Czysty kod w Pythonie

Czysty kod

Dave Thomas, założyciel OTI, ojciec chrzestny strategii Eclipse

„Czysty kod może być czytany i rozszerzany przez innego programistę niż jego autor. Posiada on testy jednostkowe i akceptacyjne. Zawiera znaczące nazwy. Oferuje jedną, a nie wiele ścieżek wykonania jednej operacji. Posiada minimalne zależności, które są jawnie zdefiniowane, jak również zapewnia jasne i minimalne API. Kod powinien być opisywany przy jednoczesnej zależności od języka — nie wszystkie potrzebne informacje mogą być wyrażane bezpośrednio w kodzie."

Grady Booch, autor *Object Oriented Analysis and Design with Applications*

„Czysty kod jest prosty i bezpośredni. Czysty kod czyta się jak dobrze napisaną prozę. Czysty kod nigdy nie zaciemnia zamiarów projektanta; jest pełen trafnych abstrakcji i prostych ścieżek sterowania."

Bjarne Stroustrup, twórca C++ oraz autor *The C++ Programming Language*

„Lubię, gdy mój kod jest elegancki i efektywny. Logika kodu powinna być prosta, aby nie mogły się w niej kryć błędy, zależności minimalne dla uproszczenia utrzymania, obsługa błędów kompletna zgodnie ze zdefiniowaną strategią, a wydajność zbliżona do optymalnej, aby nikogo nie kusiło psucie kodu w celu wprowadzenia niepotrzebnych optymalizacji. Czysty kod wykonuje dobrze jedną operację."

Nazwy

- Używanie nazw przedstawiających intencje

```
int d; // Czas trwania w dniach
int daysSinceCreation; // dniOdUtworzenia
```

- Unikanie dezinformacji

```
int XYZControllerForEfficientHandlingOfStrings
int XYZControllerForEfficientStorageOfStrings
```

```
a = l
if  O == l:
    a = O1
else:
    l = 01
```

- Unikanie odwzorowania mentalnego

  Osoby czytające kod nie powinny mentalnie przekształcać nazw na inne, które znają.

- Nazwy klas i metod Nazwa klasy – rzeczownik
  Nazwa metody - czasownik

- Używanie nazw, które można wymówić i łatwo wyszukać

```
 class DtaRcrd102 {
 private Date genymdhms;
 private Date modymdhms;
 private final String pszqint = "102";
/* … */
};
```

```
class Customer {
 private Date generationTimestamp;
 private Date modificationTimestamp;
 private final String recordId = "102";
/* ... */
 };
```

- **Python**
  - stosuj małe litery, a słowa oddzielaj znakiem podkreślenia

```
populated_countries_list = []
def calculate_tax_data():
   …
```

  - nazwy klas powinny składać się z wielkich i małych liter

```
class UserInformation:
      …
```

  - Nazwy stałych powinny składać się wyłącznie z wielkich liter

```
TOTAL = 100
```

Pythoniczny styl kodowania

- Metoda `join()`

```
first_name = "Jan"
last_name = "Nowak"
full_name = first_name + " " + last_name
" ".join([first_name, last_name])
```

- instrukcje `is` oraz `is not`

```
val = {}
if val:
...
if val is not None:
...
```

- warunek `is not` zamiast `not ... is`

```
if not val is None:
...
if val is not None:

...
```

- Konsekwentnie stosowanie instrukcji `return`

Jeżeli funkcja zwraca wynik, sprawdzaj, czy robi to we wszystkich swoich wątkach.

- zwykła funkcja zamiast funkcji lambda

```
square = lambda x: x * x
def square(val):
      return val * val
```

- metody `startswith()` i `endswith()`

```
data = "Witaj, swiecie"
if data[:5] == "Witaj":
...
if data.startswith("Witaj"):
...
```

- metoda `isinstance()`

```
user_ages = {"Leszek": 35, "Joanna": 89, "Igor": 12}
if type(user_ages) == dict:
...
if isinstance(user_ages, dict):
...
```

- Porównywanie wartości logicznych

```
is_empty = False
 if is_empty == False:
...
 if is_empty is False:
...
```

```
is_empty = False
if not is_empty:
...
```

Komentarze

- Komentarze prawne

  Na przykład informacje o prawach autorskich.

- Komentarze informacyjne

  Podstawowe, przydatne informacje

- Wyjaśniające zamierzenia

  Powody podjęcia danej decyzji

- Komentarze wyjaśniające

  Wyjaśnienie niejasnych argumentów lub zwracanych wartości

- Ostrzegające o konsekwencjach

  Ostrzeganie innych o konsekwencjach

- Komentarze wzmacniające

  wzmocnienia wagi danej operacji

- **Komentarze dokumentacyjne**
  - komentarz dokumentacyjny umieszcza się wewnątrz potrójnych cudzysłowów.

```
def get_prime_number():
 """Lista liczb pierwszych z zakresu od 1 do 100."""
```

  - Komentarze dokumentacyjne do modułów

    Komentarz dokumentacyjny opisujący zastosowanie modułu umieszcza się na początku pliku, przed instrukcjami `import`

  - Komentarze dokumentacyjne do klas

```
class Student:
    """Klasa implementujaca operacje wykonywane przez studenta."""
     def __init__(self):
         pass
```

  - Komentarze dokumentacyjne do funkcji

    Umieszcza się go na początku funkcji, opisuje jej działanie.

Struktury sterujące

- Wyrażenia listowe

```
numbers = [10, 45, 34, 89, 34, 23, 6]
square_numbers = map(lambda num: num**2, numbers)
square_numbers = [num**2 for num in numbers]
```

- Wyrażenia lambda

```
data = [[7], [3], [0], [8], [1], [4]]
def min_val(data):
    return min(data, key=lambda x: len(x))

min_val = min(data, key=lambda x: len(x))
```

- Funkcja `range()`

Przy użyciu funkcji `range()` dane nie są umieszczane w pamięci.

```
for item in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    print(item)
```

```
for item in range(10):
    print(item)
```

- Generatory

Wyrażenie listowe różni się od generatora tym, że zajmuje pamięć.

```
def read_file(file_name):
    fread = open(file_name, "r")
    data = [line for line in fread if line.startswith(">>")]
    return data
print(read_file("log.txt"))
```

```
def read_file(file_name):
    with open(file_name) as fread:
        for line in fread:
            yield line
for line in read_file("log.txt"):
    if line.startswith(">>"):
        print(line)
```

Wyjątki

- Często zgłaszane wyjątki

```
def division(dividend, divisor):
    try:
        return dividend/divisor
    except ZeroDivisionError as zero:
        raise ZeroDivisionError("Dzielnik musi byc rozny od zera")
```

- Obsługa konkretnych wyjątków

```
def get_even_list(num_list):
    return [item for item in num_list if item%2==0]
numbers = None
try:
    get_even_list(numbers)
except:
    print("Cos poszlo nie tak.")
>>> Cos poszlo nie tak.
```

```
def get_even_list(num_list):
    return [item for item in num_list if item%2==0]
numbers = None
 try:
    get_even_list(numbers)
except TypeError:
    print("Lista moze zawierac wylacznie liczby.")
except RuntimeError:
    print("Blad wykonania kodu.")
```

- instrukcja `finally`

  Kod w bloku `finally` jest wykonywany niezależnie od tego, czy został zgłoszony wyjątek, czy nie.
- Własne klasy wyjątków

  Dzięki własnym klasom wyjątków łatwiej będzie diagnozować przyczyny problemów.
- Zewnętrzne wyjątki

  Podczas korzystania z zewnętrznego interfejsu API bardzo ważna jest znajomość wszystkich zgłaszanych przez niego wyjątków.
- Jak najmniejsze bloki `try`

  Informują jakie błędy mogą pojawić się w danej części kodu, a także ułatwiają diagnozowanie kodu.

Struktury danych

- Zbiór
  - Nie może zawierać zduplikować danych.
  - Do poszczególnych elementów nie można odwoływać się za pomocą indeksów.
  - Odwołanie do każdego elementu zajmuje tyle samo czasu.
  - Nie można wykonywać operacji typowych dla listy.
  - Elementy są sortowane w miarę ich dodawania.
- Struktura `namedtuple`
  - Krotka, której pola mają nazwy.
  - Dobrze ją wykorzystać, gdy wykonujemy operacje na kilku wartościach, dzięki opisom kod jest bardziej czytelny
- Typ `str` i znaki diakrytyczne
  - typ `str` reprezentuje ciąg znaków.
  - Liczby przypisane znakom noszą nazwy kodów Unicode.
  - W Pythonie najczęściej stosowanie jest kodowanie UTF-8.
- Listy
  - Stosowanie iteratorów zamiast list w przypadku przetwarzania dużych ilości danych.
  - Funkcja `zip()` za pomocą której, można łatwo i wydajnie łączyć i równolegle przetwarzać dwie listy.

- Słownik
  - Dzięki niemu można szybko odwoływać się do danych.
  - Słownik nie może zawierać powtarzających się kluczy.
  - biblioteki `pprint` i `json` umożliwiają czytelne wyświetlanie zawartości słownika.
  - Słownik jako odpowiednik instrukcji `switch`:

```
def tanzania(amount):
    calculate_tax = <kod wyliczajacy podatek>
    return calculate_tax
def zambia(amount):
    calculate_tax = <kod wyliczajacy podatek>
    return calculate_tax
def eritrea(amount):
    calculate_tax = <kod wyliczajacy podatek>
    return calculate_tax
country_tax_calculate = {
"Tanzania": tanzania,
 "Zambia": zambia,
"Eritrea": eritrea
 }
def calculate_tax(country_name, amount):
    return country_tax_calculate[country_name](amount)
calculate_tax("Zambia", 8000000)
```

Przydatne biblioteki

- collections

  oferuje wiele przydanych struktur danych, m.in. `namedtuple`, `defaultdict` i `OrderedDict`.

- csv

  służy do odczytywania i zapisywania plików w formacie CSV.

- datetime i time

  umożliwia operacje na datach i godzinach.

- math

  zawiera mnóstwo prostych i zaawansowanych funkcji matematycznych.

- re

  wykonywanie operacji na wyrażeniach regularnych.

- tempfile

  tworzenie plików tymczasowych

- functools

  Oferuje mnóstwo opcji umożliwiających pisanie kodu funkcyjnego.

- sys i os

  Służą do wykonywania najróżniejszych operacji systemowych.

- subprocess

  Zawiera funkcje umożliwiające uruchamianie i zarządzanie wieloma procesami w systemie

- logging

  Oferuje system logowania.

- __future__

  pseudomoduł umożliwiający korzystanie z nowych funkcjonalności języka, które nie są kompatybilne z wykorzystywanym interpreterem.

Funkcje

- Funkcje powinny być małe.
  Funkcje powinny mieć nie więcej niż 20 wierszy.
- Funkcja powinna wykonywać tylko jedną czynność.
- Jeden poziom abstrakcji w funkcji.
- Generatory
  Generator nie umieszcza danych w pamięci, warto go zastosować gdy umieszczanie danych w strukturze spowoduje przepełnienie pamięci

```
def read_file(file_name):
    with open(file_name) as fread:
        for line in fread:
            yield line
```

- Wyjątek zamiast wyniku `None`
  Dobry wyjątek pozwoli na szybkie znalezienie przyczyny błędu.

```
def read_lines_for_python(file_name, file_type):
    if not file_name or file_type not in ("txt", "html"):
        return None
lines = []
with open(file_name, "r") as fileread:
    for line in fileread:
        if line.find("Python") != -1:
            return "Słowo Python znalezione"
if not read_lines_for_python("plik_nie_zawierający_słowa.pdf", "pdf,,)
    print("Brak słowa Python w pliku")
```

```
import os.path
def read_lines_for_python(file_name, file_type):
    if file_type not in ("txt", "html"):
        raise ValueError("Niewłaściwy format pliku")
    if not os.path.isfile(file_name):
        raise IOError("Brak pliku")
    with open(file_name, "r") as fileread:
        for line in fileread:
            if line.find("Python") != -1:
                return "Słowo Python znalezione"
    return
if not read_lines_for_python("plik_nie_zawierający_słowa_python.pdf", "pdf"):
    print("Brak słowa Python w pliku")
```

Funkcje

- Klucze i wartości domyślne w argumentach

```
def spam_emails(sender_address, recipient_address, subject,
size, sender_name, recipient_name):
...
spam_emails("adres_nadawcy@gmail.com",
"adres_odbiorcy@yahoo.com", "To jest spam", 10000, "Nazwa
nadawcy", "Nazwa odbiorcy")

spam_emails(sender_address="adres_nadawcy@gmail.com",
            recipient_address="adres_odbiorcy@yahoo.com",
            subject="To jest spam",
            size=10000,
            sender_name="Nazwa nadawcy",
            recipient_name="Nazwa odbiorcy")
```

- Funkcja lambda w wyrażeniach
  - Należy unikać kiedy:

Kod staje się przez nią mniej czytelny.

Niewłaściwe użycie może powodować błędy.

  - Można stosować kiedy:

Trzeba wykonać na tyle prostą operację, że nie warto tworzyć osobnej funkcji.

- Krytyczne podchodzenie do tworzonych funkcji
  - Dziennik (log)

dostępny jest rozbudowany i elastycznie konfigurowany moduł do tworzenia dzienników. Można w nim określać różne poziomy logowania.

  - Testy jednostkowe

Profesjonalnie przygotowany test pozwala wykrywać błędy w kodzie i daje poczucie bezpieczeństwa. Biblioteki używane do pisania testów to `pytest` i `unittest`.

Klasy

- Zasada pojedynczej odpowiedzialności

  Jeżeli klasa wykonuje kilka różnych operacji, należy ją podzielić na mniejsze klasy.

- Dekorator `@property`

  - Opatruje się nim metody odczytujące wartości zmiennych.
  - Ukrycie skomplikowanego kodu za atrybutem.
  - Sprawdzenie poprawności atrybutu.

- Metody statyczne

  Nie mają dostępu do stanu klasy i nie mogą odwoływać się do zmiennych obiektowych.

- Klasy abstrakcyjne

  - Można jej używać do definiowania interfejsów.
  - Zapobiega się używaniu interfejsów bez ich uprzedniego zaimplementowania.

- Atrybuty publiczne zamiast prywatnych

  Przyjęta została konwencja oznaczania zmiennych i metod znakiem podkreślenia, jeśli powinny być traktowane jako prywatne. Nie należy jej jednak nadużywać, ponieważ kod staje się wtedy niezrozumiały.

- Dekorator `@classmethod`

  Metody klasy przydają się w projektach obejmujących wiele klas. Przejrzyście zdefiniowane interfejsy ułatwiają utrzymanie kodu w dłuższej perspektywie.

```
class User:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    @classmethod
    def using_string(cls, names_str):
        first, second = map(str, names_str.split(" "))
        student = cls(first, second)
        return student
    @classmethod
    def using_json(cls, obj_json):
    ...
        return student
    @classmethod
    def using_file_obj(cls, file_obj):
    ...
        return student
data = User.using_string("Jan Nowak")
data = User.using_json(json_obj)
data = User.using_file_obj(file_obj)
```

Struktura Klasy

- zmienne

Zmienne oraz stałe umieszczane na początku klasy, przed konstruktorem i metodami.

- Metoda `__init__()`

Metoda `__init__()` jest konstruktorem klasy.

- Wbudowane metody specjalne

Metody specjalne zmieniają działanie klasy i wprowadzają do niej nowe funkcjonalności.

- Metody klasy

Metody klasy stanowią uzupełnienie konstruktora `__init__()`.

- Metody statyczne

Metody statyczne, w odróżnieniu od zwykłych metod, są związane z definicją klasy, a nie z jej instancją.

- Metody instancji

W metodach instancji implementuje się funkcjonalności klasy.

- Metody prywatne

Metody o nazwach rozpoczynających się od znaku podkreślenia należy traktować jako prywatne.

```
class Employee(Person):
    POSITIONS = ("Kierownik", "Menedzer", "Prezes", "Wlasciciel")
    def __init__(self, name, id, department):
        self.name = name
        self.id = id
        self.department = department
        self.age = None
        self._age_last_calculated = None
        self._recalculated_age()
    def __str__(self):
        return ("Imie i nazwisko: "+self.name+"\nDział: "+ self.department)
    @classmethod
    def no_position_allowed(cls, position):
        return [t for t in cls.POSITIONS if t != position]
    @staticmethod
    def c_positions(position):
        return [t for t in cls.TITLES if t in position]
    @property
    def id_with_name(self):
        return self.id, self.name
    def age(self):
        if (datetime.date.today() > self._age_last_recalculated):
            self.__recalculated_age()
        return self.age
    def _recalculated_age(self):
        today = datetime.date.today()
        age = today.year - self.birthday.year
        if today < datetime.date(
           today.year, self.birthday.month, self.birthday.year):
           age -= 1
        self.age = age
        self._age_last_recalculated = today
```

Moduły i metaklasy

- Moduł

  Moduł jest plikiem z rozszerzeniem .py zawierającym funkcje i klasy. Moduły stosuje się w celu podzielenia funkcjonalności kodu na logiczne części.

- Metaklasa

  Za pomocą metaklasy można modyfikować działanie klasy w zależności od potrzeb.

- Porządkowanie kodu za pomocą modułów
  - Krótkie nazwy modułów.

```
import user_card_payment
```

```
import payment
```

  - Nie umieszcza się w nazwach kropek, wielkich liter i znaków specjalnych.

```
import USERS
```

```
import users
```

  - Sposoby importowania modułów:

```
from user import *
...
cart = add_to_cart(4)
```

```
from user import add_to_cart
...
x = add_to_cart(4)
```

```
import user
...
x = user.add_to_cart(4)
```

- Plik `__init__.py`
  - Zastosowaniem pliku `__init__.py` jest dzielenie modułu na osobne pliki.
  - Dzięki niemu wszystkie funkcjonalności można umieścić w jednym module.

```
from purchase.cart import Cart
from purchase.payment import Payment
```

```
from purchase import Cart, Payment
```

- Importowanie funkcji i klas z modułów
  - Podczas importowania modułów z tego samego pakietu można używać ścieżki względnej lub bezwzględnej.
  - Moduły z innego pakietu można importować na kilka sposobów:

```
from mypackage import *
from mypackage.test import bar
import mypackage
```

- Metaklasa `__all__`

  Za pomocą specjalnej metaklasy `__all__` można zablokować import całej zawartości modułu i udostępniać tylko konkretne symbole.

Moduły i metaklasy

- Kiedy stosować metaklasy?

Najczęściej wykorzystuje się je do tworzenia interfejsów API i bibliotek oraz do implementowania skomplikowanych funkcjonalności.

- Metoda `__new__()`
  - Tworzenie instancji klasy
  - Przypisanie wartości zmiennej
  - Sprawdzanie poprawności danych

- Atrybut `__slots__`

Dzięki atrybutowi `__slots__` oszczędza się pamięć zajmowaną przez obiekty i zwiększa wydajność kodu. Nie należy go stosować:
- Podczas tworzenia klas pochodnych, które dziedziczą cechy klas wbudowanych,
- Gdy atrybutom klasy są przypisywane domyślne wartości.

- Metoda `__call__()`
  - Zabezpieczenie przed bezpośrednim tworzeniem instancji klasy,
  - Tworzenie interfejsu API według określonego wzorca ułatwiającego dostęp do zaimplementowanych funkcjonalności,
  - tworzenie singletonów,
  - zapisywanie wartości w pamięci.

- Deskryptory
  - `__get__(self, instance, owner)`: metoda automatycznie wywoływana w odwołaniu do atrybutu.
  - `__set__(self, instance, value)`: metoda automatycznie wywoływana w chwili przypisania wartości `value` atrybutowi.
  - `__delete__(set, instance)`: metoda automatycznie wywoływana w chwili usunięcia atrybutu.

Dekoratory

- Dekoratory mogą służyć do:
  - ograniczenia częstości wywołań;
  - zapisywania danych w pamięci;
  - mierzenia czasu wykonania funkcji;
  - rejestrowania działania funkcji;
  - rejestrowania i zgłaszania wyjątków;
  - uwierzytelniania i kontroli dostępu.

```
import functools
class ValidateParameters:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
    def __call__(self, *parameters):
       if any([isinstance(item, int) for item in parameters]):
           raise TypeError("Parametrem musi byc ciag znakow!")
       else:
           return self.func(*parameters)
@ValidateParameters
def join_strings(*string_list):
    return "".join(string_list)
print(join_strings("a", "n", "b"))
print(join_strings("a", 1, "c"))
```

```
anb
...
TypeError: Parametrem musi byc ciag znakow!
```

```
def add_suffix(func):
    def wrapper():
        text = func()
        result = " ".join([text, „ma kota"])
        return result
    return wrapper
def to_uppercase(func):
    def wrapper():
        text = func()
        if not isinstance(text, str):
            raise TypeError("Podaj ciag znakow")
        return text.upper()
    return wrapper
@to_uppercase
@add_suffix
def say():
   return „Ala"
print(say())

@add_suffix
@to_uppercase
def say():
   return „Ala"
print(say())
```

```
ALA  ma kota
ALA MA KOTA
```

Menedżery kontekstu

- Operacje na plikach

```
with open("plik.txt") as fread:
    for line in fread:
        print(f"Wiersz: {line}")
```

- Działanie menedżera kontekstu
  - Metoda `__enter__()` zwraca obiekt, który jest przypisywany zmiennej użytej po instrukcji `as`.
  - Metoda `__exit__()` odwołuje się do oryginalnego obiektu menedżera kontekstu.
  - Jeżeli w metodzie `__init__()` lub `__enter__()` zostanie zgłoszony wyjątek, wtedy metoda `__exit__()` nie jest wywoływana.
  - Na początku kodu objętego menedżerem kontekstu wywoływana jest metoda `__enter__()`.
  - Jeżeli metoda __exit__() zwróci wynik `True`, wtedy ewentualne zgłoszone wyjątki zostaną ukryte i wykonywanie bloku kodu objętego menedżerem kontekstu zostanie zakończone w normalny sposób.

- Biblioteka `contextlib`

```
from contextlib import contextmanager
@contextmanager
def write_file(file_name):
    try:
        fread = open(file_name, "w")
        yield fread
    finally:
        fread.close()
with write_file("plik.txt") as f:
    f.write("Zapisywanie danych do pliku ")
    f.write("za pomoca menedzera kontekstu.")
```

- Praktyczne wykorzystanie:
  - Za pomocą menedżera kontekstu można odwoływać się do bazy danych.
  - Tworzenie atrap zgłaszających różnego rodzaju wyjątki.
  - Za pomocą instrukcji `with` można zezwalać tylko jednemu procesowi na dostęp do zasobów.
  - uzyskanie dostępu do zasobów umieszczonych w sieci lub do nawiązanie połączenia z innym komputerem.

Iteratory

- Iterator

jest to obiekt przetwarzający strumień danych. Zawiera metodę `__next__()`, która jest wykorzystywana do wykonywania operacji na kolejnych elementach danych np. za pomocą pętli `for`.

- Obiekt iterowalny

posiada metodę `__iter__()` zwracającą iterator, który wykorzystuje się do przetwarzania danych. Przykładami obiektów iterowalnych są ciągi znaków, listy i słowniki.

```
import csv
sum_data = 0
with open('liczby.csv', 'r') as f:
    reader = csv.reader(f)
    for row in list(reader)[0:]:
        sum_data += sum(map(int, row))
print(sum_data)
```

- Moduł `itertools`

- `itertools.combinations(iterable, r)` – tworzy krotki zawierające kombinacje r elementów zawartych w iterowalnym obiekcie `iterable`.
- `itertools.permutations(iterable, r)` – tworzy krotki zawierające permutacje r elementów zawartych w iterowalnym obiekcie `iterable`.
- `itertools.product(iterable, r)` - zwraca iloczyn kartezjański elementów zawartych w obiekcie `iterable`.
- `itertools.count(start=0, step=1)` - zwraca kolejne liczby, począwszy od `start`, różniące się o wartość `step`.
- `itertools.groupby(iterable, key=None)` - grupuje jednakowe elementy.

- Wyrażenia listowe i iteratory

- iterator generujący liczby parzyste od 0 do 400: `(x*2 for x in range(200))`
- wyrażenie listowe: `[x*2 for x in range(200)]`

Generatory

- Generator:
  - przydaje się do odczytywania dużych ilości danych lub dużej liczby plików.
  - jego działanie można wstrzymywać i wznawiać.
  - zwraca obiekty, które można iterować tak jak listę.
  - w porównaniu z innymi strukturami danych zajmuje znacznie mniej miejsca w pamięci.

```
def generate_numbers(limit):
    for item in range(limit):
        yield item*item
        print(f"Wartosc przetwarzana przez instrukcje yield: {item}")
numbers = generate_numbers(10) print(numbers)
for item in numbers:
    print(item)
```

```
<generator object generate_numbers at 0x021ED840>
0
Wartosc przetwarzana przez instrukcje yield: 0
1
Wartosc przetwarzana przez instrukcje yield: 1
4
Wartosc przetwarzana przez instrukcje yield: 2
...
```

- Instrukcja `yield`

Działa podobnie jak `return`, ale nie powoduje wyjścia z funkcji, tylko wstrzymanie jej wykonywania do następnego wywołania.

- Instrukcja `yield from`

```
def flat_list(iter_values):
    for item in iter_values:
        if hasattr(item, '__iter__'):
            yield from flat_list(item)
        else:
            yield item
print(list(flat_list([1, [2], [3, [4]]])))
```

```
[1, 2, 3, 4]
```

- Instrukcja `yield` jest szybka

Jeżeli trzeba szybko przetwarzać duże ilości danych, wtedy zamiast struktur takich jak listy lub krotki, należy stosować generatory. Instrukcja yield jest znacznie szybsza niż odwołanie do listy.

Programowanie asynchroniczne

- Funkcja `asyncio.run()`
  - Wywołuje się ją w programie tylko raz.
  - Uruchamia podaną w argumencie koprocedurę.
  - Zarządza pętlą zdarzeń.

- Instrukcja `await`

wstrzymuje wykonywanie kodu koprocedury i przekazuje sterowanie z powrotem do pętli zdarzeń.

```
import asyncio
async def hello(first_print, second_print):
    print(first_print) await asyncio.sleep(1)
    print(second_print)
asyncio.run(hello("Dzien dobry", "Do widzenia"))
```

- Koprocedura

Funkcja zdefiniowana za pomocą instrukcji `async def`.

- Obiekt koprocedury

Wynik zwracany przez koprocedurę jest nazywany obiektem koprocedury.

- Obiekt oczekiwalny

obiekt, w którym użyta jest instrukcja `await`.

- Zadanie

Koprocedura uruchamiana za pomocą funkcji `asyncio.create_task()`.

- Futura

obiekt oczekiwalny reprezentujący przyszły wynik operacji asynchronicznej.

- Czasomierze

Służą do określenia maksymalnego czaus oczekiwania na zakończenie zadania. Do tego celu służy funkcja `asyncio.wait_for(aws, timeout, *)`.

Programowanie asynchroniczne

- Aby wstrzymać wykonywanie kodu asynchronicznego należy użyć funkcji `await asyncio.sleep()`, która nie przekazuje sterowania z powrotem do pętli zdarzeń.
- Użycie funkcji blokujących znacząco spowolni wykonywanie się kodu.
- Przy zastosowaniu kodu synchronicznego i asynchronicznego w tym samym programie nieuniknione jest powielanie kodu.
- Podczas pisania kodu asynchronicznego należy liczyć się z utratą kontroli nad jego wykonaniem.
- Diagnozowanie kodu asynchronicznego jest trudniejsze niż synchronicznego.
- Testowanie kodu asynchronicznego jest uciążliwe.
- Użycie instrukcji `async` w kodzie asynchronicznym jest błędem składniowym.

Typy danych

Python jest językiem dynamicznym - w kodzie nie trzeba deklarować typów danych.

```
def is_key_present(data: dict, key: str) -> bool:
    if key in data:
        return True
    else:
        return False
```

Czy typy danych spowalniają kod?

Interpreter języka Python nie sprawdza typów przed uruchomieniem kodu, więc deklaracje nie wpływają na jego wydajność.

Dzięki modułowi `typing` można sprawdzać zgodność typów danych i wykrywać ewentualne błędy przed uruchomieniem kodu w środowisku produkcyjnym. Służą do tego celu specjalne programy, takie jak `mypy`.

- Moduł `typing`

Dostępne są w nim zasadnicze typy danych, takie jak `Any`, `Union`, `Tuple`, `Callable`, `TypeVar`, `Generic` i wiele innych.

- Typ `Union`

Jeżeli nie wiadomo zawczasu dokładnie, jaki typ będą miały dane, które zostaną umieszczone w argumentach funkcji, ale będzie to jeden z podstawowych typów.

- Typ `Any`

Obiekt tego typu zawiera wszystkie właściwości i metody. Stosuje się go wtedy, gdy nie wiadomo, jakiego typu argumenty ma funkcja.

- Typ `Tuple`

Ten typ reprezentuje krotkę.

- Typy `TypeVar` i `Generics`

Przydatny w sytuacji gdy trzeba zdefiniować własny typ danych lub zmienić nazwę istniejącego.

- Typ `Optional`

Stosuje się po to, aby w argumencie funkcji oprócz zwykłych wartości można było umieszczać wartość `None`.

Inne funkcjonalności

- Metoda `super()`

Za jej pomocą można odwoływać się do elementów klasy nadrzędnej.

```
class PaidStudent(Student):
    def __int__(self):
        super().__init__(self)
```

- Biblioteka `pathlib`

`pathlib` jest modułem ułatwiającym m.in. odczytywanie plików, łączenie ścieżek i wyświetlanie drzewa katalogów.

- f-ciągi

Ulepszony typ ciągu znaków, dzięki niemu kod jest bardziej czytelny.

```
user_id = "Jan"
amount = 50
print(f"{user_id} zaplacil {amount} zl")
```

```
Jan zaplacil 50 zl
```

- Obowiązkowe argumenty pozycyjne

za pomocą znaku * można definiować obowiązkowe argumenty pozycyjne.

```
def create_report(user, *, file_type, location):
 ...
create_report("skpl", file_type="txt", location="/user/skpl")
```

- Kontrolowana kolejność elementów w słownikach

W celu kontroli kolejności elementów w słowniku nie jest konieczne stosowanie klasy `OrderDict()`.

```
population_raking = {}
population_raking["Chiny"] = 1
population_raking["Indie"] = 2
population_raking["USA"] = 3
print(f"{population_raking}")
```

```
{'Chiny': 1, 'Indie': 2, 'USA': 3}
```

- Iteracyjne rozpakowywanie struktur

W nowej wersji języka można iteracyjnie rozpakowywać struktury danych.

```
*a, = [1]                          # a = [1]
(a, b), *c = 'PC', 5, 6            # a = "P", b = "C", c = [5, 6]
*a, = range(10)
```

Diagnostyka kodu

- Program `pdb`

Jest jednym z narzędzi do diagnozowania kodu w wierszu poleceń. Dostarcza informacji o stosie wywołań, wartościach parametrów funkcji oraz wykonywanych instrukcjach. Aby przygotować kod do diagnostyki, należy umieścić w nim poniższe wiersze:

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

Instrukcja diagnostyczna powoduje wstrzymanie wykonywania kodu.

- Program `ipdb`

podobnie jak `pdb`, działa w wierszu poleceń i oferuje te same funkcjonalności co `pdb`. Jego dodatkową zaletą jest możliwość pracy w środowisku IPython.

- Program `pudb`

to bardzo rozbudowane narzędzie oferujące więcej funkcjonalności. Jest to wizualne środowisko diagnostyczne działające w trybie tekstowym. Diagnostyka nie odbywa się w wierszu poleceń, tylko w okienkowym interfejsie zawierającym kod.

- Funkcja `breakpoint()`

Jej działanie jest podobne do opisanych wcześniej narzędzi przeznaczonych dla wiersza poleceń.

- Moduł logging

Służy do rejestrowania działania kodu.

```
import logging
logging.getLogger(__name__).addHandler(logging.NullHandler())
```

- Identyfikowanie słabych punktów kodu za pomocą metryk

metryki to liczba błędów w określonej części kodu, czasy odpowiedzi zewnętrznego interfejsu API lub liczba użytkowników zalogowanych do aplikacji WWW.

Metryki wydajnościowe:

- Przpływność
- Błędy
- Wydajność

Metryki wykorzystania zasobów:

- Zajętość
- Dostępność

Testy kodu

- Aby mieć pewność, że tworzony kod jest poprawny, trzeba go zacząć testować jak najwcześniej. Bez testów trudno jest stwierdzić, że kod działa zgodnie z oczekiwaniami.
- Dzięki testom można szybko wykrywać zmiany zakłócające działanie kodu.
- Testy dokumentują kod, dzięki temu nie trzeba specjalnie opisywać każdej części kodu.
- Testy są bardzo cenne dla programistów dołączających do projektu. Nowy programista, czytając i uruchamiając testy, może zapoznać się z działaniem kodu.

Testy kodu

- Biblioteki `pytest` i `unittest`
  - `pytest` jest zewnętrzną, a `unittest` wbudowaną biblioteką.
  - Biblioteka `pytest` umożliwia pisanie testów za pomocą klas lub funkcji.
  - biblioteka `unittest` jest elastyczniej konfigurowalna i oferuje więcej metod testujących.

- Testowanie oparte na właściwościach

polega na przeprowadzaniu testów funkcji z różnymi danymi wejściowymi. W języku Python dostępna jest biblioteka `hypothesis` przeznaczona do tego celu.

- Program `virtualenv`

Za jego pomocą można tworzyć odizolowane środowiska programistyczne.

- Tworzenie raportów z testów

Tworzone raporty zawierają wyniki testów i pozwalają kontrolować pokrycie kodu testami. Raport powinien zawierać szczegółowe informacje o udanych i nieudanych testach.

- Automatyzacja testów jednostkowych

Testy jednostkowe wykonane po scaleniu zmodyfikowanego fragmentu kodu z jego główną częścią dają pewność, że wprowadzone zmiany nie zakłócają działania istniejących funkcjonalności. Do tego celu potrzebny jest system kontroli wersji.

- Testy integracyjne

mają na celu zweryfikowanie, czy poszczególne części kodu współpracują ze sobą zgodnie z oczekiwaniami.

- Analizatory i spójność kodu

Analizatory kodu służą do wykrywania w nim potencjalnych błędów, m.in.:

- błędów składniowych;
- błędów strukturalnych, np. niezdefiniowanych zmiennych;
- odstępstw od przyjętych zasad kodowania.

Bibliografia

- Czysty kod. Podręcznik dobrego programisty

  Robert C. Martin

- Czysty kod w Pythonie

  Sunil Kapil