` używanym jako operator przypisania wartości do klucza w literałach słownikowych. Nawiasy pozwalają rozwiązać ten problem.

Wyrażenia przypisania są narzędziem do „dopieszczania" kodu i niczym więcej. Zawsze sprawdzaj, czy ich zastosowanie rzeczywiście poprawia czytelność, a nie zmniejsza ją.

Wskazówki dotyczące typów w typach generycznych

Adnotacje określające typ są w pełni opcjonalnym mechanizmem Pythona, ale zyskują coraz większą popularność. Dodaje się je do zmiennych, argumentów i typu wartości zwracanych przez funkcję. Tego rodzaju adnotacje pełnią funkcję dokumentacji, ale można też wykorzystać

je do sprawdzania poprawności kodu za pomocą zewnętrznych narzędzi. Wiele środowisk IDE potrafi przetwarzać adnotacje dotyczące typów i wizualnie wskazywać możliwe problemy z typami. Dostępne są też narzędzia do statycznego sprawdzania typów, na przykład **mypy** i **pyright**, które można wykorzystać do zbadania całego kodu bazowego i wskazania wszystkich błędów związanych z typami w jednostkach kodu opatrzonych adnotacjami.

> Bardzo ciekawa jest historia projektu mypy. Początkowo powstał on w ramach rozprawy doktorskiej Jukki Lehtosalo, ale zaczął nabierać kształtu, gdy Jukka zaczął pracować nad nim razem z Guidem van Rossumem (twórcą Pythona) w firmie Dropbox. Więcej na ten temat dowiesz się z listu pożegnalnego skierowanego do Guido (*https://blog.dropbox.com/topics/company/thank-you--guido*).

W najprostszym podejściu wskazówki dotyczące typów można stosować do podawania oczekiwanych typów dla wbudowanych lub niestandardowych typów, argumentów wejściowych funkcji, wartości zwracanych przez funkcje i zmiennych lokalnych. Przyjrzyj się poniższej funkcji. Umożliwia ona wyszukiwanie kluczy bez uwzględniania wielkości liter w słowniku, którego kluczami są łańcuchy znaków:

```
from typing import Any

def get_ci(d: dict, key: str) -> Any:
    for k, v in d.items():
        if key.lower() == k.lower():
            return v
```

> Ten przykład jest oczywiście naiwną implementacją wyszukiwania bez uwzględniania wielkości liter. Aby uzyskać wyższą wydajność, prawdopodobnie trzeba zastosować specjalną klasę. Do tego problemu wracamy w dalszej części książki.

Pierwsza instrukcja importuje typ `Any` z modułu `typing`. Ten typ definiuje, że zmienna lub argument może być dowolnego typu. W sygnaturze funkcji określone jest, że pierwszy argument, `d`, powinien być słownikiem, a drugi, `key`, łańcuchem znaków. Sygnatura kończy się specyfikacją zwracanej wartości, która może być dowolnego typu.

Jeśli używasz narzędzi do sprawdzania typów, takie adnotacje wystarczą do wykrycia wielu pomyłek. Jeżeli na przykład w jednostce wywołującej zostanie przestawiona kolejność argumentów pozycyjnych, będzie można szybko wykryć błąd, ponieważ adnotacje argumentów `key` i `d` określają inne typy. Jednak te narzędzia nie zgłoszą problemu w sytuacji, gdy użytkownik przekaże słownik używający różnych typów dla kluczy.

Z tego powodu typy generyczne takie jak `tuple`, `list`, `dict`, `set`, `frozenset` i wiele innych można opatrzyć adnotacjami określającymi typ zawartości. Dla słowników takie adnotacje mają następującą postać:

```
dict[TypKlucza, TypWartości]
```

Sygnatura funkcji `get_ci()` z bardziej restrykcyjnymi adnotacjami określającymi typ może wyglądać tak:

```
def get_ci(d: dict[str, Any], key: str) -> Any: ...
```

W starszych wersjach Pythona nie można było tak łatwo dodawać adnotacji określających typ zawartości wbudowanych kolekcji. Moduł `typing` obejmuje specjalne typy, które można stosować w tym celu. Oto niektóre z tych typów:

- `typing.Dict` dla słowników,
- `typing.List` dla list,
- `typing.Tuple` dla krotek,
- `typing.Set` dla zbiorów,
- `typing.FrozenSet` dla zablokowanych zbiorów.

Te typy są przydatne, jeśli kod ma działać w różnych wersjach języka. Jeżeli jednak piszesz kod dostosowany tylko do Pythona 3.9 i nowszych wersji, należy używać wbudowanych typów generycznych. Importowanie ich z modułu `typing` to przestarzałe rozwiązanie, które w przyszłości zostanie wycofane z Pythona.

> Adnotacje określające typ omawiamy dokładniej w rozdziale 4., „Porównanie Pythona z innymi językami".

Parametry czysto pozycyjne

Python daje dużą swobodę, jeśli chodzi o przekazywanie argumentów do funkcji. Są dwa sposoby przekazywania argumentów do funkcji:

- w formie **argumentów pozycyjnych**,
- za pomocą **nazw argumentów**.

W wielu funkcjach to programista jednostki wywołującej decyduje o tym, jak przekazywać argumenty. Jest to korzystne, ponieważ użytkownik funkcji może uznać, że w danej sytuacji dana technika jest czytelniejsza lub wygodniejsza. Przyjrzyj się funkcji, która złącza łańcuchy znaków z użyciem określonego separatora:

```
def concatenate(first: str, second: str, delim: str):
    return delim.join([first, second])
```

Funkcję tę można wywołać na wiele sposobów:

- za pomocą argumentów pozycyjnych — `concatenate("Jan", "Kowalski", " ")`,
- za pomocą nazw argumentów — `concatenate(first="Jan", second="Kowalski", delim=" ")`,
- łącząc argumenty pozycyjne i nazwy argumentów — `concatenate("Jan", "Kowalski", delim=" ")`.

Jeśli piszesz bibliotekę do wielokrotnego użytku, możliwe, że wiesz, w jaki sposób będzie ona używana. Czasem z doświadczenia wiadomo, że określony wzorzec wywołań zwiększy lub zmniejszy czytelność wynikowego kodu. Być może jeszcze nie masz pewności co do projektu lub chcesz się upewnić, że API biblioteki można będzie w bliskiej przyszłości zmienić bez wpływu na jej użytkowników. W obu sytuacjach warto tworzyć sygnatury funkcji tak, aby zachęcać do zalecanego sposobu użytkowania, a jednocześnie umożliwić wprowadzanie zmian w przyszłości.

Po opublikowaniu biblioteki sygnatura funkcji stanowi kontrakt jej użytkowania. Każda zmiana nazw argumentów lub ich kolejności może spowodować błędy w aplikacjach programistów, którzy korzystają z danej biblioteki.

Jeśli w przyszłości stwierdzisz, że nazwy argumentów `first` i `second` nie wyjaśniają poprawnie ich przeznaczenia, nie będzie można ich zmienić bez naruszania zgodności wstecz. Wynika to z tego, że jakiś programista mógł zastosować następujące wywołanie:

```
concatenate(first="Jan", second="Kowalski", delim=" ")
```

Jeżeli zechcesz przekształcić funkcję na postać, która przyjmuje dowolną liczbę łańcuchów znaków, też nie będzie to wykonalne, ponieważ jakiś programista mógł napisać poniższy kod:

```
concatenate("Jan", "Kowalski", " ")
```

Na szczęście w Pythonie 3.8 dodano możliwość definiowania określonych argumentów jako czysto pozycyjnych. Dzięki temu możesz wskazać, których argumentów nie można przekazywać za pomocą nazw. Pomaga to uniknąć późniejszych problemów z zachowaniem zgodności wstecz. Możesz też oznaczyć, że określone argumenty można wywoływać tylko za pomocą nazw. Staranne ustalenie, które argumenty należy przekazywać tylko jako pozycyjne, a które tylko za pomocą nazw, powoduje, że w przyszłości funkcję łatwiej będzie zmodyfikować. Funkcja `concatenate()` z argumentami czysto pozycyjnymi i podawanymi tylko za pomocą nazw może wyglądać tak:

```
def concatenate(first: str, second: str, /, *, delim: str):
    return delim.join([first, second])
```

Tę definicję należy czytać następująco:

- wszystkie argumenty przed znakiem / są argumentami czysto pozycyjnymi;
- wszystkie argumenty po znaku * są argumentami podawanymi tylko za pomocą nazw.

Ta definicja gwarantuje, że jedynym poprawnym wywołaniem funkcji `concatenate()` jest następująca postać:

```
concatenate("Jan", "Kowalski", delim=" ")
```

Jeśli spróbujesz wywołać ją w inny sposób, wystąpi błąd `TypeError`:

```
>>> concatenate("Jan", "Kowalski", " ")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: concatenate() takes 2 positional arguments but 3 were given
```

Załóżmy, że funkcja w tej postaci została opublikowana w bibliotece. Teraz chcesz, aby przyjmowała ona nieograniczoną liczbę argumentów pozycyjnych. Ponieważ istnieje tylko jeden sposób używania tej funkcji, w celu wprowadzenia tej zmiany możesz zastosować rozpakowywanie argumentów:

```
def concatenate(*items, delim: str):
    return delim.join(items)
```

Argument `*items` reprezentuje wszystkie argumenty pozycyjne z krotki `items`. Dzięki takim zmianom użytkownicy będą mogli korzystać z tej funkcji z różną liczbą argumentów pozycyjnych, tak jak w poniższych przykładach:

```
>>> concatenate("Jan", "Kowalski", delim=" ")
'Jan Kowalski'
>>> concatenate("Ronald", "Reuel", "Tolkien", delim=" ")
'Ronald Reuel Tolkien'
>>> concatenate("Jacek", delim=" ")
'Jacek'
>>> concatenate(delim=" ")
''
```

Argumenty czysto pozycyjne i podawane tylko za pomocą nazw to doskonałe narzędzia dla autorów bibliotek, ponieważ umożliwiają późniejsze wprowadzanie w projekcie zmian, które nie dotykają użytkowników. Są też znakomitym narzędziem w trakcie pisania aplikacji, zwłaszcza jeśli współpracujesz z innymi programistami. Możesz wykorzystać argumenty czysto pozycyjne i podawane tylko za pomocą nazw, aby mieć pewność, że funkcja będzie wywoływana w oczekiwany sposób. Ułatwi to późniejszą refaktoryzację kodu.

Moduł zoneinfo

Obsługa czasu i stref czasowych jest jednym z największych wyzwań w programowaniu. Głównym tego powodem są liczne błędne założenia programistów w tym obszarze. Inną przyczyną są niekończące się zmiany definicji stref czasowych. Te zmiany mogą pojawiać się każdego roku i często są wprowadzane z przyczyn politycznych.

Python od wersji 3.9 zapewnia łatwiejszy niż kiedykolwiek wcześniej dostęp do informacji na temat bieżących i historycznych stref czasowych. Biblioteka standardowa Pythona udostępnia moduł `zoneinfo`, który jest interfejsem dla bazy danych ze strefami czasowymi dostępnej w systemie operacyjnym lub pobieranej jako pakiet standardowy `tzdata` z PyPI.

> Pakiety z PyPI są uznawane za pakiety zewnętrzne, natomiast moduły biblioteki standardowej to pakiety standardowe. Pakiet `tzdata` jest wyjątkowy, ponieważ zarządzają nim główni programiści implementacji CPython. Zawartość bazy danych IANA została przeniesiona do odrębnych pakietów w PyPI, aby zagwarantować regularne aktualizacje niezależne od częstotliwości udostępniania implementacji CPython.

Aby użyć tego modułu, należy utworzyć obiekt typu `ZoneInfo` za pomocą następującego wywołania konstruktora:

```
ZoneInfo(klucz_strefy_czasowej)
```

W tym kodzie `klucz_strefy_czasowej` to nazwa pliku z bazy danych stref czasowych organizacji IANA. Te nazwy plików odpowiadają sposobowi, w jaki strefy czasowe są często prezentowane w różnych aplikacjach. Oto kilka przykładów:

- `Europe/Warsaw`,
- `Asia/Tel_Aviv`,
- `America/Fort_Nelson`,
- `GMT-0`.

Instancje klasy `ZoneInfo` mogą być używane jako parametr `tzinfo` w konstruktorze obiektów `datetime`:

```
from datetime import datetime
from zoneinfo import ZoneInfo

dt = datetime(2020, 11, 28, tzinfo=ZoneInfo("Europe/Warsaw"))
```

W ten sposób można tworzyć obiekty `datetime` uwzględniające strefę czasową. Takie obiekty są niezbędne do poprawnego obliczania różnic między określonymi strefami czasowymi, ponieważ potrafią uwzględniać takie kwestie jak czas letni i zimowy, a także historyczne zmiany wprowadzane w bazie danych stref czasowych organizacji IANA.

Pełną listę wszystkich stref czasowych dostępnych w systemie możesz pobrać za pomocą funkcji `zoneinfo.available_timezones()`.

Moduł graphlib

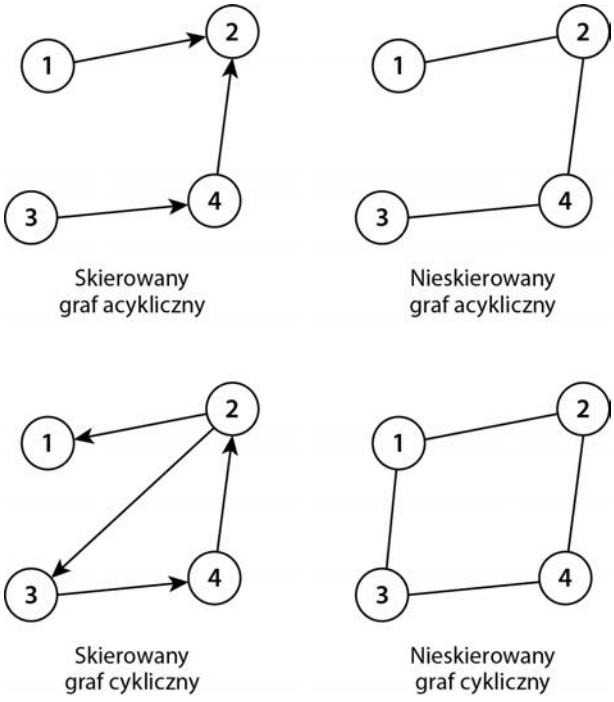
Innym ciekawym dodatkiem do biblioteki standardowej Pythona jest moduł `graphlib`, dodany w Pythonie 3.9. Ten moduł zapewnia narzędzia do pracy ze strukturami grafowymi.

Graf jest strukturą danych składającą się z węzłów połączonych krawędziami. Grafy są koncepcją z dziedziny matematyki nazywanej **teorią grafów**. W zależności od typu krawędzi można wyróżnić dwa podstawowe rodzaje grafów:

- **Graf nieskierowany** to graf, w którym wszystkie krawędzie są nieskierowane. Jeśli graf reprezentuje system miast połączonych drogami, krawędzie w grafie nieskierowanym są drogami dwukierunkowymi, którymi można się poruszać w obie strony. Dlatego jeżeli dwa węzły, *A* i *B*, są połączone krawędzią *E* w grafie nieskierowanym, można przejść z *A* do *B* i z *B* do *A*, używając tej samej krawędzi *E*.
- **Graf skierowany cechuje się tym, że każda krawędź jest w nim skierowana. Jeśli graf reprezentuje system miast połączonych drogami, krawędzie w grafie skierowanym odpowiadają drogom jednokierunkowym, którymi można podróżować**

tylko z jednego punktu źródłowego. Jeżeli dwa węzły, *A* i *B*, są połączone jedną krawędzią *E*, która wychodzi z węzła *A*, ta krawędź umożliwia przejście z *A* do *B*, ale już nie z *B* do *A*.

Ponadto grafy mogą być cykliczne lub acykliczne. **Graf cykliczny** ma przynajmniej jeden cykl — zamkniętą ścieżkę, która rozpoczyna się i kończy w tym samym węźle. **Graf acykliczny** nie obejmuje żadnych cykli. Na rysunku 3.1 pokazana jest przykładowa reprezentacja grafów skierowanych i nieskierowanych.



Rysunek 3.1. Wizualna reprezentacja różnych rodzajów grafów

Teoria grafów analizuje wiele problemów matematycznych, których modele można przedstawić za pomocą grafu. W programowaniu grafy są używane do rozwiązywania wielu problemów algorytmicznych. W informatyce grafy można wykorzystać do reprezentowania przepływu danych lub relacji między obiektami i mają wiele praktycznych zastosowań. Oto niektóre z nich:

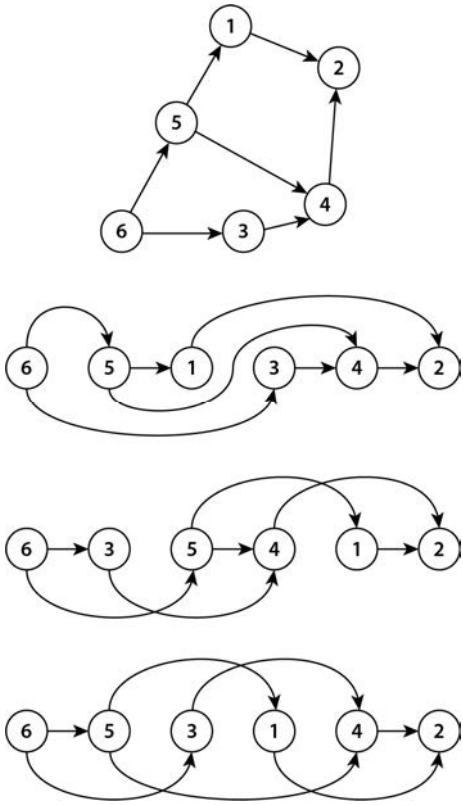
- modelowanie drzew zależności,
- reprezentowanie wiedzy w formacie czytelnym dla maszyn,
- wizualizowanie informacji,
- modelowanie systemów transportu.

Moduł `graphlib` ma ułatwiać programistom Pythona pracę z grafami. Jest to nowy moduł, dlatego obecnie obejmuje jedną klasę narzędziową, `TopologicalSorter`. Zgodnie z nazwą ta klasa przeprowadza sortowanie topologiczne skierowanych grafów acyklicznych.

Sortowanie topologiczne jest operacją porządkowania węzłów **skierowanych grafów acyklicznych** w określony sposób. Wynikiem sortowania topologicznego jest lista wszystkich węzłów, na której każdy węzeł pojawia się przed wszystkimi węzłami, do jakich można z niego przejść. Oznacza to, że:

- jako pierwszy zapisany jest węzeł, do którego nie można przejść z żadnego innego węzła;
- z każdego następnego węzła nie można przejść do poprzednich węzłów;
- z ostatniego węzła nie można przejść do żadnego innego.

W niektórych grafach występuje wiele uporządkowań zgodnych z warunkami sortowania topologicznego. Rysunek 3.2 przedstawia przykładowy skierowany graf acykliczny z trzema możliwymi porządkami topologicznymi.



Rysunek 3.2. Różne sposoby topologicznego sortowania tego samego grafu

Aby lepiej zrozumieć zastosowania sortowania topologicznego, rozważ następujący problem. Istnieje złożona operacja, która składa się z wielu zależnych od siebie zadań. Tą operacją może być na przykład przenoszenie wielu tabel bazy danych między dwoma różnymi systemami bazodanowymi. Jest to często występujący problem i istnieje już wiele narzędzi umożliwiających migrację danych do różnych systemów zarządzania bazami. Jednak w celach edukacyjnych załóżmy, że takie rozwiązanie nie istnieje i trzeba zbudować je od podstaw.

W relacyjnym systemie bazodanowym wiersze w tabelach często zawierają referencje do wierszy z innych tabel. Integralność takich referencji jest chroniona za pomocą **ograniczeń klucza obcego**. Jeśli chcesz zagwarantować, że w danym momencie docelowa baza danych zachowuje integralność referencyjną, musisz przenosić wszystkie tabele w określonej kolejności. Załóżmy, że używane są następujące tabele:

- tabela `customers` zawierająca informacje o klientach;
- tabela `accounts` przechowująca informacje o kontach użytkowników, w tym o stanie tych kont; jeden użytkownik może mieć wiele kont (na przykład osobiste i firmowe), ale jedno konto nie może być przypisane do kilku użytkowników;
- tabela `products` z informacjami na temat produktów dostępnych w sprzedaży w systemie;
- tabela `orders` obejmująca pojedyncze zamówienia wielu produktów złożone z jednego konta przez jednego użytkownika;
- tabela `order_products` z danymi o liczbie poszczególnych produktów w jednym zamówieniu.

Python nie ma specjalnego typu danych reprezentującego grafy. Dostępny jest jednak typ słownikowy, który świetnie nadaje się do zapisywania relacji między kluczami i wartościami. Zdefiniuj referencje między wymienionymi fikcyjnymi tabelami:

```
table_references = {
    "customers": set(),
    "accounts": {"customers"},
    "products": set(),
    "orders": {"accounts", "customers"},
    "order_products": {"orders", "products"},
}
```

Jeśli graf referencji nie ma cykli, można go posortować topologicznie. Wynikiem tego sortowania jest kolejność przenoszenia tabel. Konstruktor klasy `graphlib.TopologicalSorter` przyjmuje jako dane wyjściowe jeden słownik, w którym kluczami są węzły źródłowe, a wartościami — zbiory węzłów docelowych. To oznacza, że możesz przekazać zmienną `tabel_references` bezpośrednio do konstruktora klasy `TopologicalSorter()`. W celu przeprowadzenia sortowania topologicznego możesz użyć wywołania `static_order()`, tak jak w poniższym zapisie sesji interaktywnej:

```
>>> from graphlib import TopologicalSorter
>>> table_references = {
...     "customers": set(),
...     "accounts": {"customers"},
...     "products": set(),
...     "orders": {"accounts", "customers"},
...     "order_products": {"orders", "products"},
... }
>>> sorter = TopologicalSorter(table_references)
>>> list(sorter.static_order())
['customers', 'products', 'accounts', 'orders', 'order_products']
```

Sortowanie topologiczne można wykonywać tylko na skierowanych grafach acyklicznych. Klasa `TopologicalSorter` w trakcie inicjalizacji nie sprawdza występowania cykli, choć wykrywa cykle na etapie sortowania. Po znalezieniu cyklu metoda `static_order()` zgłasza wyjątek `graphlib.CycleError`.

> Ten przykład jest oczywiście prosty i łatwo można wykonać zadanie ręcznie. Jednak prawdziwe bazy danych często zawierają dziesiątki, a nawet setki tabel. Ręczne przygotowanie planu dla tak dużych baz to bardzo żmudne zadanie, w którym łatwo o pomyłki.

Mechanizmy omówione do tego miejsca są stosunkowo nowe, dlatego musi minąć trochę czasu, zanim staną się powszechnie stosowane w Pythonie. Wynika to z tego, że nie są zgodne wstecz, a wiele osób zajmujących się konserwacją bibliotek nadal zapewnia obsługę starszych wersji Pythona.

W następnym podrozdziale omawiamy ważne elementy Pythona wprowadzone w wersjach 3.6 i 3.7, a więc dostępne w większej liczbie wersji. Jednak nie wszystkie te nowe elementy są popularne, dlatego mamy nadzieję, że nauczysz się czegoś nowego.

Nie tak nowe, ale wciąż błyszczące

Każda wersja Pythona zawiera jakieś nowości. Niektóre zmiany są prawdziwym przełomem — znacznie ułatwiają programowanie i są prawie natychmiast przyjmowane przez społeczność. Jednak zalety innych modyfikacji nie zawsze są od razu oczywiste, dlatego ich spopularyzowanie zajmuje więcej czasu.

Dotyczy to na przykład adnotacji funkcji, które są dostępne w Pythonie już od wersji 3.0. Musiały minąć lata, aby powstał ekosystem narzędzi korzystających z takich adnotacji. Obecnie adnotacje są stosowane w prawie wszystkich nowoczesnych aplikacjach Pythona.

Główni programiści Pythona bardzo konserwatywnie podchodzą do dodawania nowych modułów do biblioteki standardowej, dlatego rzadko pojawiają się w niej nowości. Możliwe, że zapomnisz o modułach `graphlib` i `zoneinfo`, jeśli nie będziesz mieć okazji do pracy nad problemami, które wymagają operowania grafami lub starannego uwzględniania stref czasowych. Możliwe, że już nie pamiętasz o innych przydatnych dodatkach do Pythona, które zostały wprowadzone w kilku ostatnich latach. Dlatego przedstawiamy krótki przegląd ważnych zmian wprowadzonych do wersji 3.7. Są to albo małe, ale interesujące dodatki, które łatwo jest przeoczyć, albo rozwiązania, do których trzeba się przyzwyczaić.

Funkcja breakpoint()

Debugery omówiliśmy w rozdziale 2., „Nowoczesne środowiska programistyczne Pythona”. Wspomnieliśmy tam o funkcji `breakpoint()` jako o idiomatycznym sposobie pracy z debugerem Pythona.

Funkcja ta została wprowadzona w Pythonie 3.7, dlatego jest dostępna już od jakiegoś czasu. Jest to jednak jedna ze zmian, do których przyzwyczajenie się wymaga trochę wysiłku. Przez wiele lat uczono nas, że najprostszym sposobem wywoływania debugera w kodzie Pythona jest następujący fragment:

```
import pdb; pdb.set_trace()
```

Ten kod nie jest ani zgrabny, ani prosty, ale jeśli ktoś — podobnie jak wielu innych programistów — stosował go codziennie przez wiele lat, posługuje się nim automatycznie. Wystąpił problem? Wystarczy przejść do kodu, wpisać kilka znaków, aby wywołać debuger `pdb`, a potem ponownie uruchomić program. Znajdziesz się wtedy w powłoce interpretera w miejscu wystąpienia błędu. Gotowe? Wystarczy wrócić do kodu, usunąć fragment `import pdb; pdb.set_trace()`, a następnie zacząć pracę nad poprawkami.

Dlaczego więc masz się tym przejmować? Czy to nie kwestia indywidualnych preferencji? Czy punkty przerwania są czymś, co kiedykolwiek trafia do kodu produkcyjnego?

Prawda jest taka, że debugowanie często jest samotniczym i bardzo osobistym zadaniem. Często przez wiele godzin zmagamy się z błędami, szukamy wskazówek i w kółko czytamy ten sam kod, desperacko starając się zlokalizować drobny błąd, przez który aplikacja nie działa. Gdy koncentrujesz się na znalezieniu przyczyny problemu, zdecydowanie warto używać najwygodniejszego dla Ciebie narzędzia. Niektórzy programiści preferują debugery zintegrowane ze środowiskiem IDE. Inni w ogóle nie korzystają z debugerów i wolą rozbudowane wywołania `print()` w różnych miejscach kodu. Zawsze wybieraj to, co jest dla Ciebie najwygodniejsze.

Jeśli jednak masz w zwyczaju używać staromodnego debugera działającego w powłoce, funkcja `breakpoint()` może ułatwić Ci pracę. Główną zaletą tej funkcji jest to, że nie jest powiązana z jednym debugerem. Domyślnie uruchamia ona sesję debugera `pdb`, ale można to zmienić, używając zmiennej środowiskowej `PYTHONBREAKPOINT`. Jeśli wolisz korzystać z innego debugera (na przykład `ipdb`, który opisaliśmy w rozdziale 2., „Nowoczesne środowiska programistyczne Pythona"), możesz przypisać do tej zmiennej wartość informującą Pythona, jaką funkcję ma wywoływać.

Standardową praktyką jest konfigurowanie preferowanego debugera w skrypcie profilu powłoki, dzięki czemu nie trzeba modyfikować zmiennej w każdej sesji powłoki. Na przykład jeśli używasz Basha i zawsze chcesz korzystać z `ipdb` zamiast z `pdb`, możesz wstawić poniższą instrukcję w pliku *.bash_profile*:

```
PYTHONBREAKPOINT=ipdb.set_trace()
```

To podejście dobrze sprawdza się także w zespołach. Na przykład jeśli ktoś chce, abyś pomógł mu w debugowaniu, możesz poprosić, aby wstawił instrukcje `breakpoint()` w miejscach, które mogą powodować problem. Dzięki temu gdy uruchomisz kod na własnym komputerze, użyty zostanie wybrany przez Ciebie debuger.

Jeśli nie wiesz, gdzie umieścić punkt przerwania, ale aplikacja kończy pracę w wyniku wystąpienia nieobsługiwanego wyjątku, możesz wykorzystać mechanizm debugowania poawaryjnego z `pdb`. Poniższe polecenie uruchamia skrypt Pythona w sesji diagnostycznej, w której działanie kodu jest wstrzymywane w momencie zgłoszenia wyjątku:

```
python3 -m pdb -c continue script.py
```

Tryb roboczy

Od wersji 3.7 interpreter Pythona można uruchomić w specjalnym trybie roboczym, w którym w czasie wykonywania kodu stosowane są dodatkowe analizy. Pomaga to w diagnozowaniu ewentualnych problemów, jakie mogą wystąpić w trakcie działania programu. W poprawnie działającym kodzie takie analizy generują niepotrzebne koszty, dlatego domyślnie są nieaktywne.

Tryb roboczy można aktywować na dwa sposoby:

- Za pomocą opcji `-X dev` przy wywołaniu interpretera Pythona w wierszu poleceń, na przykład:

```
python -X dev my_application.py
```

- Za pomocą zmiennej środowiskowej `PYTHONDEVMODE`, na przykład:

```
PYTHONDEVMODE=1 my_application
```

Oto najważniejsze skutki zastosowania tego trybu:

- haczyki związane z przydziałem pamięci, co pozwala wykrywać przepełnienie i niedopełnienie bufora, naruszenia API alokatorów pamięci i przypadki niebezpiecznego użycia blokad **Global Interpreter Lock** (**GIL**);
- ostrzeżenia związane z możliwymi błędami przy imporcie modułów;
- ostrzeżenia dotyczące nieprawidłowego zarządzania zasobami, na przykład niezamykania otwartych plików;
- ostrzeżenia związane z elementami biblioteki standardowej, które są uznawane za przestarzałe i w przyszłości zostaną usunięte;
- włączona obsługa błędów, co powoduje wyświetlenie śladu stosu aplikacji po otrzymaniu przez nią sygnału systemowego `SIGSEGV`, `SIGFPE`, `SIGABRT`, `SIGBUS` lub `SIGILL`.

Ostrzeżenia generowane w trybie roboczym oznaczają, że coś nie działa w oczekiwany sposób. Może to pomóc w wykryciu problemów, które nie przyjmują formy błędów w trakcie standardowej pracy kodu, ale mogą prowadzić do poważnych usterek, gdy aplikacja będzie działać przez długi czas.

Nieprawidłowe zamykanie otwartych plików może w pewnym momencie doprowadzić do wyczerpania się zasobów w środowisku, w którym aplikacja działa. Zasobami są na przykład deskryptory plików, podobnie jak pamięć RAM i przestrzeń dyskowa. Każdy system operacyjny ma ograniczoną liczbę plików, które mogą być jednocześnie otwarte. Jeśli aplikacja otwiera nowe pliki, ale ich nie zamyka, w pewnym momencie nie będzie mogła otworzyć kolejnych.

Tryb roboczy pozwala wcześniej wykryć takie problemy. To dlatego zaleca się korzystanie z niego na etapie testowania aplikacji. Z powodu dodatkowych kosztów analiz aktywnych w trybie roboczym nie zaleca się stosowania go w środowiskach produkcyjnych.

Czasem tryb roboczy można wykorzystać także do diagnozowania istniejących problemów. Przykładem wysoce problematycznej sytuacji jest błąd segmentacji w aplikacji.

Gdy taki błąd występuje w Pythonie, zwykle nie są wyświetlane żadne szczegółowe informacje na jego temat. Zobaczysz tylko krótki komunikat wyświetlony w powłoce:

```
Segmentation fault: 11
```

W momencie wystąpienia błędu segmentacji proces Pythona otrzymuje sygnał systemowy `SIGSEGV` i natychmiast kończy pracę. W niektórych systemach operacyjnych wyświetlany jest zrzut pamięci, czyli zapis stanu pamięci procesu zarejestrowany w momencie awarii. Można go wykorzystać przy debugowaniu aplikacji. Niestety w interpreterze CPython wyświetlany jest zapis pamięci procesu interpretera, dlatego debugowanie odbywa się na poziomie kodu w języku C.

Tryb roboczy dodaje kod do obsługi błędów, który wyświetla ślad stosu Pythona po otrzymaniu sygnału błędu. Dzięki temu otrzymujesz dodatkowe informacje na temat tego, które fragmenty kodu mogą powodować problem. Oto przykładowy kod, o którym wiadomo, że powoduje błąd segmentacji w Pythonie 3.9:

```
import sys

sys.setrecursionlimit(1 << 30)

def crasher():
    return crasher()

crasher()
```

Jeśli wykonasz ten kod w interpreterze Pythona z włączoną opcją `-X dev`, otrzymasz dane wyjściowe podobne do poniższych:

```
Fatal Python error: Segmentation fault

Current thread 0x000000010b04edc0 (most recent call first):
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  File "/Users/user/dev/crashers/crasher.py", line 6 in crasher
  ...
```

Ten mechanizm obsługi błędów można też aktywować poza trybem roboczym. W tym celu użyj opcji `-X faulthandler` w wierszu poleceń lub przypisz do zmiennej środowiskowej `PYTHONFAULTHANDLER` wartość 1.

Niełatwo jest wywołać błędy segmentacji w Pythonie. Często występują one dla rozszerzeń Pythona napisanych w języku C lub C++ oraz dla funkcji wywoływanych z poziomu bibliotek współdzielonych (plików DLL, *.dylib* lub *.so*). Jednak istnieje kilka znanych i dobrze udokumentowanych scenariuszy, w których ten błąd może wystąpić w kodzie w czystym Pythonie. W repozytorium implementacji CPython znajduje się kolekcja tego typu znanych „prowokatorów awarii". Znajdziesz ją na stronie *https://github.com/python/cpython/tree/master/Lib/test/crashers*.

Funkcje __getattr__() i __dir__() na poziomie modułu

W każdej klasie Pythona można zdefiniować niestandardowe funkcje `__getattr__()` i `__dir__()`, aby zmodyfikować dynamiczny dostęp do atrybutów obiektów. Funkcja `__getattr__()` jest wywoływana, gdy nie można znaleźć podanej nazwy atrybutu — pozwala to obsłużyć wyszukiwanie atrybutu i ewentualnie wygenerować jego wartość „w locie". Metoda `__dir__()` jest uruchamiana po przekazaniu obiektu do funkcji `dir()`. Powinna ona zwracać listę nazw atrybutów obiektu.

Od Pythona 3.7 funkcje `__getattr__()` i `__dir__()` można definiować na poziomie modułu. Działają one podobnie jak metody obiektów. Funkcja `__getattr__()` z poziomu modułu jest wywoływana po nieudanym wyszukiwaniu składowej modułu. Funkcja `__dir__()` jest uruchamiana po przekazaniu obiektu modułu do funkcji `dir()`.

Omawiane możliwości mogą być przydatne dla osób odpowiedzialnych za konserwację biblioteki, gdy funkcje lub klasy z modułu zostają uznane za przestarzałe. Załóżmy, że funkcja `get_ci()` z podrozdziału „Wskazówki dotyczące typów w typach generycznych" jest udostępniana w otwartej bibliotece *dict_helpers.py*. Jeśli chcesz zmienić nazwę funkcji na `lookup_ci()`, ale umożliwić importowanie jej po dawną nazwą, możesz zastosować następujący wzorzec wycofywania elementów kodu:

```
from typing import Any
from warnings import warn

def ci_lookup(d: dict[str, Any], key: str) -> Any:
    ...

def __getattr__(name: str):
    if name == "get_ci":
        warn(f"{name} jest przestarzała", DeprecationWarning)
        return ci_lookup

    raise AttributeError(f"Moduł {__name__} nie zawiera atrybutu {name}")
```

Ten wzorzec generuje ostrzeżenie `DeprecationWarning` niezależnie od tego, czy funkcja `get_ci()` została zaimportowana bezpośrednio z modułu (na przykład za pomocą instrukcji `from dict_helpers import get_ci`), czy użyta jako atrybut `dict_helpers.get_ci`.

Ostrzeżenia o przestarzałych elementach nie są domyślnie wyświetlane. Możesz włączyć ich wyświetlanie w trybie roboczym.

Formatowanie łańcuchów znaków za pomocą obiektów f-string

Obiekty f-string, nazywane też **formatowanymi literałami tekstowymi**, są jedną z najbardziej lubianych funkcji Pythona, które pojawiły się w wersji 3.6. Zostały opisane w dokumencie PEP 498 i zapewniają nowy sposób formatowania łańcuchów znaków. W wersjach starszych niż 3.6 dostępne były już dwie różne metody formatowania łańcuchów znaków. Tak więc obecnie łańcuch znaków można sformatować za pomocą trzech technik. Oto one:

- Formatowanie z użyciem symbolu `%`. Jest to najstarsza technika. Używany jest w niej wzorzec podstawiania przypominający składnię funkcji `printf()` z biblioteki standardowej języka C:

```
>>> import math
>>> "Przybliżona wartość liczby π: %f" % math.pi
'Przybliżona wartość liczby π: 3.141593'
```

- Formatowanie za pomocą metody `str.format()`. Ta technika jest wygodniejsza i mniej narażona na błędy niż formatowanie z użyciem symbolu `%`, ale wymaga więcej kodu. Umożliwia stosowanie nazwanych zastępowanych pól, a także wielokrotne wykorzystanie tej samej wartości:

```
>>> import math
>>> "Przybliżona wartość liczby π: {:f}".format(pi=math.pi)
'Przybliżona wartość liczby π: 3.141593'
```

- Formatowanie za pomocą obiektów **f-string**. Jest to najbardziej zwięzła, wszechstronna i wygodna technika formatowania łańcuchów znaków. Automatycznie zastępuje elementy literałów na podstawie wartości zmiennych i wyrażeń z lokalnej przestrzeni nazw:

```
>>> import math
>>> f"Przybliżona wartość liczby π: {math.pi:f}"
'Przybliżona wartość liczby π: 3.141593'
```

Obiekty f-string są oznaczone za pomocą przedrostka `f`, a ich składnia przypomina składnię metody `str.format()`, ponieważ stosuje się w niej podobne techniki do podawania zastępowanych pól w formatowanym tekście. W metodzie `str.format()` wstawiany tekst jest określany za pomocą argumentów pozycyjnych i nazw argumentów. Obiekty f-string są wyjątkowe dlatego, że jako zastępowane pola można podawać dowolne wyrażenia Pythona przetwarzane w czasie wykonywania programu. W takich miejscach można używać każdej zmiennej, która jest dostępna w przestrzeni nazw obejmującej formatowany literał.

Możliwość stosowania wyrażeń jako zastępowanych pól sprawia, że kod definiujący formatowanie jest prostszy i krótszy. Dostępne są też te same specyfikatory formatu zastępowanych pól (określające dopełnienie, wyrównanie, wyświetlanie znaków itd.) co w metodzie `str.format()`. Oto używana składnia:

```
f"{wyrażenie_dla_zastępowanego_pola:specyfikator_formatowania}"
```

Poniżej przedstawiamy prosty przykład wykonania w interaktywnej sesji kodu, który wyświetla pierwszych 10 potęg liczby 10 za pomocą obiektów f-string i wyrównuje wyniki z wykorzystaniem dopełnienia:

```
>>> for x in range(10):
...     print(f"10^{x} == {10**x:10d}")
...
10^0 ==          1
10^1 ==         10
10^2 ==        100
10^3 ==       1000
10^4 ==      10000
10^5 ==     100000
10^6 ==    1000000
10^7 ==   10000000
10^8 ==  100000000
10^9 == 1000000000
```

Kompletna specyfikacja formatowania łańcuchów znaków Pythona tworzy odrębny minijęzyk. Najlepszym źródłem informacji o nim jest oficjalna dokumentacja — *https://docs.python.org/3/library/string.html*. Inną przydatną stroną poświęconą temu zagadnieniu jest *https://pyformat.info/* — znajdziesz tam omówienie najważniejszych elementów tej specyfikacji razem z praktycznymi przykładami.

Podkreślenia w literałach liczbowych

Podkreślenia w literałach liczbowych są bardzo łatwe do wprowadzenia, ale nadal nie stały się tak popularne, jak mogłyby być. Od Pythona 3.6 można używać znaku podkreślenia (_) do oddzielania cyfr w literałach liczbowych. Poprawia to czytelność dużych liczb. Przyjrzyj się następującemu przypisaniu wartości:

```
account_balance = 100000000
```

Przy tak dużej liczbie zer trudno jest szybko stwierdzić, czy chodzi tu o setki milionów, czy o miliardy. Możesz więc użyć znaku podkreślenia do rozdzielania tysięcy, milionów, miliardów itd.:

```
account_balance = 100_000_000
```

Teraz łatwiej jest od razu stwierdzić, że zmienna `account_balance` ma wartość sto milionów; nie wymaga to starannego liczenia zer.

Moduł secrets

Jednym błędów w obszarze zabezpieczeń często popełnianych przez wielu programistów jest zakładanie losowości modułu `random`. Losowość liczb generowanych przez ten moduł wystarcza dla celów statystycznych. W tym module używany jest generator liczb pseudolosowych Mersenne Twister. Zapewnia on znany równomierny rozkład i ma wystarczającą długość okresu, dlatego można go stosować w symulacjach, modelowaniu lub całkowaniu.

Jednak algorytm Mersenne Twister jest w pełni deterministyczny, podobnie jak moduł `random`. To oznacza, że gdy znane są początkowe warunki (**ziarno**), można wygenerować te same liczby pseudolosowe. Ponadto kiedy znana jest wystarczająca liczba kolejnych wartości z generatora pseudolosowego, zwykle można ustalić ziarno i przewidzieć kolejne wyniki. Dotyczy to także algorytmu Mersenne Twister.

> Jeśli chcesz się dowiedzieć, jak przewidzieć liczby generowane przez algorytm Mersenne Twister, zapoznaj się z następującym projektem z serwisu GitHub: *https://github.com/kmyk/mersenne-twister-predictor*.

Ta cecha generatorów liczb pseudolosowych oznacza, że nigdy nie należy ich stosować do generowania losowych wartości na potrzeby zabezpieczeń. Na przykład jeśli chcesz wygenerować losową sekwencję, którą można wykorzystać jako hasło użytkownika lub token, znajdź inne źródło losowości.

Moduł `secrets` służy właśnie do tego. Wykorzystuje on najlepsze źródło losowości w poszczególnych systemach operacyjnych. W Uniksie i systemach uniksowych jest nim urządzenie `/dev/urandom`, a w systemie Windows — generator `CryptGenRandom`.

Oto trzy najważniejsze funkcje z tego modułu:

- `secrets.token_bytes(nbytes=None)` — zwraca `nbytes` losowych bajtów. Ta funkcja jest używana wewnętrznie przez funkcje `secrets.token_hex()` i `secrets.token_urlsafe()`. Jeśli programista nie poda wartości `nbytes`, zwracana jest domyślna liczba bajtów, opisana w dokumentacji jako „rozsądna”.
- `secrets.token_hex(nbytes=None)` — zwraca `nbytes` losowych bajtów w postaci szesnastkowego łańcucha znaków (nie jest to obiekt taki, jaki zwraca funkcja `bytes()`). Ponieważ jeden bajt jest kodowany za pomocą dwóch cyfr szesnastkowych, wynikowy łańcuch znaków zawiera `nbytes` * 2 znaków. Jeśli wartość `nbytes` nie jest podana, zwracana jest ta sama domyślna liczba bajtów co w funkcji `secrets.token_bytes(nbytes=None)`.
- `secrets.token_urlsafe(nbytes=None)` — zwraca `nbytes` losowych bajtów w postaci dostosowanego do adresów URL łańcucha znaków w formacie base64. Ponieważ jeden bajt wymaga mniej więcej 1,3 znaku w kodowaniu base64, wynikowy łańcuch znaków zawiera `nbytes` * 1,3 znaków. Jeśli wartość `nbytes` nie jest podana, zwracana jest ta sama domyślna liczba bajtów co w funkcji `secrets.token_bytes(nbytes=None)`.

Inną ważną, choć często pomijaną funkcją jest `secrets.compare_digest(a, b)`. Porównuje ona dwa łańcuchy znaków lub obiekty bajtowe w sposób, który nie umożliwia napastnikowi oceny na podstawie czasu porównywania, czy wartości przynajmniej częściowo do siebie pasują. Porównywanie dwóch sekretów poprzez standardowe porównywanie łańcuchów znaków (za pomocą operatora `==`) jest podatne na atak przez pomiar czasu. W takim scenariuszu napastnik może sprawdzać wiele sekretów i na podstawie analiz statystycznych stopniowo odgadywać kolejne znaki oryginalnej wartości.

Co może się pojawić w przyszłości?

W czasie gdy powstaje ta książka, Python 3.9 nadal ma tylko kilka miesięcy. Możliwe jednak, że gdy Ty będziesz ją czytać, Python 3.10 będzie prawie gotowy lub nawet już dostępny.

Ponieważ proces rozwoju Pythona jest otwarty i transparentny, mamy wgląd (w dokumentach PEP) w to, co zostało zaakceptowane, a także w to, co zostało już zaimplementowane w wersjach alfa i beta. Dzięki temu możemy omówić funkcje, które pojawią się w Pythonie 3.10. Dalej znajdziesz krótkie omówienie najważniejszych zmian, których można oczekiwać w najbliższej przyszłości.

Tworzenie sumy typów za pomocą operatora |

W Pythonie 3.10 wprowadzone zostanie następne uproszczenie składni wskazówek dotyczących typów. Dzięki nowej składni łatwiej będzie tworzyć adnotacje określające sumę typów.

Python jest językiem z typowaniem dynamicznym i bez obsługi polimorfizmu. Dlatego funkcje mogą przyjmować ten sam argument, który w różnych wywołaniach może być innego typu, i poprawnie go przetwarzać, jeśli używane typy mają ten sam interfejs. Aby lepiej to zrozumieć, wróć do sygnatury funkcji, która umożliwia wyszukiwanie wartości ze słownika z kluczami tekstowymi bez uwzględniania wielkości liter:

```
def get_ci(d: dict[str, Any], key: str) -> Any: ...
```

Wewnętrznie używana jest metoda `upper()` kluczy pobranych ze słownika. Jest to główny powód, dla którego typ argumentu `d` został zdefiniowany jako `dict[str, Any]`, a typ argumentu `key` jako `str`.

Jednak typ `str` nie jest jedynym typem wbudowanym udostępniającym metodę `upper()`. Innym typem z tą metodą jest `bytes`. Jeśli chcesz umożliwić stosowanie w funkcji `get_ci()` słowników z kluczami w postaci łańcuchów znaków i kluczami w postaci bajtów, musisz podać sumę dozwolonych typów.

Obecnie jedynym sposobem na podawanie sumy typów jest użycie wskazówki `typing.Union`. Ta wskazówka umożliwia zapisanie sumy typów `bytes` i `str` jako `typing.Union[bytes, str]`. Kompletna sygnatura funkcji `get_ci()` wyglądałaby więc tak:

```
def get_ci(
    d: dict[Union[str, bytes], Any],
    key: Union[str, bytes]
) -> Any:
    ...
```

Już ten zapis jest długi, a w bardziej skomplikowanych funkcjach sytuacja jest jeszcze gorsza. Dlatego w Pythonie 3.10 możliwe będzie podawanie sumy typów za pomocą operatora |. W przyszłości wystarczy prosty zapis:

```
def get_ci(d: dict[str | bytes, Any], key: str | bytes) -> Any: ...
```

W odróżnieniu od wskazówek dotyczących typów w typach generycznych wprowadzenie operatora sumy typów nie sprawi, że wskazówka `typing.Union` stanie się przestarzała. Dlatego obu konwencji będzie można używać wymiennie.

Strukturalne dopasowywanie wzorców

Strukturalne dopasowywanie wzorców jest najbardziej kontrowersyjną nową cechą Pythona w ostatniej dekadzie — i z pewnością także najbardziej skomplikowaną.

Akceptacja tego mechanizmu była poprzedzona licznymi zaciętymi dyskusjami i niezliczonymi szkicami projektu. Złożoność tego zagadnienia jest dobrze widoczna, jeśli przyjrzeć się wszystkim dokumentom PEP, które go dotyczą. Oto tabela wszystkich dokumentów PEP związanych ze strukturalnym dopasowywaniem wzorców (status z marca 2021 roku).

Data	PEP	Tytuł	Typ	Status
23 czerwca 2020	622	*Structural Pattern Matching*	Standardy	Zastąpiony przez PEP 634
12 września 2020	634	*Structural Pattern Matching: Specification*	Standardy	Zaakceptowany
12 września 2020	635	*Structural Pattern Matching: Motivation and Rationale*	Informacyjny	Wersja finalna
12 września 2020	636	*Structural Pattern Matching: Tutorial*	Informacyjny	Wersja finalna
26 września 2020	642	*Explicit Pattern Syntax for Structural Pattern Matching*	Standardy	Szkic
2 lutego 2021	653	*Precise Semantics for Pattern Matching*	Standardy	Szkic

To sporo dokumentów, a żaden z nich nie jest krótki. Czym więc jest strukturalne dopasowywanie wzorców i do czego może być przydatne?

W strukturalnym dopasowywaniu wzorców wprowadzono **instrukcję dopasowywania** i dwa nowe kontekstowe słowa kluczowe — `match` i `case`. Zgodnie z nazwami można ich używać do dopasowywania (`match`) określonej wartości do listy podanych przypadków (`case`) i wykonywać operacje na podstawie wyniku dopasowania.

> Kontekstowe słowo kluczowe to słowo kluczowe, które nie w każdym kontekście jest zarezerwowane. Poza kontekstem instrukcji dopasowywania słowa `match` i `case` mogą być stosowane jak zwykłe nazwy zmiennych lub funkcji.

Niektórym programistom składnia instrukcji dopasowywania przypomina składnię instrukcji `switch` z języków takich jak C, C++, Pascal, Java i Go. Rzeczywiście, można ją stosować do zaimplementowania tego samego wzorca, ale daje ona zdecydowanie większe możliwości.

Ogólna (i uproszczona) składnia instrukcji dopasowywania wygląda tak:

```
match wyrażenie:
    case wzorzec:
        ...
```

Jako `wyrażenie` można zastosować dowolne poprawne wyrażenie Pythona; `wzorzec` reprezentuje dopasowywany wzorzec i jest nowością w Pythonie. W bloku `case` można umieścić wiele instrukcji. Złożoność instrukcji dopasowywania wynika głównie z wprowadzenia **dopasowywanych wzorców**, które początkowo mogą być trudne do zrozumienia. Wzorce łatwo też pomylić z wyrażeniami, jednak wzorce nie są przetwarzane w taki sam sposób jak zwykłe wyrażenia.

Zanim szczegółowo omówimy dopasowywane wzorce, warto przyjrzeć się przykładowej instrukcji dopasowywania, która działa tak jak instrukcje `switch` z innych języków programowania:

```
import sys

match sys.platform:
    case "windows":
        print("Uruchomiono w systemie Windows")
    case "darwin" :
        print("Uruchomiono w systemie macOS")
    case "linux":
        print("Uruchomiono w systemie Linux")
    case _:
        raise NotImplementedError(
            f"{sys.platform} nie jest obsługiwany!"
        )
```

Jest to oczywiście bardzo prosty przykład, ale już uwidacznia kilka ważnych aspektów. Po pierwsze, jako wzorce mogą być stosowane literały. Po drugie, używany jest specjalny wzorzec wieloznaczny w postaci podkreślenia (`_`). Wzorce wieloznaczne i inne wzorce, które z powodu składni zawsze pasują do szukanych wartości, tworzą **domyślny blok case**. Można go umieścić tylko jako ostatni blok instrukcji dopasowywania.

Ten przykład można oczywiście zapisać za pomocą prostej sekwencji instrukcji `if`, `elif` i `else`. Typowe zadanie na rozmowach rekrutacyjnych na niższe stanowiska polega na napisaniu programu FizzBuzz.

Program FizzBuzz ma odliczać od 0 do dowolnej wartości i w zależności od sprawdzanej liczby wykonywać jedną z czterech operacji:

- wyświetlać `Fizz`, gdy wartość jest podzielna przez 3;
- wyświetlać `Buzz`, gdy wartość jest podzielna przez 5;
- wyświetlać `FizzBuzz`, gdy wartość jest podzielna jednocześnie przez 3 i 5;
- wyświetlać sprawdzaną liczbę we wszystkich pozostałych scenariuszach.

Jest to prosty problem, ale zaskakujące jest, jak wiele osób ma trudności nawet z najłatwiejszymi zadaniami pod presją stresu w trakcie rozmowy rekrutacyjnej. Rozwiązanie można oczywiście opracować za pomocą kilku instrukcji `if`, jednak instrukcja dopasowywania pozwala napisać kod cechujący się naturalną elegancją:

```
for i in range(100):
    match (i % 3, i % 5):
        case (0, 0): print("FizzBuzz")
        case (0, _): print("Fizz")
        case (_, 0): print("Buzz")
        case _: print(i)
```

W tym przykładzie wartość jest w każdej iteracji dopasowywana do warunku `(i % 3, i % 5)`. Potrzebne są obie operacje modulo, ponieważ efekt każdej iteracji zależy od obu wyników. Kod kończy sprawdzanie wzorców po znalezieniu pasującego bloku i wykonuje tylko jeden blok kodu.

Widoczną różnicą w porównaniu z poprzednim przykładem jest to, że teraz zastosowaliśmy wzorce w postaci sekwencji zamiast wzorców w postaci literałów:

- Wzorzec `(0, 0)` pasuje do dwuelementowej sekwencji, w której oba elementy są równe 0.
- Wzorzec `(0, _)` pasuje do dwuelementowej sekwencji, w której pierwszy element jest równy 0. Drugi element może być dowolną wartością dowolnego typu.
- Wzorzec `(_, 0)` pasuje do dwuelementowej sekwencji, w której drugi element jest równy 0. Pierwszy element może być dowolną wartością dowolnego typu.
- Wzorzec wieloznaczny _ pasuje do wszystkich wartości.

W wyrażeniach dopasowywania można stosować nie tylko proste literały i ich sekwencje. Możesz też dopasowywać wartości do określonych klas. I to właśnie wzorce w postaci klas pozwalają tworzyć fantastyczny kod. Jest to zdecydowanie najbardziej skomplikowany aspekt omawianej funkcji.

W czasie gdy powstaje ta książka, Python 3.10 nie został jeszcze udostępniony. Dlatego trudno jest przedstawić typowe i praktyczne zastosowanie dopasowywania wzorców w postaci klas. Zamiast tego przyjrzymy się przykładowi z oficjalnego samouczka. Oto zmodyfikowany przykład z dokumentu PEP 636 obejmujący prostą funkcję `where_is()`, która potrafi dopasować wartość do struktury przekazanej instancji klasy `Point`:

```
class Point:
    x: int
    y: int

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

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Początek układu")
        case Point(x=0, y=y):
```

```
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Inne miejsce")
        case _:
            print("To nie jest punkt")
```

W tym przykładzie dzieje się wiele rzeczy, dlatego warto omówić wszystkie występujące tu wzorce:

- `Point(x=0, y=0)` zostaje dopasowany, jeśli `point` jest instancją klasy `Point`, a atrybuty `x` i `y` mają wartość 0.
- `Point(x=0, y=y)` zostaje dopasowany, jeśli `point` jest instancją klasy `Point`, a jej atrybut `x` ma wartość 0. Atrybut `y` jest zapisywany w zmiennej `y`, którą można wykorzystać w bloku `case`.
- `Point(x=x, y=0)` zostaje dopasowany, jeśli `point` jest instancją klasy `Point`, a jej atrybut `y` ma wartość 0. Atrybut `x` jest zapisywany w zmiennej `x`, którą można wykorzystać w bloku `case`.
- `Point()` zostaje dopasowany, jeśli `point` jest instancją klasy `Point`.
- `_` zostaje dopasowany zawsze.

Widać tu, że dopasowywanie wzorców pozwala sprawdzać atrybuty obiektu. Choć wzorzec `Point(x=0, y=0)` wygląda jak wywołanie konstruktora, Python nie wywołuje konstruktorów w trakcie sprawdzania wzorców. Nie sprawdza też argumentów pozycyjnych ani nazwanych metody `__init__()`, dlatego w dopasowywanym wzorcu można uzyskać dostęp do wartości dowolnych atrybutów.

Przy dopasowywaniu wzorców można też posługiwać się składnią pozycyjną dla atrybutów, jednak wymaga to więcej pracy. Trzeba udostępnić w klasie dodatkowy atrybut `__match_args__`, który określa naturalną kolejność atrybutów instancji klasy. Oto przykład:

```
class Point:
    x: int
    y: int

    __match_args__ = ["x", "y"]

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

def where_is(point):
    match point:
        case Point(0, 0):
            print("Początek układu")
        case Point(0, y):
            print(f"Y={y}")
        case Point(x, 0):
            print(f"X={x}")
```

```
        case Point():
            print("Inne miejsce")
        case _:
            print("To nie jest punkt")
```

A to tylko mała próbka możliwości. Instrukcje dopasowywania są dużo bardziej skomplikowane, niż można to pokazać w tym krótkim podrozdziale. Jeśli mielibyśmy uwzględnić wszystkie zastosowania, warianty składni i przypadki brzegowe, musielibyśmy poświęcić im cały rozdział. Jeżeli chcesz dowiedzieć się więcej na temat instrukcji dopasowywania, koniecznie przeczytaj trzy „kanoniczne" dokumenty PEP: 634, 635 i 636.

Podsumowanie

W tym rozdziale omówiliśmy najważniejsze zmiany w składni języka i bibliotece standardowej wprowadzone w czterech ostatnich wersjach Pythona. Jeśli nie śledzisz aktywnie informacji o wersjach lub nie przeszedłeś jeszcze na Pythona 3.9, powinno to zapewnić Ci wystarczającą wiedzę o bieżącym stanie języka.

W tym rozdziale wprowadziliśmy też koncepcję „idiomów programistycznych". Będziemy do niej wielokrotnie wracać w tej książce. W następnym rozdziale dokładnie przyjrzymy się wielu idiomom z Pythona, porównując wybrane mechanizmy z Pythona i z innych języków programowania. Jeśli jesteś doświadczonym programistą, ale dopiero od niedawna korzystasz z Pythona, będzie to doskonała okazja, aby się dowiedzieć, „jak się to robi w Pythonie". Zobaczysz też, gdzie Python się wyróżnia, a gdzie wciąż pozostaje w tyle za konkurencją.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

Ź

Ż

Koduj wszystko w Pythonie. Obiektywnie, strukturalnie i funkcjonalnie!

Python cechuje się dużą prostotą, a przy tym jest wszechstronny. Ma bardzo szeroki zakres zastosowania, przez co coraz więcej osób podejmuje naukę programowania w tym języku. Python należy do języków najczęściej używanych przez programistów, którzy tworzą w nim gry i aplikacje webowe. Świetnie sprawdza się ponadto w pracy z wykorzystaniem sztucznej inteligencji i uczenia maszynowego. Tym, co programiści doceniają w Pythonie, jest też obiektywność. Ucząc się, przyswajamy bowiem również zasady programowania obiektywnego, a więc koncepcji dla wielu innych języków.

Oto książka, którą docenią i osoby rozpoczynające przygodę z programowaniem, i programiści znający już inne języki. Znajdziesz tu zarówno podstawowe informacje o Pythonie, jak i wskazówki dotyczące pisania rozszerzeń, dzięki którym będziesz w stanie korzystać z atutów kilku języków. Przydatnym uzupełnieniem są liczne przykłady, pokazujące, jak rozwiązywać częste problemy. To już czwarte wydanie tego praktycznego podręcznika — docenianego za to, że pozwala dobrze poznać Pythona i uczy, jak pisać wydajny i czytelny kod.

W książce między innymi:

- jakie są najważniejsze usprawnienia w Pythonie
- jak przeprowadzić izolację środowiska
- jak używać najnowszych funkcji w Pythonie
- czym Python się różni od innych języków
- co to jest współbieżność i wielowątkowość
- na czym polega programowanie sterowane zdarzeniami
- jakie są elementy metaprogramowania
- jak przeprowadzić automatyzację kontroli jakości
- jak optymalizować kod

ISBN 978-83-283-8743-0