Kurs z zakresu Python PCAP i PCPP

Dr inż. Marcin Caryk

1 Kopiowanie płytkie i głębokie

Shallow and deep copy

Shallow and deep copy - wstęp

- Rozważmy prosty przykład przypisania listy do zmiennej

  a_list = [1, 'New York', 100]

- Najpierw w pamięci komputera tworzony jest obiekt (w tym przykładzie lista). Teraz przedmiot ma swoją tożsamość,
- następnie obiekt jest wypełniany innymi obiektami,
- w końcu powstaje zmienna, którą należy traktować jako etykietę lub powiązanie nazwy, a etykieta ta odnosi się do określonego miejsca w pamięci komputera.
- Instrukcje przypisania w Pythonie nie tworzą kopii obiektów, a jedynie wiążą nazwy z obiektem. W przypadku niezmiennych obiektów zwykle nie ma to znaczenia.
- Jednak podczas pracy ze zmiennymi obiektami lub kolekcjami zmiennymi obiektami możesz szukać sposobu na tworzenie „prawdziwych kopii” lub „klonów” tych obiektów.
- Płytka kopia oznacza skonstruowanie nowego obiektu kolekcji, a następnie zapełnienie go odniesieniami do obiektów potomnych znalezionych w oryginale. W istocie płytka kopia ma głębokość tylko jednego poziomu. Proces kopiowania nie powtarza się i dlatego nie tworzy kopii samych obiektów potomnych.
- Głęboka kopia sprawia, że proces kopiowania jest rekurencyjny. Oznacza to najpierw skonstruowanie nowego obiektu kolekcji, a następnie rekurencyjne zapełnienie go kopiami obiektów potomnych znalezionych w oryginale. Kopiowanie obiektu w ten sposób przechodzi przez całe drzewo obiektów, aby stworzyć w pełni niezależny klon oryginalnego obiektu i wszystkich jego dzieci.

Shallow and deep copy – id()

- Wbudowana funkcja id() zwraca „tożsamość” obiektu.
- Jest to liczba całkowita, która gwarantuje, że będzie niepowtarzalna i stała dla tego obiektu podczas jego życia.
- Dwa obiekty o nienakładających się okresach życia mogą mieć tę samą wartość id().
- Szczegóły implementacji CPython: To jest adres obiektu w pamięci.
- Id() funkcja jest rzadko używana w aplikacjach.
- Częściej będziesz go używać do debugowania kodu lub eksperymentowania podczas kopiowania obiektów.
- Efektem ubocznym tego rzadkiego użycia jest to, że niektórzy programiści zapominają o jego istnieniu i tworzą własne zmienne zatytułowane id do przechowywania jakiejś tożsamości lub identyfikatora.
- W rezultacie zmienna o nazwie id przesłania prawdziwą funkcję i czyni ją nieosiągalną w zakresie, w jakim zmienna została zdefiniowana
- Gdy masz dwie zmienne odnoszące się do tego samego obiektu, wartości zwracane przez funkcję id() muszą być takie same.
- W drugim przykładzie nie utworzyliśmy nowej listy, tylko stworzyliśmy nową etykietę, która odwołuje się do już utworzonej listy.

```
a_string = '10 days to departure'
b_string = '20 days to departure'

print('a_string identity:', id(a_string))
print('b_string identity:', id(b_string))
```

```
a_string = '10 days to departure'
b_string = a_string

print('a_string identity:', id(a_string))
print('b_string identity:', id(b_string))
```

Shallow and deep copy – „==” i „is”

- Aby porównać dwa obiekty, należy jak zwykle zacząć od operatora „==”. Ten operator porównuje wartości obu operandów i sprawdza równość wartości. Mamy więc tutaj do czynienia z porównaniem wartości.
- W rzeczywistości można by porównać dwa różne obiekty posiadające te same wartości, a wynikiem byłaby „Prawda”. Co więcej, gdy porównasz dwie zmienne odnoszące się do tego samego obiektu, wynikiem będzie również „Prawda”.
- Aby sprawdzić, czy oba operandy odnoszą się do tego samego obiektu, należy użyć operatora „is”. Innymi słowy, odpowiada na pytanie: „Czy obie zmienne odnoszą się do tej samej tożsamości?”

```
a_string = ['10', 'days', 'to', 'departure']
b_string = a_string

print('a_string identity:', id(a_string))
print('b_string identity:', id(b_string))
print('The result of the value comparison:', a_string == b_string)
print('The result of the identity comparison:', a_string is b_string)

print()

a_string = ['10', 'days', 'to', 'departure']
b_string = ['10', 'days', 'to', 'departure']

print('a_string identity:', id(a_string))
print('b_string identity:', id(b_string))
print('The result of the value comparison:', a_string == b_string)
print('The result of the identity comparison:', a_string is b_string)
```

Shallow and deep copy – płytka kopia

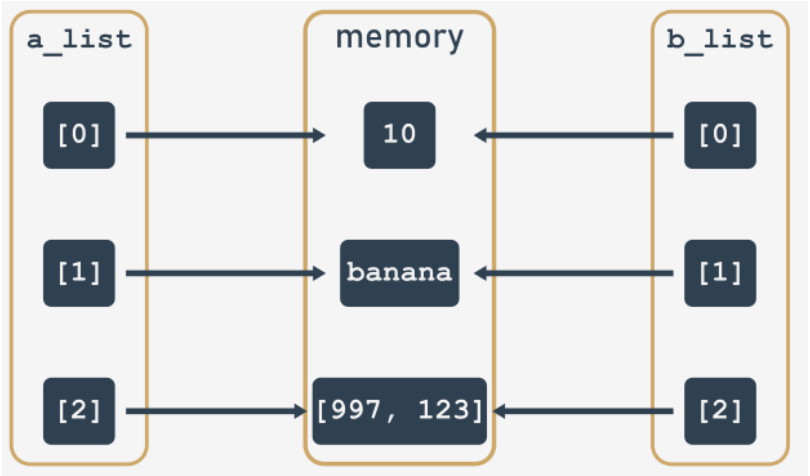
- Podczas przetwarzania danych dojdziesz do punktu, w którym możesz chcieć mieć odrębne kopie obiektów, które możesz modyfikować bez jednoczesnej automatycznej modyfikacji oryginału.
- Spoglądając na kod widzimy że jego intencją jest:
  - zrobić prawdziwą, niezależną kopię a_list (nie tylko odniesienie do kopii).
  - Używając [:], które jest składnią wycinków tablicy, otrzymujemy świeżą kopię obiektu a_list;
  - zmodyfikować oryginalny obiekt;
  - zobaczyć zawartość obu obiektów.
- Należy zwrócić uwagę na kod przedstawiony w prawym okienku, którego a_list jest obiektem złożonym

```
print("Part 1")
print("Let's make a copy")
a_list = [10, "banana", [997, 123]]
b_list = a_list[:]
print("a_list contents:", a_list)
print("b_list contents:", b_list)
print("Is it the same object?", a_list is b_list)

print()
print("Part 2")
print("Let's modify b_list[2]")
b_list[2][0] = 112
print("a_list contents:", a_list)
print("b_list contents:", b_list)
print("Is it the same object?", a_list is b_list)
```

Shallow and deep copy – płytka kopia

Wyjaśnienie zachowania przedstawionego na poprzedniej stronie jest następujące:

- obiekt „a_list” jest obiektem złożonym;
- uruchomiliśmy płytką kopię, która konstruuje nowy obiekt złożony, w naszym przykładzie b_list, a następnie zapełniliśmy go odniesieniami do obiektów znalezionych w oryginale;
- jak widać, płytka kopia ma głębokość tylko jednego poziomu. Proces kopiowania nie powtarza się i dlatego nie tworzy kopii obiektów potomnych, ale zamiast tego zapełnia b_list odniesieniami do już istniejących obiektów.



Shallow and deep copy – płytka kopia

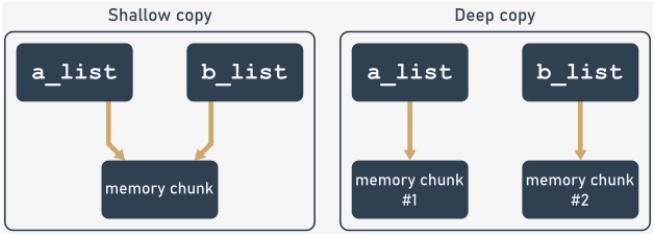
- W poniższym przykładzie utworzymy nową zagnieżdżoną listę, a następnie powierzchownie ją skopiujemy za pomocą funkcji fabrycznej list()
- Oznacza to, że ys będzie teraz nowym i niezależnym obiektem o tej samej zawartości co xs. Możesz to sprawdzić, sprawdzając oba obiekty
- Aby potwierdzić, że ys naprawdę jest niezależne od oryginału, przeprowadźmy mały eksperyment. Możesz spróbować dodać nową podlistę do oryginału (xs), a następnie sprawdzić, czy ta modyfikacja nie wpłynęła na kopię (ys)
- Jak widać, przyniosło to oczekiwany skutek. Modyfikowanie skopiowanej listy na poziomie „powierzchownym” nie stanowiło żadnego problemu.
- Ponieważ jednak utworzyliśmy tylko płytką kopię oryginalnej listy, ys nadal zawiera odniesienia do oryginalnych obiektów potomnych przechowywanych w xs.
- Te dzieci nie zostały skopiowane. Zostały one jedynie ponownie wymienione w skopiowanej liście.
- Dlatego gdy zmodyfikujesz jeden z obiektów podrzędnych w xs, ta modyfikacja zostanie odzwierciedlona również w ys — to dlatego, że obie listy współdzielą te same obiekty podrzędne. Kopia jest tylko płytką, jednopoziomową głęboką kopią

```
xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ys = list(xs)

print(xs)
print(ys)

xs.append(['new list'])

print(xs)
print(ys)

xs[1][0] = 'X'

print(xs)
print(ys)
```

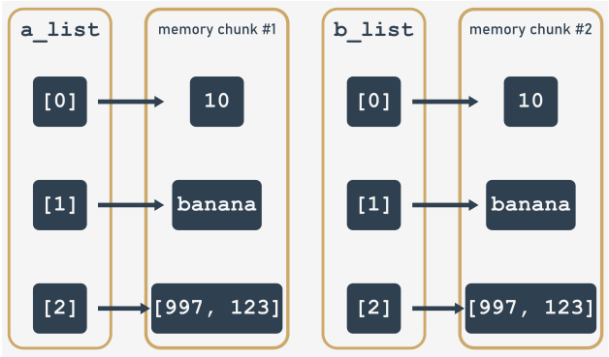
Shallow and deep copy – głęboka kopia

Jeśli chcesz wykonać niezależną kopię obiektu złożonego (listy, słownika, niestandardowej instancji klasy) powinieneś skorzystać z głębokiej kopii, która:

- konstruuje nowy obiekt złożony, a następnie rekurencyjnie wstawia do niego kopie obiektów znalezionych w oryginale;
- zajmuje więcej czasu, ponieważ jest o wiele więcej operacji do wykonania;
- jest zaimplementowana przez funkcję deepcopy(), dostarczaną przez moduł „copy” Pythona



- Kod tworzący niezależną kopię obiektu a_list powinien wyglądać tak, jak kod przedstawiony obok.




```
import copy

print("Let's make a deep copy")
a_list = [10, "banana", [997, 123]]
b_list = copy.deepcopy(a_list)
print("a_list contents:", a_list)
print("b_list contents:", b_list)
print("Is it the same object?", a_list is b_list)

print()
print("Let's modify b_list[2]")
b_list[2][0] = 112
print("a_list contents:", a_list)
print("b_list contents:", b_list)
print("Is it the same object?", a_list is b_list)
```

Shallow and deep copy – głęboka kopia

- Tym razem utworzymy głęboką kopię za pomocą funkcji deepcopy() zdefiniowanej w module kopiowania,
- Kiedy sprawdzisz xs i jego klony zs, które stworzyliśmy za pomocą copy.deepcopy(), zobaczysz, że oba znów wyglądają identycznie — tak jak w poprzednim przykładzie
- Jeśli jednak dokonasz modyfikacji jednego z obiektów podrzędnych w oryginalnym obiekcie (xs), zobaczysz, że ta modyfikacja nie wpłynie na głęboką kopię (zs).
- Oba obiekty, oryginał i kopia, są tym razem w pełni niezależne. xs został sklonowany rekurencyjnie, w tym wszystkie jego obiekty potomne.
- Płytkie kopie za pomocą funkcji w module kopiowania. Funkcja copy.copy() tworzy płytkie kopie obiektów.
- Jest to przydatne, jeśli chcesz wyraźnie zakomunikować, że gdzieś w kodzie tworzysz płytką kopię. Jednak w przypadku wbudowanych kolekcji za bardziej Pythoniczne uważa się po prostu użycie funkcji list, dict i set factory do tworzenia płytkich kopii.

```
import copy
xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
zs = copy.deepcopy(xs)

print(xs)
print(zs)

xs[1][0] = 'X'

print(xs)
print(zs)
```

Shallow and deep copy – wydajność deepcopy()

- W poniższym przykładzie porównamy wydajność trzech sposobów kopiowania dużego obiektu złożonego (milion trzyelementowych krotek).
- Pierwsze podejście to prosta kopia referencyjna. Robi się to bardzo szybko, ponieważ procesor nie ma prawie nic do zrobienia – wystarczy kopia odniesienia do „a_list”.
- Drugie podejście to płytka kopia. Jest to wolniejsze niż poprzedni kod, ponieważ utworzono 1 000 000 odniesień (nie obiektów).
- Trzecie podejście to głęboka kopia. Jest to najbardziej kompleksowa operacja, ponieważ tworzonych jest 3 000 000 obiektów.

```
import copy
import time

a_list = [(1,2,3) for x in range(1_000_000)]

print('Single reference copy')
time_start = time.time()
b_list = a_list
print('Execution time:', round(time.time() - time_start, 3))
print('Memory chunks:', id(a_list), id(b_list))
print('Same memory chunk?', a_list is b_list)

print()

print('Shallow copy')
time_start = time.time()
b_list = a_list[:]
print('Execution time:', round(time.time() - time_start, 3))
print('Memory chunks:', id(a_list), id(b_list))
print('Same memory chunk?', a_list is b_list)

print()

print('Deep copy')
time_start = time.time()
b_list = copy.deepcopy(a_list)
print('Execution time:', round(time.time() - time_start, 3))
print('Memory chunks:', id(a_list), id(b_list))
print('Same memory chunk?', a_list is b_list)
```

Shallow and deep copy – użycie deepcopy()

- Tej samej metody deepcopy() można użyć, gdy chcesz skopiować słowniki lub niestandardowe obiekty klas.
- Zwróć uwagę, że metoda __init__() jest wykonywana tylko raz, mimo że posiadamy dwie instancje przykładowej klasy.
- Ta metoda nie jest wykonywana dla obiektu b_example, ponieważ funkcja deepcopy kopiuje już zainicjowany obiekt.

```
import copy

a_dict = {
    'first name': 'James',
    'last name': 'Bond',
    'movies': ['Goldfinger (1964)', 'You Only Live Twice']
    }
b_dict = copy.deepcopy(a_dict)
print('Memory chunks:', id(a_dict), id(b_dict))
print('Same memory chunk?', a_dict is b_dict)
print("Let's modify the movies list")
a_dict['movies'].append('Diamonds Are Forever (1971)')
print('a_dict movies:', a_dict['movies'])
print('b_dict movies:', b_dict['movies'])
```

```
import copy

class Example:
    def __init__(self):
        self.properties = ["112", "997"]
        print("Hello from __init__()")

a_example = Example()
b_example = copy.deepcopy(a_example)
print("Memory chunks:", id(a_example), id(b_example))
print("Same memory chunk?", a_example is b_example)
print()
print("Let's modify the movies list")
b_example.properties.append("911")
print('a_example.properties:', a_example.properties)
print('b_example.properties:', b_example.properties)
```

Zadanie 6a

- Twoim zadaniem jest napisanie kodu, który przygotuje propozycję obniżek cen na cukierki, których łączna waga przekracza 300 jednostek wagowych (nieważne czy to kilogramy czy funty)
- Twoje dane wejściowe to lista słowników; każdy słownik reprezentuje jeden rodzaj cukierków.
- Każdy rodzaj cukierka zawiera klucz zatytułowany „waga”, który powinien naprowadzić cię na całkowitą wagę danego przysmaku. Dane wejściowe są prezentowane w edytorze;
- Przygotuj kopię listy źródłowej (należy to zrobić za pomocą jednej linijki), a następnie powtórz ją, aby obniżyć cenę każdego przysmaku o 20%, jeśli jego waga przekroczy wartość 300;
- Przedstaw oryginalną listę cukierków oraz listę zawierającą propozycje;
- Sprawdź, czy Twój kod działa poprawnie podczas kopiowania i modyfikowania szczegółów słodyczy.

Zadanie 6b

- Teraz przeróbmy trochę kod:
- wprowadź klasę Przysmak, aby reprezentować ogólny przysmak. Obiekty tej klasy zastąpią słowniki starej szkoły.
- Sugerowane nazwy atrybutów: nazwa, cena, waga;twoja klasa powinna zaimplementować metodę __str__() do reprezentowania każdego stanu obiektu;
- poeksperymentuj z metodami copy.copy() i deepcopy.copy() , aby zobaczyć różnicę w sposobie kopiowania obiektów przez każdą metodę.

3

List comprehensions

List comprehensions

List comprehensions – wstęp

- Istnieje kilka różnych sposobów tworzenia list w Pythonie.
- Aby lepiej zrozumieć kompromisy związane z używaniem rozumienia list w Pythonie, najpierw zobaczmy, jak tworzyć listy za pomocą tych podejść.
- Najpopularniejszym typem pętli jest pętla for. Możesz użyć pętli for, aby utworzyć listę elementów w trzech krokach:
  - Utwórz instancję pustej listy.
  - Zapętlaj iterowalny lub zakres elementów.
  - Dołącz każdy element na końcu listy.
  - Jeśli chcesz utworzyć listę zawierającą pierwsze dziesięć idealnych kwadratów, możesz wykonać te kroki w trzech liniach kodu:

```
# for
squares = []
for i in range(10):
    squares.append(i * i)

print(squares)

# map
txns = [1.09, 23.56, 57.84, 4.56, 6.78]
TAX_RATE = .08
def get_price_with_tax(txn):
    return txn * (1 + TAX_RATE)

final_prices = map(get_price_with_tax, txns)
print(list(final_prices))

# list
squares = [i * i for i in range(10)]
print(squares)
```

List comprehensions – wstęp

- Wywołanie list comprehention ma następującą strukturę

  new_list = [expression for member in iterable]

- Każde rozumienie listy w Pythonie zawiera trzy elementy:

  1. wyrażenie to sam element członkowski, wywołanie metody lub dowolne inne prawidłowe wyrażenie, które zwraca wartość. W powyższym przykładzie wyrażenie i * i jest kwadratem wartości elementu.
  2. Member jest obiektem lub wartością na liście lub iterowalną. W powyższym przykładzie wartością składową jest i.
  3. iterowalna to lista, zestaw, sekwencja, generator lub dowolny inny obiekt, który może zwracać swoje elementy pojedynczo. W powyższym przykładzie iterowalny to range(10)

- Ponieważ wymagania dotyczące wyrażeń są tak elastyczne, rozumienie list w Pythonie dobrze sprawdza się w wielu miejscach, w których można by użyć map().

```
squares = [i * i for i in range(10)]
print(squares)
```

List comprehensions – zalety używania list comprehension

- Wyrażenia listowe są często opisywane jako bardziej Pythoniczne niż pętle czy map().
- Jedną z głównych zalet używania rozumienia list w Pythonie jest to, że jest to pojedyncze narzędzie, którego można używać w wielu różnych sytuacjach. Oprócz standardowego tworzenia list, wyrażenia listowe mogą być również używane do mapowania i filtrowania.
- To jest główny powód, dla którego listy składane są uważane za Pythona, ponieważ Python obejmuje proste, potężne narzędzia, których można używać w wielu różnych sytuacjach. Dodatkową korzyścią uboczną jest to, że za każdym razem, gdy używasz rozumienia listy w Pythonie, nie będziesz musiał zapamiętywać właściwej kolejności argumentów, tak jak w przypadku wywołania funkcji map().
- Listy składane są również bardziej deklaratywne niż pętle, co oznacza, że są łatwiejsze do odczytania i zrozumienia. Pętle wymagają skupienia się na sposobie tworzenia listy. Musisz ręcznie utworzyć pustą listę, zapętlić elementy i dodać każdy z nich na koniec listy.
- Bardziej kompletny opis formuły rozumienia dodaje obsługę opcjonalnych warunków warunkowych. Najczęstszym sposobem dodania logiki warunkowej do listy złożonej jest dodanie warunku na końcu wyrażenia

  new_list = [expression for member in iterable (if conditional)]
- Warunkowe są ważne, ponieważ pozwalają listom złożonym odfiltrować niechciane wartości, co normalnie wymagałoby wywołania funkcji filter()

```
sentence = 'always look on the right side of life'
vowels = [i for i in sentence if i in 'aeiou']
print(vowels)
```

List comprehensions – zalety używania list comprehension

- Warunek można zastosować również w postaci funkcji
- tworzymy złożony filtr is_consonant() i przekazujesz tę funkcję jako instrukcję warunkową do rozumienia listy

```
def is_consonant(sentence):
    vowels = 'aeiou'
    return sentence.isalpha() and sentence.lower() not in vowels

consonants = [i for i in sentence if is_consonant(i)]

print(consonants)
```

- Można też umieścić tryb warunkowy na początku wyrażenia

  new_list = [expression (if conditional) for member in iterable]

- Wykorzystany został tu warunek jeśli i > 0, w przeciwnym razie 0. Warunek określa, aby wypisać wartość i, jeśli liczba jest dodatnia, ale zmienić i na 0, jeśli liczba jest ujemna.

```
original_prices = [1.25, -9.45, 10.22, 3.78, -5.92, 1.16]
prices = [i if i > 0 else 0 for i in original_prices]
print(prices)
```

- Tworząc funkcję get_price można zamknąć w niej warunek

```
def get_price(price):
    return price if price > 0 else 0
prices = [get_price(i) for i in original_prices]
print(prices)
```

List comprehensions – set i dict comprehensions

- Chociaż list comprehentions w Pythonie jest powszechnym narzędziem, możesz także tworzyć wyrażenia złożone i słownikowe.
- Set comprehentions jest prawie dokładnie takie samo, jak list comprehentions w Pythonie.
- Różnica polega na tym, że zestawy wyrażeń zapewniają, że dane wyjściowe nie zawierają duplikatów.
- Możesz utworzyć zestaw comprehentions, używając nawiasów klamrowych zamiast nawiasów kwadratowych

```
quote = 'always look on the right side of life'
unique_vowels = {i for i in quote if i in 'aeiou'}
print(unique_vowels)

squares = {i: i * i for i in range(10)}
print(squares)
```

List comprehensions – set, list, dict przykłady

output_list = [output_exp for var in input_list if (var satisfies this condition)]

```
# Załóżmy, że chcemy utworzyć listę wyjściową, która
zawiera tylko liczby parzyste

ls = [1, 2, 3, 4, 4, 5, 6, 7, 7, 11, 15, 18]

out_ls = [var for var in ls if var % 2 == 0]

print("Output List using list comprehensions:", out_ls)


# Załóżmy, że chcemy utworzyć listę wyjściową
zawierającą kwadraty wszystkich liczb od 1 do 9

out_ls2 = [var ** 2 for var in range(1, 10)]

print("Output List using list comprehensions:", out_ls2)
```

```
# Załóżmy, że chcemy utworzyć słownik wyjściowy, który zawiera tylko liczby
nieparzyste

ls = [1, 2, 3, 4, 5, 6, 7, 11, 13, 22, 56, 77, 84, 92, 101]

out_ls3 = {var: var ** 3 for var in ls if var % 2 != 0}

print("Output Dictionary using dictionary comprehensions:", out_ls3)

# Mając dwie listy zawierające nazwy krajów i odpowiadające im stolice,
# skonstruuj słownik, który odwzorowuje stany z ich stolicami

kraj = ['Polska', 'Niemcy', 'Francja']
stolica = ['Warszawa', 'Berlin', 'Paryz']

out4 = {key: value for (key, value) in zip(kraj, stolica)}

print("Output Dictionary using dictionary comprehensions:", out4)
```

output_dict = {key:value for (key, value) in iterable if (key, value satisfy this condition)}

List comprehensions – set, list, dict przykłady

```
# Załóżmy, że chcemy utworzyć zestaw wyjściowy, który zawiera
tylko liczby parzyste.
ls = [1, 2, 3, 4, 5, 6, 7, 11, 13, 22, 56, 77, 84, 92, 101]

out = {var for var in ls if var % 2 == 0}

print("Output Set using set comprehensions:", out)

# generator

output_gen = (var for var in ls if var % 2 == 0)

print("Output values using generator comprehensions:", end=' ')

for var in output_gen:
    print(var, end=' ')
```

```
# Warunkowe if list comprehentions
out2 = [ x for x in range(30) if x % 2 == 0]
print("\nOutput using list comprehensions with if condition:", out2)

# Zagnieżdzone if w list comprehentions
out3 = [y for y in range(100) if y % 2 == 0 if y % 5 == 0]
print("Output using list comprehensions with nested if condition:", out3)

# if else w list comprehention
out4 = ["Even" if i%2==0 else "Odd" for i in range(10)]
print("Output using list comprehensions with if else condition:", out4)

# Transponowana macierz przy użyciu zagnieżdzonej pętli

matrix = [[1, 2], [3,4], [5,6], [7,8]]
out5 = [[row[i] for row in matrix] for i in range(2)]
print("Output using list comprehensions with nested for:", out5)
```

List comprehensions – Operator Morsa (Walrus Operator)

- Wyrażenie przypisania znane również jako operator morsa został wprowadzony w pythonie 3.8.
- Załóżmy, że musisz wykonać dziesięć żądań do interfejsu API, który zwróci dane dotyczące temperatury.
- Chcemy zwrócić tylko wyniki większe niż 100 stopni Fahrenheita.
- Załóżmy, że każde żądanie zwróci inne dane. W takim przypadku nie ma sposobu, aby użyć list comprehention w Pythonie do rozwiązania problemu.
- Wyrażenie *expression for member in iterable (if conditional)* nie umożliwia warunkowemu przypisania danych do zmiennej, do której wyrażenie ma dostęp.
- Operator morsa rozwiązuje ten problem. Pozwala na uruchomienie wyrażenia przy jednoczesnym przypisaniu wartości wyjściowej do zmiennej.
- Przykład pokazuje, jak jest to możliwe, używając funkcji get_weather_data() do generowania fałszywych danych pogodowych

```
import random

def get_weather_data():
    return random.randrange(90, 110)

hot_temps = [temp for _ in range(20) if (temp :=
get_weather_data()) >= 100]

print(hot_temps)
```

```
f = lambda s: s**2
dat = [1, 2, 3, 4, 5, 6, 7, 8]
filtered_power_dat = [g for s in dat if (g := f(s)) > 20]
print("Output using list comprehensions with nested for:", filtered_power_dat)


data = {"A": [40, 62, 65], "B": [35, 62, 70, 82], "C": [28, 45, 80], "D":[91, 77]}
out = {g: mean for g in data if (mean := (sum(data[g]) // len(data[g]))) > 60}
print("Output using list comprehensions with nested for:", out)
```

3 Iteratory i generatory

Iterators and generators

Iterators – wstęp

- Iterator jest obiektem który implementuje protokół iteracyjny, czyli klasa która zawiera __next()__ metodę,
- Klasa ta zawiera informacje o obecnym stanie wartości iterowanej
- Iteratory pozwalają pracować z nieskończonymi sekwencjami bez potrzeby relokowania zasobów
- Python posiada kilka wbudowanych iteratorów takich jak listy, krotki, łańcuchy znaków, słowniki i pliki.
- W Pythonie jest wiele narzędzi do iteratorów
- Methoda __iter__() zwraca obiekt iteratora
- Iterator jest obiektem który implementuje protokół iteracyjny, czyli klasa która zawiera __next()__ metodę,
- Klasa ta zawiera informacje o obecnym stanie wartości iterowanej
- Iteratory pozwalają pracować z nieskończonymi sekwencjami bez potrzeby relokowania zasobów
- Python posiada kilka wbudowanych iteratorów związanych z typami danych takich jak listy, krotki, łańcuchy znaków, słowniki.
- W Pythonie jest wiele narzędzi do iteratorów
- Methoda __iter__() zwraca obiekt iteratora
- Kontener jest jest to object przechowujące poszczególne wartości danych.
- Kontenery są iterowalne
- Kontenerami są listy, sety, słowniki, tuple, I ciągi znaków. Są nimi również otwarte pliki, otwarte sockety.

```
my_list = [5, 1, 6, 3]

my_iter = iter(my_list)
print(my_iter)
print(next(my_iter))
print(next(my_iter))
print(next(my_iter))
print(next(my_iter))
#print(next(my_iter))
```

```
mystr = "iterators"
myit = iter(mystr)

print(myit)
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

Iterators – pliki i klasy

```
# W plikach readline == next()

f1 = open('Always.txt')
print(f1.readline())
print(f1.readline())
print(f1.readline())


f2 = open('Always.txt')
print(f2.__next__())
print(f2.__next__())
print(f2.__next__())
```

```
# Skaner plików
f = open('Always.txt')
while True:
    line = f.readline()
    if not line: break
    print(line)
```

```
class MyIter(object):
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

n_list = MyIter(2,10)
print(list(n_list))
```

Iterators – metody wbudowane

- Iteratory mogą też być metoadami wbudowanymi w typ danych np. metody w słownikach takie jak keys(), values(), items()
- Są też to funkcje wbudowane takie jak range(), map(), zip(), filter()

```
# range
r = range(10)
print(r)
print(type(r))
print(list(r))

# map
from math import sqrt
m = map(sqrt, range(9))
print(m)
print(type(m))
print(list(m))

# zip
import random

gracz = ['Gracz1', 'Gracz2', 'Gracz3']
wynik = [random.randint(1, 6), random.randint(1, 6),
random.randint(1, 6)]

z = zip(gracz, wynik)
print(z)
print(type(z))
print(list(z))
```

Iterators – itertools

```
# itertools - accumulate
from operator import add
from itertools import accumulate
numbers = [5, 8, 10, 20, 50, 100]
print(list(accumulate(numbers, add)))

# itertools - chain
from itertools import chain
a = range(3)
b = range(5)
print(list(chain(a, b)))

# itertools - combination
from itertools import combinations
print(list(combinations(range(3), 2)))
```

```
# itertool - permutation
from itertools import permutations
print(list(permutations(range(3), 2)))

# itertool - compress
from itertools import compress
print(list(compress(range(1000), [0, 1, 1, 1, 0, 1, 0, 0, 1])))

# itertools - dropwhile, takewhile
from itertools import dropwhile,takewhile
print(list(dropwhile(lambda x: x <= 3, [1, 3, 5, 4, 2])))
print(list(takewhile(lambda x: x <= 3, [1, 3, 5, 4, 2])))
```

```
# itertools - islice
from itertools import islice

lines = ["line1", "line2", "line3", "line4", "line5",
         "line6", "line7", "line8", "line9", "line10"]

first_five_lines = islice(lines, 1, 5, 2)
for line in first_five_lines:
    print(line)

# itertools - pairwise
from itertools import pairwise

for pair in pairwise('ABCDEFGHI'):
    print(pair[0], pair[1])
```

Generators – przykłady

```
# function
def count(start=0, step=1, stop=10):
    n = start
    while n <= stop:
        yield n
        n += step
print(type(count))
print(type(count(10, 2.5, 20)))
for x in count(10, 2.5, 20):
    print(x)

# comprehentions
generator = (x ** 2 for x in range(4))
print(type(generator))
for x in generator:
    print(x)
```

```
class Count(object):
    def __init__(self, start=0, step=1, stop=10):
        self.n = start
        self.step = step
        self.stop = stop

    def __iter__(self):
        return self

    def __next__(self):
        n = self.n

        if n > self.stop:
            raise StopIteration()
        self.n += self.step
        return n


for x in Count(10, 2.5, 20):
    print(x)
```

Generators – yeild

```
def go(word):
    yield word
    yield 2 * word
    yield 4 * word


a = go('Hello')
print(next(a))
print(next(a))
print(next(a))
```

```
# generator
def gensq(N):
    for i in range(N):
        yield i ** 2


for x in gensq(10): print(x, end=' : ')

# comprehansion
print('\n')
for x in [n ** 2 for n in range(10)]: print(x, end=' : ')

# iterator map
print('\n')
for x in map((lambda n: n ** 2), range(10)): print(x, end=' : ')
```

Generators – przykłady

```
# Generator liczb pierwszych
def gen_primes(N):
    primes = set()
    count = 0
    n = 2
    while count < N:
        if all(n % p > 0 for p in primes):
            primes.add(n)
            yield n
            count = count + 1
        n = n + 1


print(type(gen_primes(100)))
print(gen_primes(10))
print(*gen_primes(10))
print([*gen_primes(10)])

a = [x for x in gen_primes(10)]
print(a)
```

```
# Generator liczb fibonacciego
def gen_fibonacci(N):
    c, n = 0, 1
    for i in range(N):
        c, n = n, c + n
        yield c


print(type(gen_fibonacci(10)))
print(*gen_fibonacci(10))
```

Zadanie 7

Napisz generator który tworzy słownik zawierający parę liczb pierwszych i fibonacciego z wykorzystaniem dict comprehentions.

4 Zagadnienia sieciowe

Network

Network - REST

Rest nie jest tak naprawdę słowem - to akronim. Pochodzi z trzech słów o równym znaczeniu: Representational – Reprezentacyjny, State – Stan, Transfer – Przenosić

Reprezentacyjny (Representational)

- RE oznacza Reprezentacyjny. Oznacza to, że nasza maszyneria przechowuje, przesyła i odbiera reprezentacje, podczas gdy termin reprezentacja odzwierciedla sposób, w jaki dane lub stany są przechowywane w systemie i prezentowane użytkownikom (ludziom lub komputerom).
- REST używa bardzo ciekawego sposobu przedstawiania swoich danych - zawsze jest to tekst. Czysty, prosty tekst.

Stan (State)

- S oznacza stan. Słowo stan jest kluczem do zrozumienia, czym jest REST i do czego może być używany.
- Wyobraź sobie dowolny przedmiot. Obiekt zawiera zestaw (najlepiej zestaw niepusty) właściwości.
- Można powiedzieć, że wartości wszystkich właściwości obiektu składają się na jego stan.
- Jeśli którakolwiek z właściwości zmieni swoją wartość, nieuchronnie pociąga to za sobą efekt zmiany stanu całego obiektu.
- Taka zmiana jest często nazywana przejściem.

Przenosić (Transfer)

- T oznacza transfer. Sieć (nie tylko Internet) może pełnić rolę nośnika umożliwiającego przesyłanie reprezentacji stanów do iz serwera.
- Pprzekazowi podlega nie obiekt, ale jego stany lub działania mogące zmieniać stany.
- Można powiedzieć, że przenoszenie stanów pozwala na osiągnięcie rezultatów podobnych do tych, jakie wywołują wywołania metod.

BSD sockets - wstęp

- Gniazdo (Sockets) jest rodzajem punktu końcowego (end-point). Punkt końcowy (end-point) to punkt, z którego można pobrać dane i do którego można je wysłać. Twój program w Pythonie może łączyć się z punktem końcowym (end-point) i używać go do wymiany komunikatów między sobą a innym programem działającym gdzieś daleko w Internecie.
- Historia gniazdek rozpoczęła się w 1983 roku na Uniwersytecie Kalifornijskim w Berkeley, gdzie sformułowano koncepcję i przeprowadzono pierwsze udane wdrożenie.
- Powstałe rozwiązanie było uniwersalnym zestawem funkcji, nadającym się do implementacji w niemal wszystkich systemach operacyjnych i dostępnym we wszystkich współczesnych językach programowania.
- Nazwano go BSD sockets - nazwa została zapożyczona od Berkeley Software Distribution, nazwy systemu operacyjnego klasy Unix, w którym gniazda zostały wdrożone po raz pierwszy.
- Po pewnych poprawkach standard został przyjęty przez POSIX (standard współczesnych systemów operacyjnych klasy Unix) jako gniazda POSIX.
- Można powiedzieć, że wszystkie nowoczesne systemy operacyjne implementują gniazda BSD w mniej lub bardziej dokładny sposób. Pomimo różnic, ogólna idea pozostaje taka sama.
- Gniazda BSD były oryginalnie zaimplementowane w języku programowania „C”.
- Główna idea gniazd BSD jest ściśle powiązana z filozofią Uniksa zawartą w słowach „wszystko jest plikiem”. Gniazdo może być często traktowane jako bardzo specyficzny rodzaj pliku. Zapisanie do gniazda powoduje wysłanie danych przez sieć. Odczyt z gniazda umożliwia odbiór danych pochodzących z sieci.
- Nawiasem mówiąc, MS Windows reimplementuje gniazda BSD w postaci WinSock. Na szczęście nie jesteś w stanie odczuć różnicy podczas programowania w Pythonie. Python bardzo dokładnie je ukrywa.

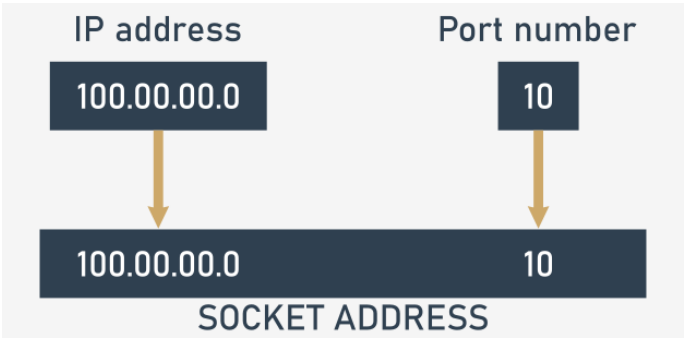
Domains, addresses, ports, protocols and services - wstęp

Doment Gniazd:

- Początkowo gniazda BSD miały na celu organizowanie komunikacji w dwóch różnych domenach Były to dwie domeny:
- Domena Unix (w skrócie Unix) - część gniazd BSD służąca do komunikowania się programów pracujących w ramach jednego systemu operacyjnego (tj. jednocześnie obecnych w tym samym systemie komputerowym)
- Domena internetowa (w skrócie INET) - część API gniazda BSD służąca do komunikowania się programów pracujących w różnych systemach komputerowych, połączonych ze sobą za pomocą sieci TCP/IP (nie wyklucza to wykorzystania gniazd INET do komunikowania się procesów pracujących w ten sam układ)

Socket address:

- Oba programy, które chcą wymieniać swoje dane, muszą umieć się identyfikować – dokładniej muszą mieć możliwość jednoznacznego wskazania gniazda, przez które chcą się połączyć.
- Gniazda domeny INET są identyfikowane (adresowane) przez pary wartości: adres IP systemu teleinformatycznego, w którym umieszczona jest gniazdo i numer portu (częściej określany jako numer usługi)



Domains, addresses, ports, protocols and services - wstęp

Adress IP:

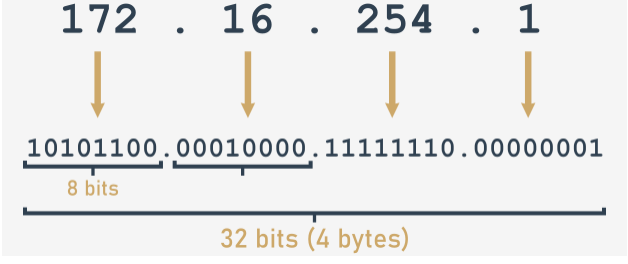
- Adres IP (dokładniej: adres IP4) to 32-bitowa wartość służąca do identyfikacji komputerów podłączonych do dowolnej sieci TCP/IP.
- Wartość jest zwykle przedstawiana jako cztery liczby z zakresu 0..255 (czyli o długości ośmiu bitów) sprzężone z kropkami (np. 87.98.239.87).
- Istnieje również nowszy standard IP, nazwany IP6, wykorzystujący 128 bitów w tym samym celu. Jendnak ograniczymy się do rozważania do IP4.

Socket/service numer:

- Numer gniazda/usługi to 16-bitowa liczba całkowita identyfikująca gniazdo w danym systemie. Jak zapewne już się domyśliłeś, istnieje 65 536 (2 ** 16) możliwych numerów gniazd/usług.
- Termin numer usługi wziął się z faktu, że wiele standardowych usług sieciowych zwykle korzysta z tych samych, stałych numerów gniazd, np. protokół HTTP, nośnik danych używany przez REST, zwykle wykorzystuje port 80.

Protokół:

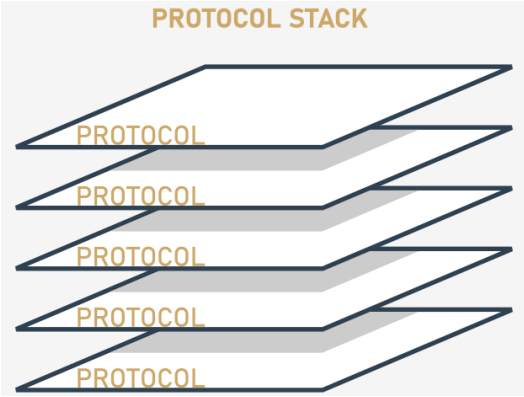
Protokół to znormalizowany zestaw reguł umożliwiający procesom wzajemną komunikację. Można powiedzieć, że protokół to swego rodzaju sieciowy savoir-vivre określający zasady zachowania wszystkich uczestników.



Domains, addresses, ports, protocols and services - wstęp

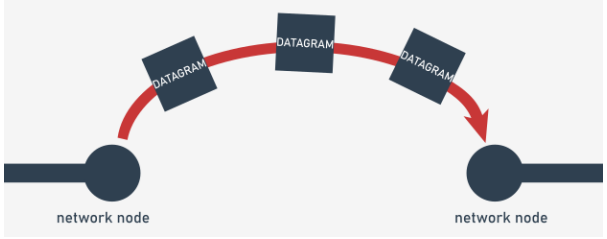
Stos protokołów:

- Stos protokołów to wielowarstwowy (stąd nazwa) zestaw współpracujących ze sobą protokołów zapewniający ujednolicony repertuar usług.
- Stos protokołów TCP/IP przeznaczony jest do współpracy z sieciami opartymi na protokole IP (sieci IP).
- Konceptualny model usług sieciowych opisuje stos protokołów w taki sposób, że najbardziej podstawowe, elementarne usługi znajdują się na dole stosu, a najbardziej zaawansowane i abstrakcyjne na górze.
- Przyjmuje się, że każda wyższa warstwa realizuje swoje funkcjonalności za pomocą usług świadczonych przez sąsiednią niższą warstwę (uwaga: jest tak samo jak w innych częściach systemu operacyjnego, np. ty program realizuje swoją funkcjonalność za pomocą usług OS, a usługi OS implementują swoje funkcjonalności z wykorzystaniem urządzeń sprzętowych).



IP:

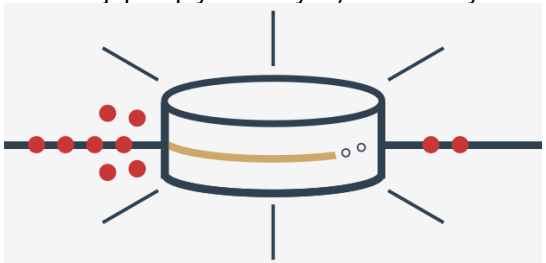
- IP (Internetwork Protocol) jest jedną z najniższych części stosu protokołów TCP/IP. Jego funkcjonalność jest bardzo prosta - jest w stanie przesłać pakiet danych (datagram) pomiędzy dwoma węzłami sieci.
- IP jest bardzo zawodnym protokołem. Nie gwarantuje, że:
    - dowolny z wysłanych datagramów dotrze do celu (ponadto, jeśli któryś z datagramów zostanie utracony, może pozostać niewykryty)
    - datagram dotrze do celu w stanie nienaruszonym,
    - para wysłanych datagramów dotrze do celu w tej samej kolejności, w jakiej zostały wysłane.
- Wyższe warstwy są w stanie zrekompensować wszystkie słabości IP.



Domains, addresses, ports, protocols and services - wstęp

TCP:

- Protokół TCP (Transmission Control Protocol) jest najwyższą częścią stosu protokołów TCP/IP.
- Wykorzystuje datagramy (dostarczane przez niższe warstwy) i uściski dłoni (zautomatyzowany proces synchronizacji przepływu danych) do budowy niezawodnego kanału komunikacyjnego zdolnego do przesyłania i odbierania pojedynczych znaków.
- Jego funkcjonalność jest bardzo rozbudowana, gdyż gwarantuje, że:
  - strumień danych dociera do celu lub nadawca zostaje poinformowany o niepowodzeniu komunikacji,
  - dane docierają do celu w stanie nienaruszonym.



UDP:

- UDP (User Datagram Protocol) znajduje się w wyższej części stosu protokołów TCP/IP, ale niżej niż TCP.
- Nie używa uścisków dłoni, co ma dwie poważne konsekwencje:
  - jest szybszy niż TCP (ze względu na mniejsze koszty ogólne)
  - jest mniej niezawodny niż TCP.
- Oznacza to:
  - TCP jest protokołem pierwszego wyboru dla aplikacji, w których bezpieczeństwo danych jest ważniejsze niż wydajność (np. WWW, REST, przesyłanie poczty itp.)
  - UDP jest bardziej odpowiedni dla aplikacji, w których czas odpowiedzi ma kluczowe znaczenie (DNS.)

Clients and servers - wstęp

Komunikacja zorientowana na połączenie a komunikacja bezpołączeniowa:

- Formą komunikacji, która wymaga pewnych wstępnych kroków w celu nawiązania połączenia i innych kroków w celu jego zakończenia, jest komunikacja zorientowana na połączenie.
- Zwykle obie strony zaangażowane w proces nie są symetryczne, tj. Ich role i procedury są różne. Obie strony komunikacji są świadome, że druga strona jest połączona.
- Rozmowa telefoniczna jest doskonałym przykładem komunikacji zorientowanej na połączenie:
  - role są ściśle określone: jest dzwoniący i jest odbiorca;
  - dzwoniący musi wybrać numer dzwoniącego i czekać, aż sieć skieruje połączenie;
  - dzwoniący musi czekać, aż dzwoniący odbierze połączenie (odbiorca może odrzucić połączenie lub po prostu nie odebrać połączenia)
  - rzeczywista komunikacja nie rozpocznie się, dopóki wszystkie poprzednie kroki nie zostaną pomyślnie zakończone;
  - komunikacja kończy się, gdy jedna ze stron rozłączy się.
- Sieci TCP/IP używają następujących nazw dla obu stron komunikacji:
  - strona inicjująca połączenie (rozmówca) nazywa się klientem;

Komunikacja, którą można nawiązać ad-hoc jest komunikacją bezpołączeniową. Obie strony mają zwykle równe prawa, ale żadna ze stron nie jest świadoma stanu drugiej strony.

Korzystanie z krótkofalówek jest bardzo dobrą analogią do komunikacji bezpołączeniowej, ponieważ:

- każda ze stron komunikacji może zainicjować komunikację w dowolnym momencie; wymaga jedynie naciśnięcia przycisku rozmowy;
- mówienie do mikrofonu nie gwarantuje, że ktoś usłyszy (żeby mieć pewność, trzeba poczekać na odpowiedź przychodzącą)

Komunikacja bezpołączeniowa jest zwykle budowana na UDP.

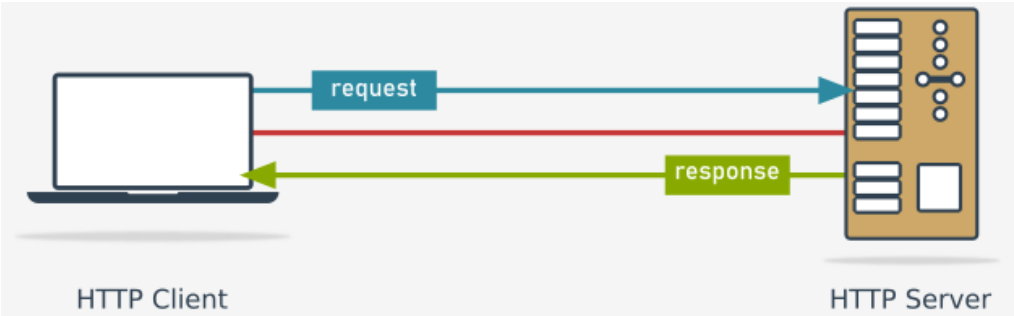
Sockets in Python - wstęp

Jak pobrać dokument z serwera za pomocą Pythona:

- chcemy napisać program, który wczytuje adres strony WWW (np. pythoninstitute.org) za pomocą standardowej funkcji input() i pobiera dokument główny (główny dokument HTML strony WWW) wskazanej strony;
- program wyświetla dokument na ekranie;
- program używa protokołu TCP do łączenia się z serwerem HTTP.

Nasz program musi wykonać następujące kroki:

1. stworzyć nowe gniazdo zdolne do obsługi transmisji zorientowanych połączeniowo w oparciu o protokół TCP;
2. połączyć gniazdo z serwerem HTTP o podanym adresie;
3. wyślij żądanie do serwera (serwer chce wiedzieć, czego od niego chcemy)
4. otrzymać odpowiedź serwera (będzie zawierać żądany dokument główny witryny)
5. zamknij gniazdko (zakończ połączenie)



Sockets in Python - użycie

Importowanie gniazda (socket): `import socket`

Dane wprowadzane przez użytkownika mogą przybierać dwie różne formy:

- może to być nazwa domeny serwera (np. www.pythoninstitute.org, ale bez wiodącego http://)
- może to być adres IP serwera (np. 87.98.235.184), ale trzeba stanowczo powiedzieć, że ten wariant jest potencjalnie niejednoznaczny. Czemu? Ponieważ pod tym samym adresem IP może znajdować się więcej niż jeden serwer HTTP - serwer, na który trafisz, może nie być serwerem, z którym zamierzałeś się połączyć.

Moduł gniazd zawiera wszystkie narzędzia potrzebne do obsługi gniazd. Nie będziemy prezentować wszystkich jego możliwości nie będziemy się skupiać na programowaniu sieciowym. Chcemy pokazać, jak działa protokół TCP/IP i jak może pełnić rolę nośnika REST.

Można powiedzieć, że TCP/IP jest dla nas interesujący tylko o tyle, o ile jest w stanie przenosić ruch HTTP, a HTTP o ile jest w stanie pełnić funkcję przekaźnika dla REST.

Moduł gniazda udostępnia klasę o nazwie gniazdo, która zawiera pakiet właściwości i działań związanych z rzeczywistym zachowaniem gniazd. Oznacza to, że pierwszym krokiem jest stworzenie obiektu klasy - tak przeprowadzamy tworzenie:

```
import socket

server_addr = input("What server do you want to connect to? ")
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Sockets in Python - użycie

Importowanie gniazda (socket): `import socket`

Dane wprowadzane przez użytkownika mogą przybierać dwie różne formy:

- może to być nazwa domeny serwera (np. www.pythoninstitute.org, ale bez wiodącego http://)
- może to być adres IP serwera (np. 87.98.235.184), ale trzeba stanowczo powiedzieć, że ten wariant jest potencjalnie niejednoznaczny. Czemu? Ponieważ pod tym samym adresem IP może znajdować się więcej niż jeden serwer HTTP - serwer, na który trafisz, może nie być serwerem, z którym zamierzałeś się połączyć.

Moduł gniazd zawiera wszystkie narzędzia potrzebne do obsługi gniazd. Nie będziemy prezentować wszystkich jego możliwości nie będziemy się skupiać na programowaniu sieciowym. Chcemy pokazać, jak działa protokół TCP/IP i jak może pełnić rolę nośnika REST.

Można powiedzieć, że TCP/IP jest dla nas interesujący tylko o tyle, o ile jest w stanie przenosić ruch HTTP, a HTTP o ile jest w stanie pełnić funkcję przekaźnika dla REST.

Moduł gniazda udostępnia klasę o nazwie gniazdo, która zawiera pakiet właściwości i działań związanych z rzeczywistym zachowaniem gniazd. Oznacza to, że pierwszym krokiem jest stworzenie obiektu klasy - tak przeprowadzamy tworzenie:

```
import socket

server_addr = input("What server do you want to connect to? ")
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Sockets in Python - użycie

Jak widać, konstruktor przyjmuje dwa argumenty, oba zadeklarowane w module.

- pierwszym argumentem jest kod domeny (możemy tu użyć symbolu AF_INET, aby określić domenę gniazda internetowego.
- domena docelowa musi być w tej chwili znana
- tym ostatnim argumentem jest kod typu gniazda (możemy tutaj użyć symbolu SOCK_STREAM, aby określić gniazdo wysokiego poziomu mogące pełnić rolę urządzenia znakowego - urządzenia obsługującego pojedyncze znaki, ponieważ interesuje nas przesyłanie danych bajt po bajcie, nie jako bloki o stałym rozmiarze (np. terminal jest urządzeniem znakowym, a dysk nie)
- Takie gniazdo jest przygotowane do pracy na protokole TCP - jest to domyślna konfiguracja gniazda.
- Jeśli chcesz utworzyć gniazdo do współpracy z innym protokołem, takim jak UDP, będziesz musiał użyć innej składni konstruktora.J
- ak widać, do nowo utworzonego obiektu gniazda będzie odwoływać się zmienna o nazwie sock.

```
import socket

server_addr = input("What server do you want to connect to? ")
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Sockets in Python – połaczenie do servera

- Jeśli korzystamy z gniazda po stronie klienta, jesteśmy gotowi do jego wykorzystania. Serwer ma jednak jeszcze kilka kroków do wykonania. Ogólnie rzecz biorąc, serwery są zwykle bardziej złożone niż klienci (jeden serwer obsługuje wielu klientów jednocześnie) - w tym momencie przestają działać nasze analogie telefoniczne.
- Skonfigurowane gniazdo (podobnie jak nasze) można podłączyć do swojego odpowiednika po stronie serwera. Spójrz na kod w edytorze - tak wykonujemy połączenie.
- Metoda connect() robi to, co obiecuje - próbuje połączyć twoje gniazdo z usługą o podanym adresie i numerze portu (usługi).
- korzystamy z wariantu, w którym dwie wartości są przekazywane do metody jako elementy krotki. Dlatego widzisz tam dwie pary nawiasów. Pominięcie jednego z nich spowoduje oczywiście błąd.
- postać docelowego adresu usługi (para składająca się z adresu rzeczywistego i numeru portu) jest specyficzna dla domeny INET.
- Jeśli coś pójdzie nie tak, metoda connect() (i każda inna metoda, której wyniki mogą się nie powieść) zgłasza wyjątek.

```
import socket

server_addr = input("What server do you want to connect to? ")
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((server_addr, 80))
```

Sockets in Python – metoda GET

- Protokół HTTP jest jednym z najprostszych protokołów internetowych
- Rozmowa z serwerem HTTP składa się z żądań (wysyłanych przez klienta) oraz odpowiedzi (wysyłanych przez serwer).
- HTTP definiuje zestaw akceptowanych żądań - są to metody żądań lub słowa HTTP. Metoda zwracająca się do serwera o przesłanie określonego dokumentu o podanej nazwie nazywa się GET
- Aby pobrać dokument główny z serwisu o nazwie www.site.com klient powinien wysłać zapytanie zawierające poprawnie sformułowany opis metody GET.

Metoda GET wymaga:

- wiersz zawierający nazwę metody (tj. GET), po której następuje nazwa zasobu, który klient chce otrzymać; dokument główny jest określony jako pojedynczy ukośnik (tj. /);
- wiersz musi zawierać również wersję protokołu HTTP (tj. HTTP/1.1) i musi kończyć się znakami \r\n; uwaga: wszystkie linie muszą kończyć się w ten sam sposób;
- linia zawierająca nazwę witryny (np. www.site.com) poprzedzoną nazwą parametru (np. Host:)
- wiersz zawierający parametr Connection: wraz z jego wartością close, który wymusza na serwerze zamknięcie połączenia po obsłużeniu pierwszego żądania; uprości to kod naszego klienta;
- pusta linia jest terminatorem żądania.

```
GET / HTTP/1.1\r\n
Host: www.site.com\r\n
Connection: close\r\n
\r\n
```

Sockets in Python – Żądanie dokumentu z serwera

- Metoda send() natywnie nie przyjmuje łańcuchów znaków - dlatego musimy użyć przedrostka b przed literalną częścią ciągu żądania (po cichu tłumaczy łańcuch na bajty - niezmienny wektor składający się z wartości z zakresu 0. .255, który send() lubi najbardziej) i dlatego też powinniśmy wywołać bytes() w celu przetłumaczenia zmiennej łańcuchowej w ten sam sposób.
- Drugi argument bytes określa kodowanie używane do przechowywania nazwy serwera. UTF8 wydaje się być najlepszym wyborem dla większości nowoczesnych systemów operacyjnych
- Akcja wykonywana przez metodę send() jest niezwykle skomplikowana - angażuje nie tylko wiele warstw systemu operacyjnego, ale także wiele urządzeń sieciowych rozmieszczonych na trasie między klientem a serwerem oraz oczywiście sam serwer.
- Oczywiście, jeśli coś w tym złożonym mechanizmie zawiedzie, wysyłanie również się nie powiedzie. Jak można się spodziewać, zgłaszany jest wtedy wyjątek.

```
import socket

server_addr = input("What server do you want to connect to? ")
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
sock.connect((server_addr, 80))
sock.send(b"GET / HTTP/1.1\r\nHost: " +
        bytes(server_addr, "utf8") +
        b"\r\nConnection: close\r\n\r\n")
```

Sockets in Python – Żądanie dokumentu z serwera

- Metoda recv() czeka na odpowiedź serwera, pobiera ją i umieszcza w nowo utworzonym obiekcie typu bytes. Spójrz na kod, który udostępniliśmy w edytorze.
- Argument określa maksymalną dopuszczalną długość danych do odebrania. Jeśli odpowiedź serwera jest dłuższa niż ten limit, pozostanie ona nieodebrana.
- Będziesz musiał ponownie wywołać recv() (może więcej niż raz), aby uzyskać pozostałą część danych. Powszechną praktyką jest wywoływanie funkcji recv() tak długo, jak zwraca ona część danych.
- Transmisja może również powodować pewne błędy.

```
import socket

server_addr = input("What server do you want to connect to? ")
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((server_addr, 80))
sock.send(b"GET / HTTP/1.1\r\nHost: " +
        bytes(server_addr, "utf8") +
        b"\r\nConnection: close\r\n\r\n")
reply = sock.recv(10000)
```

Sockets in Python – Zamykanie połączenia

- Wywołanie shutdown() jest jak wiadomość wyszeptana bezpośrednio do ucha serwera że nie chcemy odbierać informacji.
- Następujące argumenty funkcji mówią jak zakończyć połaszczenie:
- socket.SHUT_RD - nie będziemy już czytać wiadomości serwera (oświadczamy, że jesteśmy głusi)
- socket.SHUT_WR - nie powiemy ani słowa (właściwie będziemy głupi)
- socket.SHUT_RDWR - określa koniunkcję dwóch poprzednich opcji.
- Ponieważ nasze żądanie GET zażądało od serwera zamknięcia połączenia, gdy tylko odpowiedź zostanie wysłana, a serwer został poinformowany o naszych dalszych krokach (a raczej o tym, że zrobiliśmy już to, co chcieliśmy), możemy założyć, że połączenie jest w tym momencie całkowicie przerwane.
- Niektórzy powiedzieliby, że zamknięcie go jawnie jest przesadną starannością.
- Zrobi to za nas bezparametrowa metoda close() - zobacz nasz kod w edytorze.
- Wydrukujemy go za pomocą wbudowanej funkcji repr(), która dba o przejrzystą (prawie) tekstową prezentację dowolnego obiektu.
- Dlatego ostatnia linia naszego kodu wygląda następująco: print(repr(answ))

```
import socket

server_addr = input("What server do you want to connect to? ")
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((server_addr, 80))
sock.send(b"GET / HTTP/1.1\r\nHost: " +
        bytes(server_addr, "utf8") +
        b"\r\nConnection: close\r\n\r\n")
reply = sock.recv(10000)
sock.shutdown(socket.SHUT_RDWR)
sock.close()
print(repr(reply))
```

Sockets in Python – Zamykanie połączenia

- pierwszy to nagłówek odpowiedzi. Najważniejsza jest najwyższa linia, czyli to, czy serwer odesłał żądany dokument, czy nie. 200.
- dokument. Tak, to jest miejsce, w którym się zaczyna. Może być bardzo duży

```
import socket

#server_addr = input("What server do you want to connect to?
")
server_addr = "www.google.com"
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
sock.connect((server_addr, 80))
sock.send(b"GET / HTTP/1.1\r\nHost: " +
        bytes(server_addr, "utf8") +
        b"\r\nConnection: close\r\n\r\n")
reply = sock.recv(10000)
sock.shutdown(socket.SHUT_RDWR)
sock.close()
print(repr(reply))
```

Sockets in Python – błędne połączenie

Wprowadzenie nieistniejącego/błędnie sformułowanego adresu:

- Funkcja connect zgłasza wyjątek o nazwie socket.gaierror, a jej nazwa pochodzi od nazwy funkcji niskiego poziomu (zwykle dostarczanej nie przez Pythona, ale przez jądro systemu operacyjnego) o nazwie getaddrinfo(). Funkcja próbuje m.in. znaleźć pełną informację adresową dotyczącą otrzymanego argumentu.
- metoda connect() używa funkcji getaddrinfo() do nawiązywania nowego połączenia z serwerem. Jeśli getaddreinfo() zawiedzie, zgłaszany jest wyjątek i podróż kończy się przed rozpoczęciem.
- socket.gaierror obejmuje więcej niż jedną możliwą przyczynę niepowodzenia. Nasz przykład pokazuje dwa z nich - pierwszy, gdy adres jest poprawny składniowo, ale nie odpowiada żadnemu istniejącemu serwerowi, a drugi, gdy adres jest wyraźnie zniekształcony.
- Możliwe jest również, że serwer o podanym adresie istnieje i działa, ale nie zapewnia żądanej usługi. Na przykład dedykowany serwer pocztowy może nie odpowiadać na połączenia adresowane do portu o numerze 80.
- Jeśli chcesz wywołać takie zdarzenie, zamień 80 w wywołaniu connect() na dowolną pięciocyfrową liczbę nieprzekraczającą 65535 (11111 wydaje się dobrym pomysłem) i uruchom kod.

```
import socket

#server_addr = "www.google.com"
server_addr = "a.non.existent.name"
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((server_addr, 80))
sock.connect((server_addr, 11111))
sock.send(b"GET / HTTP/1.1\r\nHost: " +
        bytes(server_addr, "utf8") +
        b"\r\nConnection: close\r\n\r\n")
reply = sock.recv(10000)
sock.shutdown(socket.SHUT_RDWR)
sock.close()
print(repr(reply))
```

Wyjątek socket.timeout

- Ostatnim wyjątkiem, o którym chcemy ci powiedzieć, jest socket.timeout.
- Wyjątek ten jest zgłaszany, gdy reakcja serwera nie nastąpi w rozsądnym czasie - długość naszej cierpliwości można ustawić metodą o nazwie settimeout().

Sockets in Python – API

Wiecej informacji można znaleźć pod adresem https://docs.python.org/3/library/socket.html

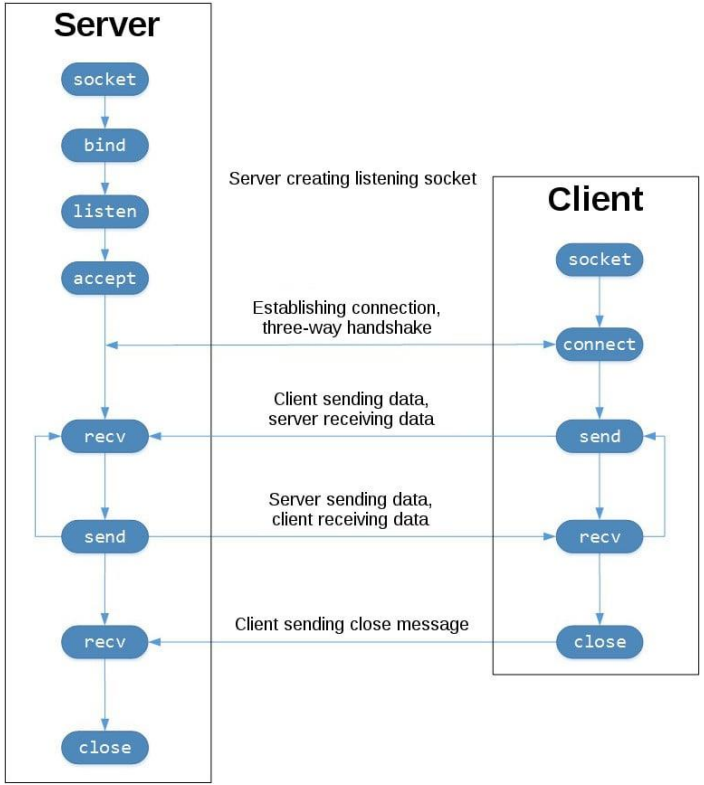
Podstawowe funkcje i metody interfejsu API gniazda w tym module to:

- socket()
- .bind()
- .listen()
- .accept()
- .connect()
- .connect_ex()
- .send()
- .recv()
- .close()

Python zapewnia wygodny i spójny interfejs API, który odwzorowuje bezpośrednio wywołania systemowe, ich odpowiedniki w C. W ramach swojej standardowej biblioteki Python ma również klasy, które ułatwiają korzystanie z funkcji gniazd niskiego poziomu. (https://docs.python.org/3/library/socketserver.html) Dostępnych jest również wiele modułów implementujących protokoły internetowe wyższego poziomu, takie jak HTTP i SMTP. Aby zapoznać się z omówieniem, zobacz Protokoły internetowe i pomoc techniczna (https://docs.python.org/3/library/internet.html)

Sockets in Python – TCP Sockets

- Tworzenie gniazda odbywa się za pomocą socket.socket(), określając typ gniazda jako socket.SOCK_STREAM. Domyślnym protokołem, który jest używany, jest protokół kontroli transmisji (TCP).
- Protokół kontroli transmisji (TCP) jest niezawodny, pakiety pozostawione w sieci są wykrywane i retransmitowane przez nadawcę. Ma dostarczanie danych w kolejności: Dane są odczytywane przez Twoją aplikację w kolejności, w jakiej zostały zapisane przez nadawcę.
- Natomiast gniazda UDP (User Datagram Protocol) utworzone za pomocą socket.SOCK_DGRAM nie są niezawodne, a dane odczytywane przez odbiorcę mogą być poza kolejnością zapisów nadawcy.
- Urządzenia sieciowe, takie jak routery i przełączniki, mają dostępną ograniczoną przepustowość i mają własne, nieodłączne ograniczenia systemowe. Mają procesory, pamięć, magistrale i bufory pakietów interfejsów, podobnie jak klienci i serwery.
- Protokół TCP zwalnia Cię z konieczności martwienia się o utratę pakietów, napływ danych poza kolejnością i inne pułapki, które niezmiennie zdarzają się podczas komunikacji przez sieć.



Sockets in Python – TCP Sockets

- Lewa kolumna reprezentuje serwer. Po prawej stronie jest klient.
- Zaczynając od lewej górnej kolumny, zwróć uwagę na wywołania API, które wykonuje serwer, aby skonfigurować gniazdo „nasłuchujące”

socket()

.bind()

.listen()

.accept()

- Gniazdo odsłuchowe robi dokładnie to, co sugeruje jego nazwa. Nasłuchuje połączeń od klientów. Gdy klient się łączy, serwer wywołuje metodę .accept() w celu zaakceptowania lub zakończenia połączenia.
- W środku znajduje się sekcja okrężna, w której dane są wymieniane między klientem a serwerem za pomocą wywołań .send() i .recv().
- Na dole klient i serwer zamykają swoje gniazda.

Sockets in Python – TCP Sockets

Server

```
import socket

HOST = "127.0.0.1"
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            print(data)
            if not data:
                break
            conn.sendall(data)
```

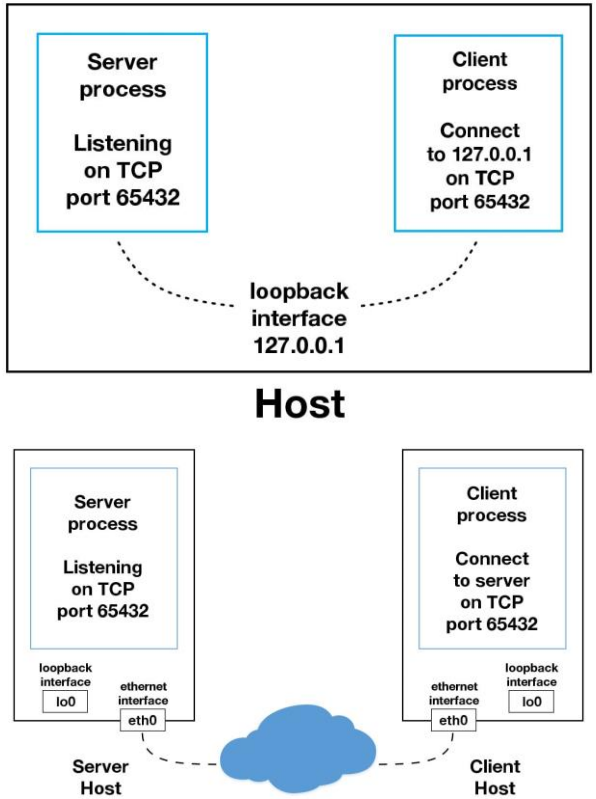
Client

```
import socket

HOST = "127.0.0.1"
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b"Hello, world")
    data = s.recv(1024)

print(f"Received {data!r}")
```

- Aby przetestować client server należy uruchomić w terminalu server a następnie w drugim terminalu client
- Komendą netstat -an można sprawdzić że połaczenie jest aktywne

Sockets in Python – Przerwa w komunikacji

- Podczas korzystania z interfejsu pętli zwrotnej (adres IPv4 127.0.0.1 lub adres IPv6 ::1) dane nigdy nie opuszczają hosta ani nie docierają do sieci zewnętrznej. Na powyższym diagramie interfejs sprzężenia zwrotnego znajduje się wewnątrz hosta. Reprezentuje to wewnętrzną naturę interfejsu sprzężenia zwrotnego i pokazuje, że połączenia i dane, które przez niego przechodzą, są lokalne dla hosta. Dlatego też usłyszysz interfejs sprzężenia zwrotnego i adres IP 127.0.0.1 lub ::1 określany jako „localhost”.
- Aplikacje wykorzystują interfejs sprzężenia zwrotnego do komunikacji z innymi procesami działającymi na hoście oraz w celu zapewnienia bezpieczeństwa i izolacji od sieci zewnętrznej. Ponieważ jest wewnętrzny i dostępny tylko z poziomu hosta, nie jest widoczny.
- Możesz to zobaczyć w akcji, jeśli masz serwer aplikacji, który korzysta z własnej prywatnej bazy danych. Jeśli nie jest to baza danych używana przez inne serwery, prawdopodobnie jest skonfigurowana do nasłuchiwania połączeń tylko na interfejsie pętli zwrotnej. W takim przypadku inne hosty w sieci nie mogą się z nim połączyć.
- Kiedy używasz w swoich aplikacjach adresu IP innego niż 127.0.0.1 lub ::1, jest on prawdopodobnie powiązany z interfejsem Ethernet podłączonym do sieci zewnętrznej. To jest twoja brama do innych gospodarzy poza twoim królestwem „localhost”:



Sockets in Python – Obsługa wielu połączeń - server

- Największą różnicą między tym serwerem a poprzednim jest wywołanie lsock.setblocking(False) w celu skonfigurowania gniazda w trybie nieblokującym. Wywołania do tego gniazda nie będą już blokowane. Kiedy jest używany z sel.select() możesz czekać na zdarzenia w jednym lub kilku gniazdach, a następnie odczytywać i zapisywać dane, gdy będą gotowe.
- sel.register() rejestruje gniazdo, które ma być monitorowane za pomocą sel.select() dla interesujących nas zdarzeń. Dla gniazda nasłuchującego chcesz odczytywać zdarzenia: selektory.EVENT_READ.
- Aby przechowywać dowolne dane wraz z gniazdem, użyjesz danych. Jest zwracany, gdy powraca funkcja .select(). Będziesz używać danych do śledzenia tego, co zostało wysłane i odebrane w gnieździe.

```
import sys
import socket
import selectors
import types

sel = selectors.DefaultSelector()
```

```
host, port = sys.argv[1], int(sys.argv[2])
lsock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
lsock.bind((host, port))
lsock.listen()
print(f"Listening on {(host, port)}")
lsock.setblocking(False)
sel.register(lsock, selectors.EVENT_READ,
data=None)
```

Sockets in Python – Obsługa wielu połączeń - server

Pętla zdarzeń:

- sel.select(timeout=None) blokuje, dopóki gniazda nie będą gotowe do wejścia/wyjścia. Zwraca listę krotek, po jednej dla każdego gniazda. Każda krotka zawiera klucz i maskę. Kluczem jest SelectorKey o nazwie Tuple, który zawiera atrybut fileobj. key.fileobj jest obiektem gniazda, a mask jest maską zdarzenia operacji, które są gotowe.
- Jeśli key.data to None, to wiesz, że pochodzi z gniazda nasłuchującego i musisz zaakceptować połączenie. Wywołasz własną funkcję accept_wrapper(), aby pobrać nowy obiekt gniazda i zarejestrować go w selektorze.
- Jeśli key.data nie ma wartości None, to wiesz, że jest to gniazdo klienta, które zostało już zaakceptowane i musisz je obsłużyć. Następnie wywoływana jest funkcja service_connection() z parametrami key i mask jako argumentami i to wszystko, czego potrzebujesz do działania na gnieździe.

```
try:
    while True:
        events = sel.select(timeout=None)
        for key, mask in events:
            if key.data is None:
                accept_wrapper(key.fileobj)
            else:
                service_connection(key, mask)
except KeyboardInterrupt:
    print("Caught keyboard interrupt, exiting")
finally:
    sel.close()
```

Sockets in Python – Obsługa wielu połączeń - server

accept_wrapper()

- Ponieważ gniazdo nasłuchujące zostało zarejestrowane dla selektorów zdarzeń.EVENT_READ, powinno być gotowe do odczytu. Wywołujesz metodę sock.accept() , a następnie conn.setblocking(False), aby przełączyć gniazdo w tryb nieblokujący.
- Pamiętaj, że jest to główny cel w tej wersji serwera, ponieważ nie chcesz, aby się blokował. Jeśli się zablokuje, cały serwer zostanie zablokowany, dopóki nie wróci. Oznacza to, że inne gniazda czekają, nawet jeśli serwer nie działa aktywnie. Jest to przerażający stan „zawieszenia”, w którym nie chcesz, aby znajdował się twój serwer.
- następnie tworzysz obiekt do przechowywania danych, które chcesz dołączyć wraz z gniazdem, używając SimpleNamespace. Ponieważ chcesz wiedzieć, kiedy połączenie klienta jest gotowe do odczytu i zapisu, oba te zdarzenia są ustawiane za pomocą bitowego operatora OR
- Maska zdarzeń, gniazdo i obiekty danych są następnie przekazywane do sel.register()

```
def accept_wrapper(sock):
    conn, addr = sock.accept()  # Should be ready to read
    print(f"Accepted connection from {addr}")
    conn.setblocking(False)
    data = types.SimpleNamespace(addr=addr, inb=b"", outb=b"")
    events = selectors.EVENT_READ | selectors.EVENT_WRITE
    sel.register(conn, events, data=data)
```

Sockets in Python – Obsługa wielu połączeń - server

service_connection()

- To serce prostego serwera z wieloma połączeniami. key to nazwana krotka zwrócona przez .select(), która zawiera obiekt gniazda (fileobj) i obiekt danych. mask zawiera zdarzenia, które są gotowe.
- Jeśli gniazdo jest gotowe do odczytu, mask & selectors.EVENT_READ zwróci wartość True, więc wywoływana jest sock.recv(). Wszelkie odczytane dane są dołączane do pliku data.outb, dzięki czemu można je później wysłać.
- Jeśli żadne dane nie zostaną odebrane, oznacza to, że klient zamknął swoje gniazdo, więc serwer też powinien. Ale nie zapomnij wywołać sel.unregister() przed zamknięciem, aby nie było już monitorowane przez .select().
- Gdy gniazdo jest gotowe do zapisu, co zawsze powinno mieć miejsce w przypadku sprawnego gniazda, wszelkie otrzymane dane przechowywane w data.outb są wysyłane do klienta za pomocą funkcji sock.send(). Wysłane bajty są następnie usuwane z bufora wysyłania

```
def service_connection(key, mask):
    sock = key.fileobj
    data = key.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(1024)  # Should be ready to read
        if recv_data:
            data.outb += recv_data
        else:
            print(f"Closing connection to {data.addr}")
            sel.unregister(sock)
            sock.close()
    if mask & selectors.EVENT_WRITE:
        if data.outb:
            print(f"Echoing {data.outb!r} to {data.addr}")
            sent = sock.send(data.outb)  # Should be ready to write
            data.outb = data.outb[sent:]
```

Sockets in Python – Obsługa wielu połączeń - client

- num_conns jest odczytywane z wiersza poleceń i jest liczbą połączeń do utworzenia z serwerem. Podobnie jak serwer, każde gniazdo jest ustawione na tryb nieblokujący.
- Używasz .connect_ex() zamiast .connect() ponieważ .connect() natychmiast zgłasza wyjątek BlockingIOError. Metoda .connect_ex() początkowo zwraca wskaźnik błędu, errno.EINPROGRESS, zamiast zgłaszać wyjątek, który zakłócałby trwające połączenie. Po nawiązaniu połączenia gniazdo jest gotowe do odczytu i zapisu i jest zwracane przez funkcję .select().
- Po skonfigurowaniu gniazda dane, które mają być przechowywane w gnieździe, są tworzone przy użyciu SimpleNamespace. Wiadomości, które klient wyśle do serwera, są kopiowane przy użyciu funkcji message.copy(), ponieważ każde połączenie wywołuje funkcję socket.send() i modyfikuje listę. Wszystko, co jest potrzebne do śledzenia tego, co klient musi wysłać, wysłał i otrzymał, w tym całkowitą liczbę bajtów w wiadomościach, jest przechowywane w danych obiektowych.

```
def start_connections(host, port, num_conns):
    server_addr = (host, port)
    for i in range(0, num_conns):
        connid = i + 1
        print(f"Starting connection {connid} to {server_addr}")
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.setblocking(False)
        sock.connect_ex(server_addr)
        events = selectors.EVENT_READ | selectors.EVENT_WRITE
        data = types.SimpleNamespace(
            connid=connid,
            msg_total=sum(len(m) for m in messages),
            recv_total=0,
            messages=messages.copy(),
            outb=b"",
        )
        sel.register(sock, events, data=data)
```

Sockets in Python – Obsługa wielu połączeń - client

- To zasadniczo to samo, ale z jedną ważną różnicą. Klient śledzi liczbę bajtów otrzymanych z serwera, aby mógł zamknąć swoją stronę połączenia. Gdy serwer to wykryje, zamyka również swoją stronę połączenia.
- Zauważ, że robiąc to, serwer zależy od tego, czy klient dobrze się zachowuje: serwer oczekuje, że klient zamknie swoją stronę połączenia, gdy zakończy wysyłanie wiadomości. Jeśli klient się nie zamknie, serwer pozostawi otwarte połączenie. W rzeczywistej aplikacji możesz chcieć zabezpieczyć się przed tym na swoim serwerze, wprowadzając limit czasu, aby zapobiec gromadzeniu się połączeń klientów, jeśli nie wyślą żądania po pewnym czasie.

```
def service_connection(key, mask):
    sock = key.fileobj
    data = key.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(1024)  # Should be ready to read
        if recv_data:
            print(f"Received {recv_data!r} from connection
{data.connid}")
            data.recv_total += len(recv_data)
        if not recv_data or data.recv_total == data.msg_total:
            print(f"Closing connection {data.connid}")
            sel.unregister(sock)
            sock.close()
    if mask & selectors.EVENT_WRITE:
        if not data.outb and data.messages:
            data.outb = data.messages.pop(0)
        if data.outb:
            print(f"Sending {data.outb!r} to connection {data.connid}")
            sent = sock.send(data.outb)  # Should be ready to write
            data.outb = data.outb[sent:]
```

- Aby przetestować client server należy uruchomić w terminalu server a następnie w drugim terminalu client

JSON – praca z jsonem w pythonie

- Json można wykorzystać w pythonie poprzez

```
import json
```

- Siłą modułu JSON jest możliwość automatycznej konwersji danych Pythona (nie wszystkich i nie zawsze) na ciąg znaków JSON. Jeśli chcesz przeprowadzić taką operację, możesz użyć funkcji o nazwie dumps().
- Funkcja robi to, co obiecuje – pobiera dane (nawet nieco skomplikowane) i generuje ciąg znaków wypełniony komunikatem JSON. Oczywiście dumps() nie jest prorokiem i nie jest w stanie czytać w twoich myślach, więc nie oczekuj cudów.

```
import json

value = 1.602176620898e-19
print(json.dumps(value))

comics = '"The Meaning of Life" by Monty Python\'s Flying Circus'
print(json.dumps(comics))

my_list = [1, 2.34, True, "False", None, ['a', 0]]
print(json.dumps(my_list))

my_dict = {'me': "Python", 'pi': 3.141592653589, 'data': (1, 2, 4, 8), 'set':
None}
print(json.dumps(my_dict))
```

JSON – praca z jsonem w pythonie

- Jak widać, Python używa niewielkiego zestawu prostych reguł do budowania komunikatów JSON ze swoich natywnych danych

Python data	JSON element
dict	object
list or tuple	array
string	string
int or float	number
True / False	true / false
None	null

- Oczywiście, jeśli nie potrzebujesz niczego więcej niż zestaw właściwości obiektu i ich wartości, możesz wykonać sztuczkę i zrzucić nie sam obiekt, ale zawartość jego właściwości __dict__.
- Istnieją co najmniej dwie opcje, z których możemy skorzystać. Pierwsza z nich opiera się na fakcie, że możemy zastąpić funkcję dumps() w celu uzyskania tekstowej reprezentacji jej argumentu.
  Należy wykonać dwa kroki:
  - napisz własną funkcję, wiedząc, jak obsługiwać swoje obiekty;
  - uświadomić to funkcji dumps(), ustawiając argument słowa kluczowego o nazwie default;
- Dzięki temu zapiszemy nazwy właściwości wraz z ich wartościami. Sprawi, że JSON będzie łatwiejszy do odczytania i bardziej zrozumiały dla ludzi.
- Zgłoszenie wyjątku TypeError jest obowiązkowe – to jedyny sposób na poinformowanie funkcji dumps(), że funkcja nie jest w stanie konwertować obiektów innych niż pochodzące z klasy Who
- proces, w którym obiekt (przechowywany wewnętrznie przez Pythona) jest konwertowany na tekstowy lub inny przenośny aspekt, jest często nazywany serializacją. Podobnie działanie odwrotne (z przenośnego na wewnętrzne) nazywa się deserializacją.

```
import json


class Who:
    def __init__(self, name, age):
        self.name = name
        self.age = age


def encode_who(w):
    if isinstance(w, Who):
        return w.__dict__
    else:
        raise TypeError(w.__class__.__name__ + ' is not JSON
serializable')


some_man = Who('John Doe', 42)
print(json.dumps(some_man, default=encode_who))
```

JSON – praca z jsonem w pythonie

- Drugie podejście opiera się na fakcie, że serializacja jest faktycznie wykonywana metodą o nazwie default(), która jest częścią klasy json.JSONEncoder. Daje to możliwość przeciążenia metody definiującej podklasę JSONEncoder i przekazania jej do funkcji dumps() za pomocą argumentu słowa kluczowego o nazwie cls – tak jak w kodzie obok.
- Funkcja, która jest w stanie pobrać ciąg znaków JSON i zamienić go na dane Pythona, nazywa się load() – pobiera ciąg znaków (stąd s na końcu nazwy) i próbuje utworzyć obiekt Pythona odpowiadający otrzymanym danym .
- Funkcja load() radzi sobie również z ciągami znaków
- Podwójne ukośniki odwrotne są potrzebne ponieważ musimy dostarczyć dokładny łańcuch JSON do funkcji load(). Oznacza to, że ukośnik odwrotny musi poprzedzać wszystkie cudzysłowy istniejące w ciągu. Usunięcie któregokolwiek z nich spowoduje, że łańcuch będzie nieważny, a load() na pewno zwrócić błąd.

```
import json

jstr = '16021766189.98'
electron = json.loads(jstr)
print(type(electron))
print(electron)

jstr = '"\\"The Meaning of Life\\" by Monty Python\'s Flying Circus"'
comics = json.loads(jstr)
print(type(comics))
print(comics)
```

```
import json


class Who:
    def __init__(self, name, age):
        self.name = name
        self.age = age


class MyEncoder(json.JSONEncoder):
    def default(self, w):
        if isinstance(w, Who):
            return w.__dict__
        else:
            return super().default(self, w)


some_man = Who('John Doe', 42)
print(json.dumps(some_man, cls=MyEncoder))
```

JSON – praca z jsonem w pythonie

- loads() radzi sobie również z listami, i słownikami
- jeśli liczba zakodowana w ciągu JSON nie ma części ułamkowej, Python utworzy liczbę całkowitą lub liczbę zmiennoprzecinkową w przeciwnym razie
- W pythonie można deserializować w ten sam spocób co przeprowadziliśmy serializację ale może wymagać to dodatkowych czynności.
- Funkcja load() nie jest w stanie odgadnąć który obiekt faktycznie potrzebujesz deserializować , należy podać odpowiednie informacje.
- Słowo kluczowe object_hook służy do wskazania funkcji odpowiedzialnej za utworzenie zupełnie nowego obiektu wymaganej klasy i wypełnienie go rzeczywistymi danymi
- funkcja decode_who() otrzymuje obiekt Pythona, a dokładniej – słownik. Ponieważ konstruktor Who oczekuje dwóch zwykłych wartości, ciągu znaków i liczby, a nie słownika, musimy zastosować małą sztuczkę – użyliśmy operatora podwójnego *, aby zamienić katalog w listę argumentów słów kluczowych zbudowaną z klucza słownika :pary wartości. Oczywiście klucze w słowniku muszą mieć takie same nazwy jak parametry konstruktora
- funkcja określona przez object_hook zostanie wywołana tylko wtedy, gdy łańcuch JSON opisuje obiekt JSON

```
jstr = '[1, 2.34, true, "False", null, ["a", 0]]'
my_list = json.loads(jstr)
print(type(my_list))
print(my_list)

json_str =
'{"me":"Python","pi":3.141592653589,
"data":[1,2,4,8],"friend":"JSON","set": null}'
my_dict = json.loads(json_str)
print(type(my_dict))
print(my_dict)
```

```
class Who:
    def __init__(self, name, age):
        self.name = name
        self.age = age


def encode_who(w):
    if isinstance(w, Who):
        return w.__dict__
    else:
        raise TypeError(w.__class__.__name__ +
'is not JSON serializable')


def decode_who(w):
    return Who(w['name'], w['age'])


old_man = Who("Jane Doe", 23)
json_str = json.dumps(old_man,
default=encode_who)
new_man = json.loads(json_str,
object_hook=decode_who)
print(type(new_man))
print(new_man.__dict__)
```

JSON – praca z jsonem w pythonie

- Możliwe jest również podejście czysto obiektowe, które opiera się na ponownym zdefiniowaniu klasy JSONDecoder. Niestety ten wariant jest bardziej skomplikowany niż jego kodujący odpowiednik.
- Nie musimy przepisywać żadnej metody, ale musimy przedefiniować konstruktor nadklasy
- Nowy konstruktor ma wykonać tylko jedną sztuczkę – ustawić funkcję do tworzenia obiektów.

```
import json


class Who:
    def __init__(self, name, age):
        self.name = name
        self.age = age


class MyEncoder(json.JSONEncoder):
    def default(self, w):
        if isinstance(w, Who):
            return w.__dict__
        else:
            return super().default(self, w)


class MyDecoder(json.JSONDecoder):
    def __init__(self):
        json.JSONDecoder.__init__(self, object_hook=self.decode_who)

    def decode_who(self, d):
        return Who(**d)


some_man = Who('Jane Doe', 23)
json_str = json.dumps(some_man, cls=MyEncoder)
new_man = json.loads(json_str, cls=MyDecoder)

print(type(new_man))
print(new_man.__dict__)
```

HTTP – json server

- HTTP będzie pracował tylko dla nas i z powodzeniem pełnić rolę fundamentu RESTFul API.
- Wykorzstany do tego zostanie json-server, zaimplementowany na środowisku Node.js
- Będzie działać dla nas jak czarna skrzynka.

Przygotowanie serwera:

- Dla systemu Windows - https://nodejs.org/en/download, pobierz i uruchom instalator Windows (plik o nazwie wyglądającej jak node-vxx.yy.z-x86.msi) z gałęzi LTS (Long Time Support); zaakceptuj wszystkie ustawienia domyślne i pozwól instalatorowi wykonać zadanie; po udanej instalacji powinieneś zobaczyć wpis o nazwie Node.js w menu startowym systemu Windows®
- Dla macOS, wejdź na adres https://nodejs.org/en/download, pobierz plik pkg i uruchom instalator. Cały proces wygląda tak samo jak w systemie Windows® - po prostu zaakceptuj wszystkie domyślne ustawienia i pozwól instalatorowi wykonać pracę;
- Dla Linuksa, musisz wejść na adres https://nodejs.org/en/download/package-manager i uzyskać bardziej szczegółową pomoc; niestety niektóre Linuksy oferują własne, natywne pakiety Node.js dopasowane do konkretnych potrzeb systemowych i te pakiety można zainstalować za pomocą wbudowanego menedżera pakietów, podczas gdy inne wymagają instalacji ręcznej; przepraszamy, nie możemy pomóc w rozwiązaniu tego problemu.

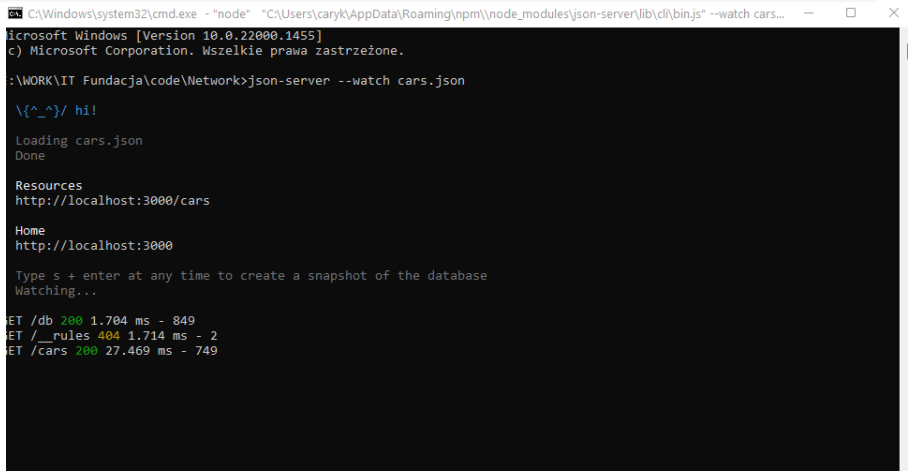
HTTP – json server

- Następnym krokiem jest otwarcie konsoli systemu operacyjnego
- dla systemu Windows, uruchom program CMD.EXE,
- dla systemu Linuksa lub macOS, uruchom swój ulubiony emulator terminala
- Kolejne kroki będą prawie takie same na wszystkich powyższych platformach.
- Node.js wykorzystuje własne natywne narzędzie do instalowania i aktualizowania komponentów. (npm)
- Używając komendy npm instalujemy json server,

```
npm install -g json-server
```

- Możemy przetestować czy działa poprawnie przy użyciu pliku cars.json

```
json-server --watch cars.json
```

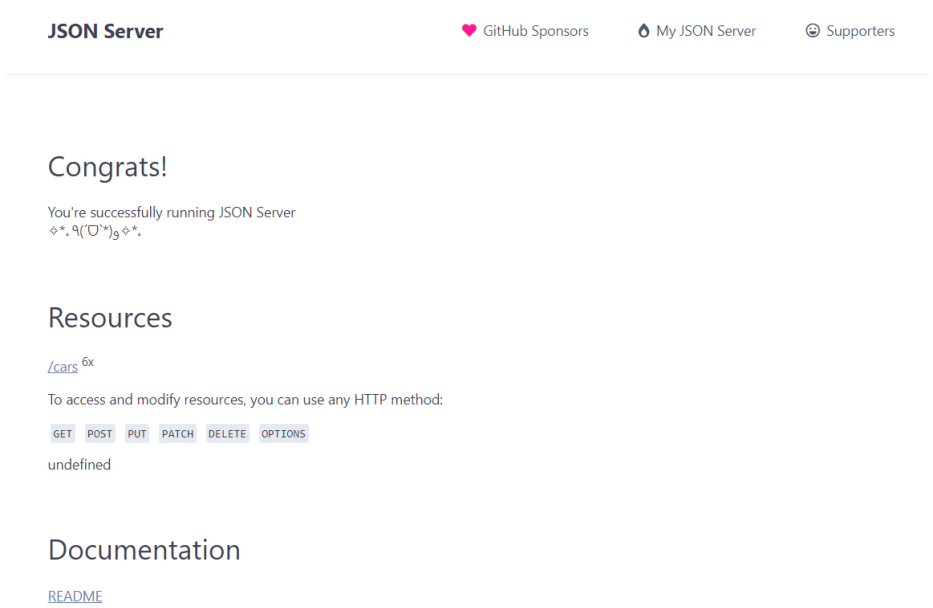


HTTP – json server

- Możemy teraz sprawdzić server w przeglądarce

```
http://localhost:3000
```

- jeśli klient nie określi zasobu, który chce uzyskać, json-server wysyła ekran powitalny.
- Jeśli chcesz, aby serwer zachowywał się w inny sposób, wykonaj następujące kroki:
  - w folderze, w którym znajduje się cars.json, utwórz podfolder o nazwie public;
  - w folderze publicznym utwórz plik o nazwie index.html, wypełnij go tekstem i zapisz.
  - Stworzyliśmy taki plik zawierający jedną linię



HTTP – requests

- Protokół HTTP działa przy użyciu metod. Można powiedzieć, że metoda HTTP jest dwukierunkową interakcją pomiędzy klientem a serwerem (uwaga: klient inicjuje transmisję) poświęconą wykonaniu określonej akcji.
- GET jest jednym z nich i służy do przekonania serwera do przesłania niektórych zasobów, o które prosi klient.
- Funkcja żądania o nazwie get() inicjuje wykonanie metody HTTP GET i odbiera odpowiedź serwera. Jak widać, kod jest niezwykle prosty i zwięzły,
- Jedyne dane, które musimy podać to adres serwera i numer portu usługi – tak jak zrobiliśmy to korzystając z linii adresowej przeglądarki. Uwaga: numer portu można pominąć, jeśli jest równy 80, czyli domyślnemu portowi HTTP.
- Jak widać, funkcja get() zwraca wynik. Jest to obiekt zawierający wszystkie informacje opisujące wykonanie metody GET.
- Oczywiście najważniejszą rzeczą, którą musimy wiedzieć, jest to, czy metoda GET się powiodła. Dlatego korzystamy z właściwości status_code – zawiera ona wystandaryzowaną liczbę opisującą odpowiedź serwera.
- Zgodnie z definicją protokołu HTTP kod 200 oznacza że wszystko jest ok.
- Wszystkie kody odpowiedzi używane przez HTTP są zebrane tutaj: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

```
import requests

reply = requests.get('http://localhost:3000')
print(reply.status_code)
```

HTTP – requests

- Moduł wniosków oferuje wiele różnych sposobów określania i rozpoznawania kodów statusu.
- Zrzuca zawartość słownika statusu. Wynik jest bardzo długi i pogmatwany można używać np. codu

  if reply.status_code == requests.codes.ok:

- Odpowiedź serwera składa się z dwóch części: nagłówka i treści. Obie części mają swoją reprezentację w obiekcie zwróconym przez funkcję get().
- Nagłówek odpowiedzi jest przechowywany we właściwości o nazwie headers (jest to słownik).
- Jak widać, nagłówek odpowiedzi składa się z wielu pól z powiązanymi wartościami (tak naprawdę każde pole zajmuje jeden wiersz odpowiedzi).
- Większość z nich nas nie interesuje, choć niektóre są kluczowe, np. Content-Type, który opisuje, jaka jest naprawdę treść odpowiedzi serwera.
- Surowa treść odpowiedzi jest przechowywana przez właściwość tekstu
- Właściwość zawiera czysty tekst pobrany bezpośrednio ze strumienia danych, dlatego jest to tylko ciąg znaków. Nie są stosowane żadne konwersje.
- gdy zarówno klient, jak i serwer są świadomi faktu, że treść zawiera wiadomość JSON, możliwe jest również użycie metody o nazwie json(), która zwraca dokładnie to, czego możemy się spodziewać – słownik lub lista słowników.

```
import requests

reply = requests.get('http://localhost:3000')
print(reply.status_code)
```

```
import requests

reply = requests.get('http://localhost:3000')
print(reply.status_code)

print(requests.codes.__dict__)
```

```
print(reply.headers)
```

```
print(reply.text)
```

HTTP – metody

- GET ma na celu pobranie informacji (zasobu) z serwera; oczywiście prosty serwer oferuje więcej niż jeden zasób, więc metoda GET umożliwia klientowi precyzyjne określenie jego wymagań. Jeśli klient nie ma żadnych żądań i inicjuje GET bez identyfikacji zasobów, odpowiedź serwera będzie zawierała dokument root – dokładnie to widzieliśmy jakiś czas temu, kiedy nasz własny serwer przesłał nam ten prosty tekst CARS DATABASE.
- POST, podobnie jak GET, służy do przesyłania zasobu, ale w przeciwnym kierunku: od klienta do serwera; podobnie jak w GET należy podać identyfikację zasobu (serwer chce wiedzieć jak nazwać otrzymaną informację). Zakłada się również, że zasób, który klient wysyła, jest nowy na serwerze – nie zastępuje ani nie nadpisuje żadnych wcześniej zebranych danych.
- PUT, podobnie jak POST, przekazuje zasób od klienta do serwera, ale intencja jest inna – wysyłany zasób ma zastąpić wcześniej przechowywane dane.
- DELETE – ta nazwa nie pozostawia wątpliwości: służy do nakazania serwerowi usunięcia zasobu z danej identyfikacji; od tego momentu zasób jest niedostępny.
- HEAD - jest identyczna z metodą GET, z tą różnicą, że serwer NIE MOŻE wysyłać treści w odpowiedzi. HEAD służy do uzyskiwania metadanych dotyczących wybranej reprezentacji bez przesyłania danych reprezentacji, często w celu przetestowania linków hipertekstowych lub znalezienia ostatnich modyfikacji.
- CONNECT żąda, aby odbiorca ustanowił tunel do docelowego serwera początkowego określonego przez cel żądania, a następnie, jeśli się powiedzie, ograniczy swoje zachowanie do ślepego przekazywania danych w obu kierunkach, aż do zamknięcia tunelu. Tunele są powszechnie używane do tworzenia kompleksowych połączeń wirtualnych za pośrednictwem jednego lub większej liczby serwerów proxy, które można następnie zabezpieczyć za pomocą protokołu TLS (Transport Layer Security).
- OPTIONS żąda informacji o opcjach komunikacji dostępnych dla zasobu docelowego na serwerze źródłowym lub pośredniku. Ta metoda umożliwia klientowi określenie opcji i/lub wymagań związanych z zasobem lub możliwościami serwera bez implikowania działania związanego z zasobami.
- TRACE żąda zdalnego zapętlenia komunikatu żądania na poziomie aplikacji. Ostateczny odbiorca żądania POWINIEN przekazać klientowi otrzymaną wiadomość, z wyłączeniem niektórych pól opisanych poniżej, jako treść odpowiedzi 200 (OK).

HTTP – request – try except

- Jak zapewne się domyślasz, wszystkie wymienione metody HTTP mają swoje odbicia (a raczej rodzeństwo) w ramach modułu request.
- Diagram, choć niezwykle uproszczony i pozbawiony istotnych szczegółów, pokazuje nam najważniejsze narzędzia, których będziemy używać wkrótce do gry na naszym serwerze.
- Wszystkie funkcje żądań mają zwyczaj zgłaszania wyjątku, gdy napotkają jakikolwiek problem z komunikacją, chociaż niektóre problemy wydają się być bardziej powszechne niż inne.
- To normalne, że serwer nie odpowiada natychmiast – nawiązywanie połączeń, przesyłanie danych, wyszukiwanie zasobów – wszystkie te kroki wymagają czasu.
- Jak widać, funkcja get() uwzględnia jeden dodatkowy argument o przekroczeniu czasu – jest to maksymalny czas (mierzony w sekundach i wystawiony na działanie rzeczywistej) jaki dopuszcza się oczekiwanie na reakcję. Jeśli czas zostanie przekroczony, get() zgłosi wyjątek o nazwie request.exceptions.Timeout.

```
import requests

try:
    reply = requests.get('http://localhost:3000', timeout=1)
except requests.exceptions.Timeout:
    print('Sorry, Mr. Impatient, you didn\'t get your data')
else:
    print('Here is your data, my Master!')
```

```
import requests

try:
    reply = requests.get('http://localhost:3001', timeout=1)
except requests.exceptions.ConnectionError:
    print('Nobody\'s home, sorry!')
else:
    print('Everything fine!')
```

HTTP – niepoprawny url

- Przy niepoprawny adresie url występuje błąd URL

```
RequestException
|___HTTPError
|___ConnectionError
|    |___ProxyError
|    |___SSLError
|___Timeout
|    |___ConnectTimeout
|    |___ReadTimeout
|___URLRequired
|___TooManyRedirects
|___MissingSchema
|___InvalidSchema
|___InvalidURL
|    |___InvalidProxyURL
|___InvalidHeader
|___ChunkedEncodingError
|___ContentDecodingError
|___StreamConsumedError
|___RetryError
|___UnrewindableBodyError
```

```python
import requests

try:
    reply = requests.get('http://////////')
except requests.exceptions.InvalidURL:
    print('Recipient unknown!')
else:
    print('Everything fine!')
```

HTTP – CRUD

- Create - Jeśli potrafisz „tworzyć”, możesz dodawać nowe elementy do zbierania danych, na przykład napisać nowy post na blogu, dodać nowe zdjęcie do galerii, przechowywać dane nowego klienta w bazie danych klientów itp.Na poziomie REST tworzenie nowych elementów realizowane jest metodą POST HTTP.
- Read - Odczyt/pobieranie to bardzo podstawowa umiejętność przeglądania danych zgromadzonych w zbiorze, np. czytanie wpisów na czyimś blogu, przeglądanie zdjęć w galerii, badanie historii klientów w bazie danych itp.Na poziomie REST pobieranie elementów realizowane jest metodą GET HTTP.
- Update - Aktualizujesz dane w kolekcji, gdy modyfikujesz zawartość wybranej pozycji bez jej usuwania, np. edytujesz swój wpis na blogu, zmieniasz rozmiar zdjęcia w galerii, wprowadzasz informacje o sprzedaży aktualnego klienta itp.Na poziomie REST aktualizacja istniejących danych realizowana jest metodą PUT HTTP.
- Delete - Usunięcie ma miejsce, gdy usuniesz swój post z bloga, usuniesz zdjęcie z galerii lub usuniesz konto klienta.Na poziomie REST usuwanie istniejących danych realizowane jest metodą DELETE HTTP.

HTTP – cars servers

JSON wykorzystamy go jako język pośredni do komunikacji z serwerem HTTP, implementacji CRUD i przechowywania przykładowego zbioru danych.

Przyjmijmy założenia:

- skorzystamy z zaprezentowanego wcześniej json-server
- nasza początkowa baza danych, przetwarzana na nasze żądanie przez serwer json, będzie zbiorem samochodów retro zapisanych w pliku cars.json
- json-server wczyta plik i obsłuży jego zawartość zgodnie z naszymi działaniami;
- każdy samochód jest opisany przez:
    - id – unikalny numer towaru;
    - note – każdy element w kolekcji musi mieć właściwość o tej nazwie – w ten sposób serwer identyfikuje każdy element i odróżnia elementy od siebie;
    - brand – string
    - Model – string
    - production_year – intiger
    - Convertible - Bool
- plik początkowy zawiera dane dla sześciu samochodów
- fakt, że json-serwer obsługuje dane pierwotnie zakodowane jako JSON, nie ma absolutnie nic wspólnego z tym, że będziemy przekazywać komunikaty JSON pomiędzy klientem (naszym kodem) a serwerem (json-serwer).
- Sposób, w jaki serwer jest używany do inicjowania i przechowywania danych, jest dla nas właściwie czarną skrzynką (chyba że wdrażamy sam serwer). Różne serwery mogą wykorzystywać różne środki

HTTP – CRUD - Read

- son-server zakłada, że kolekcja danych dziedziczy swoją nazwę po nazwie źródłowego pliku danych. Ponieważ nazwaliśmy plik samochody, serwer opublikuje również dane jako samochody. Musisz użyć nazwy w URI, chyba że chcesz otrzymać dokument domyślny (root), który jest dla nas całkowicie bezużyteczny.
- linia 1: importujemy moduł requestów;
- linia 3: będziemy łączyć się z serwerem wewnątrz bloku try – to pozwoli nam zabezpieczyć się przed możliwymi skutkami problemów z połączeniem;
- linia 4: tworzymy żądanie GET i kierujemy je do zasobu o nazwie cars znajdującego się na serwerze pracującym pod adresem określonym jako localhost, nasłuchującym na porcie numer 300;
- linia 5: czy nam się udało?
- linia 6: niestety ponieśliśmy porażkę; wygląda na to, że serwer nie działa lub jest niedostępny;
- linia 8: dobra wiadomość – serwer odpowiedział! Sprawdźmy kod statusu;
- linia 9: drukujemy dane, które przesłał nam serwer (raczej nudne); treść odpowiedzi jest przechowywana jako właściwość tekstowa obiektu odpowiedzi;
- linia 11: zła wiadomość – serwer nie lubi ani nas, ani naszej prośby.

```
import requests

try:
    reply = requests.get("http://localhost:3000/cars")
except requests.RequestException:
    print("Communication error")
else:
    if reply.status_code == requests.codes.ok:
        print(reply.text)
    else:
        print("Server error")
```

HTTP – CRUD - Read

- Serwer HTTP jest w stanie przesłać praktycznie każdy rodzaj danych: tekst, obraz, wideo, dźwięk i wiele innych.
- Nagłówek odpowiedzi serwera zawiera pole o nazwie Content-Type. Wartość pola jest analizowana przez moduł requestów i jeśli jego wartość zapowiada JSON, to metoda o nazwie json() zwraca ciąg znaków zawierający odebraną wiadomość.
- Linia 9: drukujemy wartość pola Content-Type;
- Linia 10: drukujemy tekst zwrócony przez metodę json().
- Zwróć uwagę na linię rozpoczynającą się od application/json – jest to przesłanka wykorzystywana przez moduł request do diagnozowania treści odpowiedzi.

```
import requests

try:
    reply = requests.get("http://localhost:3000/cars")
except:
    print("Communication error")
else:
    if reply.status_code == requests.codes.ok:
        print(reply.headers['Content-Type'])
        print(reply.json())
    else:
        print("Server error")
```

HTTP – CRUD - Read

- Linia 3: zebraliśmy wszystkie nazwy właściwości w jednym miejscu – użyjemy ich do wyszukiwania danych JSON i wydrukowania pięknej linii nagłówka nad tabelą;
- Linia 4: są to szerokości zajmowane przez właściwości;
- Linia 7: użyjemy tej funkcji do wydrukowania nagłówka tabeli;
- Linia 8: iterujemy przez nazwy_kluczy i szerokości_kluczy połączone razem przez funkcję zip();
- Linia 9: drukujemy nazwę każdej właściwości rozszerzoną do żądanej długości i umieszczamy kreskę na końcu;
- Linia 10: czas uzupełnić linię nagłówka;
- Linia 13: użyjemy tej funkcji do wydrukowania jednej linii wypełnionej danymi każdego samochodu;
- Linia 14: iteracja jest dokładnie taka sama jak w showhead(), ale...
- Linia 15: ...drukujemy wybraną wartość właściwości zamiast tytułu kolumny;
- Linia 19: użyjemy tej funkcji do wydrukowania zawartości wiadomości JSON jako listy pozycji;
- Linia 20: zaprezentujemy użytkownikowi uroczą tabelkę z nagłówkiem...
- Linia 21 i 19: ...oraz zestaw danych wszystkich samochodów z listy, po jednym samochodzie w wierszu;
- Linia 31: korzystamy tutaj z naszego nowego kodu.

```
import requests

key_names = ["id", "brand", "model",
"production_year", "convertible"]
kay_widths = [10, 15, 10, 20, 15]


def show_head():
    for (n, w) in zip(key_names, kay_widths):
        print(n.ljust(w), end='| ')
    print()


def show_car(car):
    for (n, w) in zip(key_names, kay_widths):
        print(str(car[n]).ljust(w), end='| ')
    print()
```

```
def show(json):
    show_head()
    for car in json:
        show_car(car)


try:
    reply =
requests.get('http://localhost:3000/cars')
except requests.RequestException:
    print('Communication error')
else:
    if reply.status_code == requests.codes.ok:
        show(reply.json())
    else:
        print('Server error')
```

HTTP – CRUD - Read

- Jeśli nie potrzebujesz całej zawartości zasobu, możesz przygotować konkretne żądanie GET, które wymaga tylko jednego elementu i używa identyfikatora jako klucza. Wtedy URI wygląda następująco

`http://server:port/resource/id`

- Linie od 26 do 34: musimy być przygotowani na to, że serwer nie wyśle listy przedmiotów, jeśli o nią poprosimy.
- Linie od 44 do 45: jeśli nie ma elementu o żądanym id, serwer ustawi kod statusu na 404 („nie znaleziono”).
- Wynik wyświetlenia będzie:

```
fid     | brand       | model     | production_year     | convertible    |
2       | Chevrolet   | Camaro    | 1988                | True           |
```

```python
import requests

key_names = ["id", "brand", "model",
"production_year", "convertible"]
key_widths = [10, 15, 10, 20, 15]


def show_head():
    for (n, w) in zip(key_names, key_widths):
        print(n.ljust(w), end='| ')
    print()


def show_empty():
    for w in key_widths:
        print(' '.ljust(w), end='| ')
    print()


def show_car(car):
    for (n, w) in zip(key_names, key_widths):
        print(str(car[n]).ljust(w), end='| ')
    print()
```

```python
def show(json):
    show_head()
    if type(json) is list:
        for car in json:
            show_car(car)
    elif type(json) is dict:
        if json:
            show_car(json)
        else:
            show_empty()


try:
    reply =
requests.get('http://localhost:3000/cars/2')
except requests.RequestException:
    print('Communication error')
else:
    if reply.status_code == requests.codes.ok:
        show(reply.json())
    elif reply.status_code ==
requests.codes.not_found:
        print("Resource not found")
    else:
        print('Server error')
```

HTTP – CRUD - Read

- Konkretny serwer może zapewniać dodatkowe udogodnienia, np. może manipulować danymi przed wysłaniem ich do klienta.
- Serwer json jest w stanie sortować elementy, używając dowolnej właściwości jako klucza sortowania (domyślnie sortuje elementy według ich identyfikatorów).
- Zwykle URI załatwia sprawę, ale pamiętaj, że nie ma wspólnego standardu obejmującego takie dodatkowe funkcje – więcej informacji znajdziesz w dokumentacji serwera.
- Serwer json zakłada, że identyfikator URI jest utworzony w następujący sposób:

http://server:port/resource?_sort=property

- powoduje, że odpowiedź jest sortowana w porządku rosnącym przy użyciu właściwości o nazwie prop. Znak ? – oddziela identyfikację zasobu od dodatkowych parametrów żądania.

id	brand	model	production_year	convertible
6	Mercedes Benz	300SL	1967	True
4	Maserati	Mexico	1970	False
1	Ford	Mustang	1972	False
5	Nissan	Fairlady	1974	False
2	Chevrolet	Camaro	1988	True
3	Aston Martin	Rapide	2010	False

```
import requests

key_names = ["id", "brand", "model",
"production_year", "convertible"]
key_widths = [10, 15, 10, 20, 15]


def show_head():
    for (n, w) in zip(key_names,
key_widths):
        print(n.ljust(w), end='| ')
    print()


def show_empty():
    for w in key_widths:
        print(' '.ljust(w), end='| ')
    print()


def show_car(car):
    for (n, w) in zip(key_names,
key_widths):
        print(str(car[n]).ljust(w), end='| ')
    print()
```

```
def show(json):
    show_head()
    if type(json) is list:
        for car in json:
            show_car(car)
    elif type(json) is dict:
        if json:
            show_car(json)
        else:
            show_empty()


try:
    reply =
requests.get('http://localhost:3000/cars?_sort=
production_year')
except requests.RequestException:
    print('Communication error')
else:
    if reply.status_code == requests.codes.ok:
        show(reply.json())
    elif reply.status_code ==
requests.codes.not_found:
        print("Resource not found")
    else:
        print('Server error')
```

HTTP – CRUD - Read

- Json-server jest również w stanie odwrócić kolejność sortowania – wystarczy przepisać URI w następujący sposób:
- `http://server:port/resource?_sort=property&_order=desc`
- Zwróć uwagę na znak & – oddziela on od siebie dodatkowe parametry żądania.

```
id | brand        | model    | production_year | convertible |
3  | Aston Martin | Rapide   | 2010            | False       |
2  | Chevrolet    | Camaro   | 1988            | True        |
5  | Nissan       | Fairlady | 1974            | False       |
1  | Ford         | Mustang  | 1972            | False       |
4  | Maserati     | Mexico   | 1970            | False       |
6  | Mercedes Benz| 300SL    | 1967            | True        |
```

- Niektóre serwery potrafią znacznie więcej, np. mogą przeprowadzać wyszukiwanie pełnotekstowe, tworzyć wycinki lub analizować złożone wyrażenia filtrujące.

```
try:
    reply =
requests.get('http://localhost:3000/cars?_sort=production_year&_
order=desc')
except requests.RequestException:
    print('Communication error')
else:
    if reply.status_code == requests.codes.ok:
        show(reply.json())
    elif reply.status_code == requests.codes.not_found:
        print("Resource not found")
    else:
        print('Server error')
```

HTTP – CRUD - Read

- Domyślnie serwer implementujący protokół HTTP w wersji 1.1 działa w następujący sposób:
- oczekuje na połączenie klienta;
- odczytuje żądanie klienta;
- wysyła swoją odpowiedź;
- utrzymuje połączenie przy życiu, czekając na kolejne żądanie klienta;
- jeśli klient jest nieaktywny przez jakiś czas, serwer po cichu zamyka połączenie;
- oznacza to, że klient jest zobowiązany do ponownego ustanowienia nowego połączenia, jeśli chce wysłać kolejne żądanie.
- Serwer informuje klienta, czy połączenie zostało utrzymane, czy też nie, za pomocą pola o nazwie Connection, umieszczonego w nagłówku odpowiedzi.
- W przykładzie program wyświetla: Connection=keep-alive
- close oznacza, że serwer zamknie połączenie, gdy tylko odpowiedź zostanie w pełni przesłana
- Jeśli klient wie, że przez jakiś czas nie będzie przeszkadzał serwerowi kolejnymi żądaniami, może skłonić serwer do tymczasowej zmiany przyzwyczajeń i natychmiastowego zamknięcia połączenia. Pozwoli to zaoszczędzić zasoby serwera.

```
try:
    reply = requests.get('http://localhost:3000/cars')
except requests.RequestException:
    print('Communication error')
else:
    print('Connection=' + reply.headers['Connection'])
    if reply.status_code == requests.codes.ok:
        show(reply.json())
    elif reply.status_code == requests.codes.not_found:
        print("Resource not found")
    else:
        print('Server error')
```

```
Connection=keep-alive
id        | brand          | model     | production_year     | convertible    |
1         | Ford           | Mustang   | 1972                | False          |
2         | Chevrolet      | Camaro    | 1988                | True           |
3         | Aston Martin   | Rapide    | 2010                | False          |
4         | Maserati       | Mexico    | 1970                | False          |
5         | Nissan         | Fairlady  | 1974                | False          |
6         | Mercedes Benz  | 300SL     | 1967                | True           |
```

HTTP – CRUD - Delete

- metoda DELETE, jest obsługiwana przez funkcję delete(). Usuniemy zamochów z naszego serwera.
- Ponadto zrobimy coś więcej – podzielimy naszą akcję na dwa etapy
- 1. poprosimy serwer o usunięcie jednego samochodu o podanym id wiedząc, że serwer utrzyma połączenie;
- 2. poprosimy serwer o przedstawienie aktualnej zawartości oferty z sugestią zamknięcia połączenia natychmiast po transmisji.
- Przeanalizujmy kod w edytorze:
- Linia 37: przygotowujemy własny nagłówek żądania, który uzupełni domyślny, wysyłany po cichu wraz z każdym żądaniem – jest to słownik z kluczem Connection (to ta sama nazwa, co wysyłany przez serwer) i zestawem wartości Close;
- Linia 39: używamy delete() – zanotuj URI, który opisuje element do usunięcia;
- Linia 40: drukujemy kod statusu serwera;
- Linia 41: prosimy serwer o pokazanie nam pełnej listy samochodów, ale wysyłamy również naszą prośbę o zamknięcie połączenia – odbywa się to poprzez ustawienie parametru o nazwie headers;
- Linia 46: chcielibyśmy sprawdzić, czy serwer uhonorował naszą rekomendację.

```
try:
    reply = requests.get('http://localhost:3000/cars')
except requests.RequestException:
    print('Communication error')
else:
    print('Connection=' + reply.headers['Connection'])
    if reply.status_code == requests.codes.ok:
        show(reply.json())
    elif reply.status_code == requests.codes.not_found:
        print("Resource not found")
    else:
        print('Server error')
```

```
res=200
Connection=close
id        | brand          | model     | production_year     | convertible    |
2         | Chevrolet      | Camaro    | 1988                | True           |
3         | Aston Martin   | Rapide    | 2010                | False          |
4         | Maserati       | Mexico    | 1970                | False          |
5         | Nissan         | Fairlady  | 1974                | False          |
6         | Mercedes Benz  | 300SL     | 1967                | True           |
```

HTTP – CRUD - Create

- Dodajmy do naszej oferty nowy samochód. Do tego wykorzystamy metodę create
- Nowe kroki są zakodowane w linii 1 i liniach od 21 do 53
- Linia 1: dodajemy json do listy importu – będzie nam potrzebny do stworzenia tekstowej reprezentacji nowego przedmiotu/samochodu;
- Linia 39: jeśli mamy coś wysłać na serwer, serwer musi wiedzieć, co to właściwie jest; jak już wiesz, serwer informuje nas o rodzaju treści za pomocą pola Content-Type; możemy użyć tej samej techniki, aby ostrzec serwer, że wysyłamy coś więcej niż zwykłe żądanie. Dlatego przygotowujemy nasze pole Content-Type z odpowiednią wartością;
- Linia 40: To nasz nowy samochód. Przygotowaliśmy wszystkie potrzebne dane i spakowaliśmy je do słownika Pythona – oczywiście przekonwertujemy je na JSON, zanim wyślemy je w świat;
- Linia 45: chcemy sprawdzić, jak wygląda wynikowy komunikat JSON
- Linia 47: tu dzieją się najważniejsze rzeczy – wywołujemy funkcję post() (zwróć uwagę na URI – wskazuje tylko na zasób, a nie na konkretny element) i ustawiamy dwa dodatkowe parametry: jeden (nagłówki) jako uzupełnienie żądania nagłówek z polem Content-Type, a drugi (data) do przekazania komunikatu JSON do żądania.

```
h_close = {'Connection': 'Close'}
h_content = {'Content-Type': 'application/json'}
new_car = {'id': 1,
           'brand': 'Porsche',
           'model': '911',
           'production_year': 1963,
           'convertible': False}
print(json.dumps(new_car))
try:
    reply = requests.post('http://localhost:3000/cars', headers=h_content,
data=json.dumps(new_car))
    print("reply=" + str(reply.status_code))
    reply = requests.get('http://localhost:3000/cars/', headers=h_close)
except requests.RequestException:
    print('Communication error')
else:
    print('Connection=' + reply.headers['Connection'])
    if reply.status_code == requests.codes.ok:
        show(reply.json())
    elif reply.status_code == requests.codes.not_found:
        print("Resource not found")
    else:
        print('Server error')
```

```
{"id": 7, "brand": "Porsche", "model": "911", "production_year": 1963, "convertible": false}
reply=201
Connection=close
id        | brand          | model     | production_year     | convertible    |
2         | Chevrolet      | Camaro    | 1988                | True           |
3         | Aston Martin   | Rapide    | 2010                | False          |
4         | Maserati       | Mexico    | 1970                | False          |
5         | Nissan         | Fairlady  | 1974                | False          |
6         | Mercedes Benz  | 300SL     | 1967                | True           |
7         | Porsche        | 911       | 1963                | False          |
```

HTTP – CRUD - Update

- Ostatnia pozostała litera to Update, więc teraz zaktualizujemy jeden z istniejących elementów. Aktualizowanie elementu jest właściwie podobne do dodawania.
- Linia 33: nagłówek jest taki sam jak poprzednio, ponieważ wyślemy json na serwer;
- Linie od 34 do 38: to są nowe dane dla pozycji o id równym 6; uwaga – zaktualizowaliśmy rok produkcji (powinien być 1957 zamiast 1967)
- Linia 40: teraz wywołujemy funkcję put(); uwaga – musimy zrobić URI, który jednoznacznie wskazuje modyfikowany element; ponadto musimy wysłać cały przedmiot, a nie tylko zmienioną właściwość.

```
h_close = {'Connection': 'Close'}
h_content = {'Content-Type': 'application/json'}
car = {'id': 6,
       'brand': 'Mercedes Benz',
       'model': '300SL',
       'production_year': 1957,
       'convertible': True}
try:
    reply = requests.put('http://localhost:3000/cars/6',
headers=h_content, data=json.dumps(car))
    print("res=" + str(reply.status_code))
    reply = requests.get('http://localhost:3000/cars/',
headers=h_close)
except requests.RequestException:
    print('Communication error')
else:
    print('Connection=' + reply.headers['Connection'])
    if reply.status_code == requests.codes.ok:
        show(reply.json())
    elif reply.status_code == requests.codes.not_found:
        print("Resource not found")
    else:
        print('Server error')
```

```
res=200
Connection=close
id        | brand          | model     | production_year     | convertible    |
2         | Chevrolet      | Camaro    | 1988                | True           |
3         | Aston Martin   | Rapide    | 2010                | False          |
4         | Maserati       | Mexico    | 1970                | False          |
5         | Nissan         | Fairlady  | 1974                | False          |
6         | Mercedes Benz  | 300SL     | 1957                | True           |
7         | Porsche        | 911       | 1963                | False          |
```

Zadanie 8b - Sprawdzanie dostępności serwera HTTP

Napisz proste narzędzie CLI (Command Line Interface), za pomocą którego można zdiagnozować aktualny stan konkretnego serwera http. Narzędzie powinno akceptować jeden lub dwa argumenty wiersza poleceń

- (obowiązkowo) adres (IP lub kwalifikowana nazwa domeny) serwera do diagnozy (diagnoza będzie niezwykle prosta, chcemy tylko wiedzieć, czy serwer jest martwy, czy żywy)
- (opcjonalnie) numer portu serwera (brak argumentu oznacza, że narzędzie powinno używać portu 80)
- użyj metody HEAD zamiast GET — zmusza serwer do wysłania pełnego nagłówka odpowiedzi, ale bez treści; wystarczy sprawdzić, czy serwer działa poprawnie; reszta żądania pozostaje taka sama jak w przypadku GET.

Zakładamy również, że:

- narzędzie sprawdza poprawność wywołania, aw przypadku braku argumentów w wywołaniu wypisuje komunikat o błędzie i zwraca kod wyjścia równy 1;
- jeśli w linii wywołania są dwa argumenty, a drugi nie jest liczbą całkowitą z zakresu 1..65535, narzędzie wypisze komunikat o błędzie i zwróci kod wyjścia równy 2;
- jeśli narzędzie przekroczy limit czasu podczas łączenia, drukowany jest komunikat o błędzie i zwracane jest 3 jako kod wyjścia;
- jeśli połączenie nie powiedzie się z jakiegokolwiek innego powodu, pojawi się komunikat o błędzie i zwrócona zostanie wartość 4 jako kod wyjścia;
- jeśli połączenie się powiedzie, drukowany jest pierwszy wiersz odpowiedzi serwera.

Poradnik:

- aby uzyskać dostęp do argumentów wiersza poleceń, użyj zmiennej argv z modułu sys; jego długość jest zawsze o jeden większa niż rzeczywista liczba argumentów, ponieważ argv[0] przechowuje nazwę twojego skryptu; oznacza to, że pierwszy argument znajduje się w argv[1], a drugi w argv[2]; nie zapominaj, że argumenty wiersza poleceń są zawsze ciągami znaków
- zwrócenie kodu wyjścia równego n można uzyskać, wywołując funkcję exit(n).

Zadanie 8b - Sprawdzanie dostępności serwera HTTP

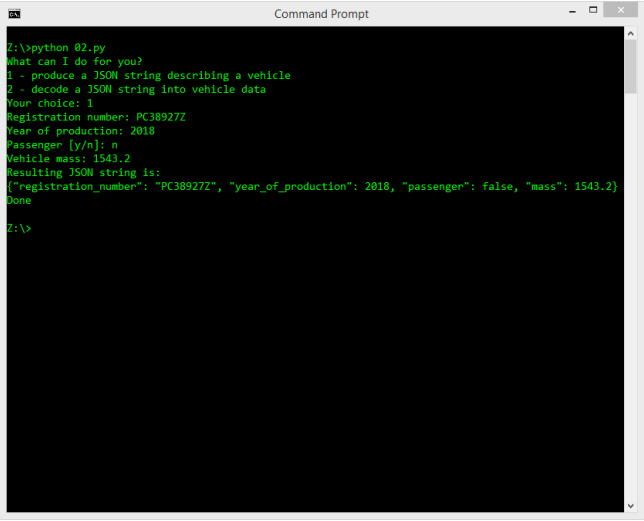
Zaimplementować dokładnie tę samą funkcjonalność, jak w swoim kodzie poprzednio 8a, ale tym razem musisz użyć modułu requestów zamiast modułu socket. Wszystko inne powinno pozostać takie samo: argumenty wiersza poleceń i dane wyjściowe muszą być nie do odróżnienia.

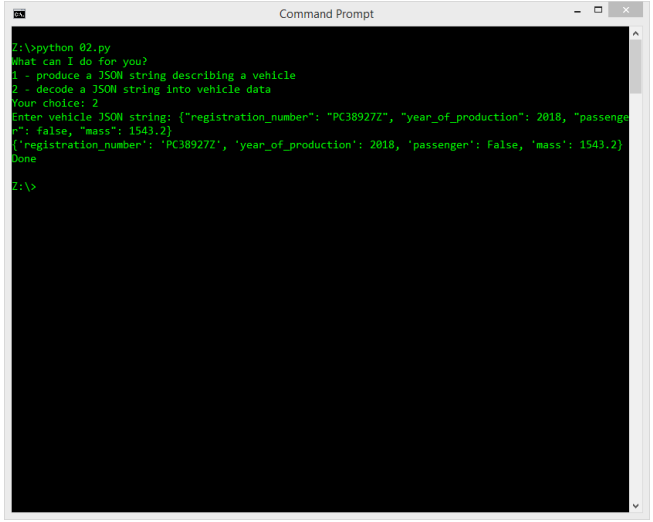
Wskazówka:

- użyj metody head() zamiast get(), ponieważ nie potrzebujesz całego dokumentu głównego wysyłanego przez serwer — wystarczy nagłówek, aby sprawdzić, czy serwer odpowiada. Na szczęście head() ma dokładnie taki sam interfejs jak get().
- Nie zapomnij obsłużyć wszystkich potrzebnych wyjątków — nie pozostawiaj użytkownika bez jasnych wyjaśnień, co poszło nie tak.

Zadanie 9 –kodowanie i dekodowanie danych

Spójrz na te dwa zrzuty ekranu. Przedstawiają dwa różne przypadki użycia tego samego programu





Twoim zadaniem jest napisanie kodu, który prowadzi dokładnie taką samą rozmowę z użytkownikiem oraz:

definiuje klasę o nazwie Vehicle, której obiekty mogą przenosić dane pojazdu pokazane powyżej (strukturę klasy należy wydedukować z powyższego okna dialogowego — nazwijmy to „inżynierią wsteczną”)

- definiuje klasę zdolną do zakodowania obiektu Vehicle w równoważny ciąg JSON;
- definiuje klasę zdolną do zdekodowania łańcucha JSON do nowo utworzonego obiektu Vehicle.

Oczywiście należy również przeprowadzić pewne podstawowe kontrole poprawności danych. Jesteśmy pewni, że jesteś wystarczająco ostrożny, aby chronić swój kod przed lekkomyślnymi użytkownikami.

Zadanie 10 – wszystko razem

Napisz aplikację w pythonie wykorzystującą bazę danych cars.json która będzie w stanie w łatwy sposób:

- Obsługiwać CRUD w celu zmiany wpisów samochodów
- Wyświetlała listę samochodów
- Szukała samochodu po określonym parametrze

5 Przetwarzanie plików

File processing

File Processing – nazwy plików

- Różne systemy operacyjne mogą traktować pliki na różne sposoby. Na przykład Windows używa innej konwencji ścieżek niż ta przyjęta w systemach Unix/Linux
- Jak widać, systemy wywodzące się z Unix/Linux nie używają litery napędu dysku (np. C:) i wszystkie katalogi wyrastają z jednego katalogu głównego o nazwie /, podczas gdy systemy Windows rozpoznają katalog główny jako \.
- Ponadto w nazwach plików systemowych Unix/Linux rozróżniana jest wielkość liter. Systemy Windows przechowują wielkość liter używanych w nazwach plików, ale w ogóle nie rozróżniają ich wielkości.
- W pythonie nazwy plików muszą być zapisywane następująco:

```
name = "\\dir\\file"
```

A pod linuxem (windows jest również w stanie obsłużyć te ścieżki)

```
name = "/dir/file" name = "c:/dir/file "
```

- Każdy program napisany w Pythonie komunikuje się z plikami za pomocą abstrakcyjnych uchwytów (handlers) albo strumieniów (streams)
- Operacja połączenia strumienia z plikiem nazywa się otwarciem pliku (open), natomiast rozłączenie tego połączenia nazywamy zamknięciem pliku (close).

Windows

```
C:\directory\file
```

Linux

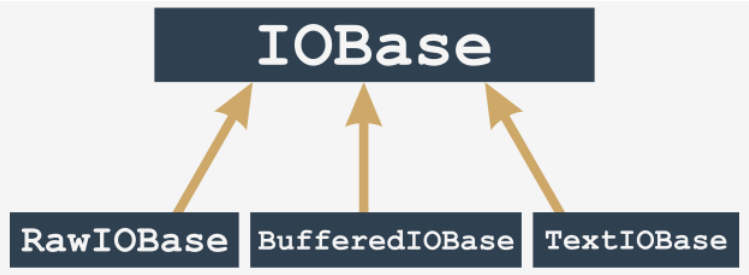
```
/directory/files
```

File Processing – strumień pliku

- Otwarcie strumienia jest nie tylko związane z plikiem, ale powinno również zadeklarować sposób, w jaki strumień zostanie przetworzony. Deklaracja ta nazywana jest trybem otwartym.
- Jeśli otwarcie się powiedzie, program będzie mógł wykonać tylko te operacje, które są zgodne z zadeklarowanym trybem otwarcia.
- Na strumieniu wykonywane są dwie podstawowe operacje:
    - czytać ze strumienia: części danych są pobierane z pliku i umieszczane w obszarze pamięci zarządzanym przez program (np. zmienna);
    - zapisuj do strumienia: części danych z pamięci (np. zmienna) są przenoszone do pliku.
- Istnieją trzy podstawowe tryby otwierania strumienia:
    - tryb odczytu: strumień otwarty w tym trybie umożliwia tylko operacje odczytu; próba zapisu do strumienia spowoduje wyjątek (wyjątek nosi nazwę UnsupportedOperation, który dziedziczy OSError i ValueError i pochodzi z modułu io);
    - tryb zapisu: strumień otwarty w tym trybie umożliwia tylko operacje zapisu; próba odczytania strumienia spowoduje wspomniany wyżej wyjątek;
    - tryb aktualizacji: strumień otwarty w tym trybie umożliwia zarówno zapis, jak i odczyt.

File Processing – uchwyt plików

- Python zakłada, że każdy plik jest ukryty za obiektem odpowiedniej klasy.
- Pliki można przetwarzać na wiele różnych sposobów - niektóre z nich zależą od zawartości pliku.
- Obiekt odpowiedniej klasy powstaje, gdy otwierasz plik i unicestwiasz go w momencie zamykania.
- Pomiędzy tymi dwoma zdarzeniami można użyć obiektu do określenia, jakie operacje mają zostać wykonane na określonym strumieniu. Operacje, których możesz użyć, są narzucone przez sposób, w jaki otworzyłeś plik.



- Aby otworzyć obiekt należy wywołać funkcję o nazwie open()
- Funkcja analizuje podane argumenty i automatycznie tworzy wymagany obiekt.
- Jeśli chcesz pozbyć się obiektu, wywołujesz metodę o nazwie close(). Wywołanie zerwie połączenie z obiektem i plikiem oraz usunie obiekt.
- Ze względu na rodzaj zawartości strumienia wszystkie strumienie dzielą się na strumienie tekstowe i binarne.
- Strumienie tekstu mają strukturę wierszy; to znaczy zawierają znaki typograficzne (litery, cyfry, znaki interpunkcyjne itp.) ułożone w rzędy (wiersze), co widać gołym okiem, gdy patrzy się na zawartość pliku w edytorze.
- Ten plik jest zapisywany (lub odczytywany) głównie znak po znaku lub wiersz po wierszu.

File Processing – uchwyt plików

- Strumienie binarne nie zawierają tekstu, ale sekwencję bajtów o dowolnej wartości. Ta sekwencja może być na przykład programem wykonywalnym, obrazem, plikiem audio lub wideo, plikiem bazy danych itp.
- Ponieważ te pliki nie zawierają wierszy, odczyty i zapisy odnoszą się do porcji danych o dowolnym rozmiarze. W związku z tym dane są odczytywane/zapisywane bajt po bajcie lub blok po bloku, gdzie rozmiar bloku zwykle waha się od jednego do dowolnie wybranej wartości.
- W systemach Unix/Linux końce linii są oznaczane pojedynczym znakiem o nazwie LF (kod ASCII 10) oznaczanym w programach Pythona jako \n.
- Inne systemy operacyjne, zwłaszcza wywodzące się z prehistorycznego systemu CP/M (dotyczy to również systemów z rodziny Windows), stosują inną konwencję: koniec linii oznaczany jest parą znaków CR i LF (kody ASCII 13 i 10), które można zakodować jako \r\n.
- Jeśli tworzysz program odpowiedzialny za przetwarzanie pliku tekstowego i jest on napisany dla systemu Windows, możesz rozpoznać końce linii po znalezieniu znaków \r\n, ale ten sam program działający w środowisku Unix/Linux będzie całkowicie bezużyteczny i vice versa: program napisany dla systemów Unix/Linux może być bezużyteczny w Windows.
- Takie niepożądane cechy programu, które uniemożliwiają lub utrudniają korzystanie z programu w różnych środowiskach, nazywane są nieprzenośnością.
- Podobnie cecha programu pozwalająca na działanie w różnych środowiskach nazywana jest przenośnością. Program wyposażony w taką cechę nazywany jest programem przenośnym.

File Processing – uchwyt plików

Obsługa tego problemu odbyło się na poziomie klasy, odbyło się to na poziomie klas, które odpowiadają za wczytywanie i zapisywanie znaków do i ze strumienia. Działa to w następujący sposób:

- gdy strumień jest otwarty i zalecono, aby dane w powiązanym pliku były przetwarzane jako tekst (lub w ogóle nie ma takiej porady), następuje przełączenie w tryb tekstowy;
- podczas odczytu/zapisu wierszy z/do powiązanego pliku w środowisku Unix nie dzieje się nic szczególnego, natomiast przy wykonywaniu tych samych operacji w środowisku Windows zachodzi proces zwany translacją znaków nowej linii: gdy czytasz wiersz z plik, każda para znaków \r\n jest zastępowana pojedynczym znakiem \n i odwrotnie; podczas operacji zapisu każdy \n znak jest zastępowany parą \r\n znaków;
- mechanizm jest całkowicie przezroczysty dla programu, który można napisać tak, jakby był przeznaczony wyłącznie do przetwarzania plików tekstowych Unix/Linux; kod źródłowy uruchamiany w środowisku Windows również będzie działał poprawnie;
- kiedy strumień jest otwarty i jest to zalecane, jego zawartość jest traktowana tak, jak jest, bez żadnej konwersji - żadne bajty nie są dodawane ani pomijane.

Otwarcie strumienia realizowane jest przez funkcję, którą można wywołać w następujący sposób:

```
stream = open(file, mode = 'r', encoding = None)
```

- Funkcja open zwraca obiekt strumienia; w przeciwnym razie zgłaszany jest wyjątek (np. FileNotFoundError, jeśli plik, który zamierzasz przeczytać, nie istnieje);
- pierwszy parametr funkcji (file) określa nazwę pliku, który ma zostać powiązany ze strumieniem;
- drugi parametr (mode) określa tryb otwarty używany dla strumienia; to ciąg znaków wypełniony sekwencją znaków, z których każdy ma swoje specjalne znaczenie (więcej szczegółów wkrótce);
- trzeci parametr (encoding) określa typ kodowania (np. UTF-8 podczas pracy z plikami tekstowymi)
- otwarcie musi być pierwszą operacją wykonaną na strumieniu.

File Processing – Otwieranie strumieni: tryby

- r tryb otwarty: odczyt
  - strumień zostanie otwarty w trybie odczytu;
  - plik powiązany ze strumieniem musi istnieć i musi być czytelny, w przeciwnym razie funkcja open() zgłasza wyjątek
- w tryb otwarty: pisz
  - strumień zostanie otwarty w trybie zapisu;
  - plik powiązany ze strumieniem nie musi istnieć; jeśli nie istnieje, zostanie utworzony; jeśli istnieje, zostanie obcięty do długości zera (usunięty); jeśli utworzenie nie jest możliwe (np. ze względu na uprawnienia systemowe), funkcja open() zgłasza wyjątek.
- tryb otwarty: dołącz
  - strumień zostanie otwarty w trybie dołączania;
  - plik powiązany ze strumieniem nie musi istnieć; jeśli nie istnieje, zostanie utworzony; jeśli istnieje, wirtualna głowica rejestrująca zostanie ustawiona na końcu pliku (poprzednia zawartość pliku pozostaje niezmieniona).
- r+ tryb otwarty: odczyt i aktualizacja
  - strumień zostanie otwarty w trybie odczytu i aktualizacji;
  - plik powiązany ze strumieniem musi istnieć i musi być zapisywalny, w przeciwnym razie funkcja open() zgłosi wyjątek;
  - dla strumienia dozwolone są zarówno operacje odczytu, jak i zapisu.
- tryb otwarty w+: pisz i aktualizuj
  - strumień zostanie otwarty w trybie zapisu i aktualizacji;
  - plik powiązany ze strumieniem nie musi istnieć; jeśli nie istnieje, zostanie utworzony; poprzednia zawartość pliku pozostaje nienaruszona;
  - dla strumienia dozwolone są zarówno operacje odczytu, jak i zapisu.

File Processing – Otwieranie strumieni: tryby

- Jeśli na końcu ciągu trybu znajduje się litera b, oznacza to, że strumień ma zostać otwarty w trybie binarnym.
- Jeśli ciąg trybu kończy się literą t, strumień jest otwierany w trybie tekstowym.
- Tryb tekstowy jest domyślnym zachowaniem przyjmowanym, gdy nie jest używany żaden specyfikator trybu binarnego/tekstowego.
- Na koniec pomyślne otwarcie pliku ustawi bieżącą pozycję pliku (wirtualny nagłówek odczytu/zapisu) przed pierwszym bajtem pliku, jeśli tryb nie jest a, i po ostatnim bajcie pliku, jeśli tryb jest ustawiony na a.

Text mode	Binary mode	Description
rt	rb	read
wt	wb	write
at	ab	append
r+t	r+b	read and update
w+t	w+b	write and update

File Processing – Otwieranie strumieni pierwszy plik

- Każda operacja na strumieniu musi być poprzedzona wywołaniem funkcji open(). Istnieją trzy dobrze zdefiniowane wyjątki od reguły.
- Kiedy nasz program się uruchamia, trzy strumienie są już otwarte i nie wymagają żadnych dodatkowych przygotowań. Co więcej, twój program może jawnie używać tych strumieni, jeśli zaimportujesz moduł sys

Nazwy tych strumieni to: sys.stdin, sys.stdout i sys.stderr.

- sys.stdin
    - stdin (jako standardowe wejście)
    - strumień stdin jest zwykle powiązany z klawiaturą, wstępnie otwarty do odczytu i traktowany jako podstawowe źródło danych dla uruchomionych programów;
    - dobrze znana funkcja input() domyślnie odczytuje dane ze standardowego wejścia.
- sys.stdout
    - stdout (jako standardowe wyjście)
    - strumień stdout jest zwykle powiązany z ekranem, wstępnie otwartym do zapisu, uważanym za główny cel wyprowadzania danych przez uruchomiony program;
    - dobrze znana funkcja print() wysyła dane do strumienia stdout.
- sys.stderr
    - stderr (jako standardowe wyjście błędu)
    - strumień stderr jest zwykle powiązany z ekranem, wstępnie otwartym do zapisu, uważanym za podstawowe miejsce, do którego działający program powinien przesyłać informacje o błędach napotkanych podczas jego pracy;
    - oddzielenie stdout (przydatne wyniki generowane przez program) od stderr (komunikaty o błędach, niezaprzeczalnie przydatne, ale nie dostarczające wyników) daje możliwość przekierowania tych dwóch rodzajów informacji do różnych celów.

File Processing – Zamykanie strumienia

Ostatnią operacją wykonaną na strumieniu (nie dotyczy to strumieni stdin, stdout i stderr, które tego nie wymagają) powinno być zamknięcie. Akcja ta jest wykonywana przez metodę wywoływaną z wnętrza obiektu open stream: stream.close().

- nazwa funkcji jest zdecydowanie samokomentująca: close()
- funkcja nie oczekuje dokładnie żadnych argumentów; strumień nie musi być otwierany
- funkcja nic nie zwraca, ale w przypadku błędu zgłasza wyjątek IOError;
- większość programistów uważa, że funkcja close() zawsze kończy się powodzeniem i dlatego nie ma potrzeby sprawdzania, czy poprawnie wykonała swoje zadanie.

To przekonanie jest tylko częściowo uzasadnione. Jeżeli strumień został otwarty do zapisu, a następnie wykonano serię operacji zapisu, może się zdarzyć, że dane wysłane do strumienia nie zostały jeszcze przesłane do fizycznego urządzenia (ze względu na mechanizm zwany buforowaniem lub buforowaniem).Ponieważ zamknięcie strumienia zmusza bufory do ich opróżnienia, może się zdarzyć, że spłukiwanie zakończy się niepowodzeniem, a zatem metoda close() również się nie powiedzie.

Obiekt IOError jest wyposażony we właściwość o nazwie errno (nazwa pochodzi od wyrażenia numer błędu) i można uzyskać do niego dostęp w następujący sposób:

```
try:
    # Some stream operations.
except IOError as exc:
    print(exc.errno)
```

File Processing – Zamykanie strumienia

Przyjrzyjmy się niektórym wybranym stałym przydatnym do wykrywania błędów strumienia:

- errno.EACCES → Permission denied - Błąd pojawia się, gdy próbujesz na przykład otworzyć plik z atrybutem tylko do odczytu do zapisu.
- errno.EBADF → Bad file numer - Błąd pojawia się, gdy próbujesz na przykład operować na nieotwartym strumieniu.
- errno.EEXIST → File exists - Błąd występuje, gdy próbujesz na przykład zmienić nazwę pliku na jego poprzednią nazwę.
- errno.EFBIG → File too large - Błąd występuje, gdy próbujesz utworzyć plik, który jest większy niż maksimum dozwolone przez system operacyjny.
- errno.EISDIR → Is a directory - Błąd występuje, gdy próbujesz traktować nazwę katalogu jako nazwę zwykłego pliku.
- errno.EMFILE → Too many open files - Błąd pojawia się, gdy próbujesz jednocześnie otworzyć więcej strumieni, niż jest to dopuszczalne dla twojego systemu operacyjnego.
- errno.ENOENT → No such file or directory - Błąd występuje, gdy próbujesz uzyskać dostęp do nieistniejącego pliku/katalogu.
- errno.ENOSPC → No space left on device - Błąd występuje, gdy na nośniku nie ma wolnego miejsca.

File Processing – Przetwarzanie pliku tekstowego

Najbardziej podstawową z tych metod jest ta oferowana przez funkcję read(), którą mogłeś zobaczyć w akcji w poprzedniej lekcji.

W przypadku zastosowania do pliku tekstowego funkcja jest w stanie:

- odczytać żądaną liczbę znaków (w tym tylko jeden) z pliku i zwrócić je jako ciąg znaków;
- przeczytaj całą zawartość pliku i zwróć je jako ciąg znaków;
- jeśli nie ma już nic do odczytania (wirtualna głowica czytająca dochodzi do końca pliku), funkcja zwraca pusty ciąg znaków.

Jak to działa:

- użyj mechanizmu try-except i otwórz plik o określonej nazwie (w naszym przypadku text.txt)
- spróbuj odczytać pierwszy znak z pliku (ch = s.read(1))
- jeśli ci się uda (potwierdza to pozytywny wynik sprawdzenia warunku while), wypisz znak (zwróć uwagę na argument end= .Nie chcesz przechodzić do nowej linii po każdym znaku!)
- zaktualizuj także licznik (cnt);
- spróbuj odczytać następny znak, a proces się powtórzy.

```
stream = open("tzop.txt", "rt", encoding = "utf-8")

print(stream.read())
```

```
from os import strerror

try:
    cnt = 0
    s = open('text.txt', "rt")
    ch = s.read(1)
    while ch != '':
        print(ch, end='')
        cnt += 1
        ch = s.read(1)
    s.close()
    print("\n\nCharacters in file:", cnt)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

File Processing – Przetwarzanie pliku tekstowego

Najbardziej podstawową z tych metod jest ta oferowana przez funkcję read(), którą mogłeś zobaczyć w akcji w poprzedniej lekcji.

W przypadku zastosowania do pliku tekstowego funkcja jest w stanie:

- odczytać żądaną liczbę znaków (w tym tylko jeden) z pliku i zwrócić je jako ciąg znaków;
- przeczytaj całą zawartość pliku i zwróć je jako ciąg znaków;
- jeśli nie ma już nic do odczytania (wirtualna głowica czytająca dochodzi do końca pliku), funkcja zwraca pusty ciąg znaków.

Jak to działa:

- użyj mechanizmu try-except i otwórz plik o określonej nazwie (w naszym przypadku text.txt)
- spróbuj odczytać pierwszy znak z pliku (ch = s.read(1))
- jeśli ci się uda (potwierdza to pozytywny wynik sprawdzenia warunku while), wypisz znak (zwróć uwagę na argument end= .Nie chcesz przechodzić do nowej linii po każdym znaku!)
- zaktualizuj także licznik (cnt);
- spróbuj odczytać następny znak, a proces się powtórzy.

```
stream = open("tzop.txt", "rt", encoding = "utf-8")

print(stream.read())
```

```
from os import strerror

try:
    cnt = 0
    s = open('text.txt', "rt")
    ch = s.read(1)
    while ch != '':
        print(ch, end='')
        cnt += 1
        ch = s.read(1)
    s.close()
    print("\n\nCharacters in file:", cnt)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

```
from os import strerror

try:
    cnt = 0
    s = open('text.txt', "rt")
    content = s.read()
    for ch in content:
        print(ch, end='')
        cnt += 1
    s.close()
    print("\n\nCharacters in file:", cnt)
except IOError as e:
    print("I/O error occurred: ", strerr(e.errno))
```

File Processing – Przetwarzanie pliku tekstowego

- Jeśli chcesz traktować zawartość pliku jako zestaw linii, a nie kilka znaków, metoda readline() ci w tym pomoże.
- Metoda próbuje odczytać cały wiersz tekstu z pliku i zwraca go jako ciąg znaków w przypadku powodzenia. W przeciwnym razie zwraca pusty ciąg.
- Otwiera to nowe możliwości - teraz możesz łatwo liczyć również linie, nie tylko znaki.
- Przy pomocy readlines gdy z pliku nie ma nic do odczytania, metoda zwraca pustą listę.
- W kodzie są dwie zagnieżdżone pętle: zewnętrzna używa wyniku readlines() do iteracji, podczas gdy wewnętrzna wypisuje wiersze znak po znaku.
- Protokół iteracji zdefiniowany dla obiektu file jest bardzo prosty - jego metoda __next__ zwraca po prostu następną linię wczytaną z pliku.
- Co więcej, można się spodziewać, że obiekt automatycznie wywoła metodę close(), gdy którykolwiek z odczytanych plików dotrze do końca pliku.

```
from os import strerror

try:
    ccnt = lcnt = 0
    s = open('text.txt', 'rt')
    line = s.readline()
    while line != '':
        lcnt += 1
        for ch in line:
            print(ch, end='')
            ccnt += 1
        line = s.readline()
    s.close()
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:      ", lcnt)
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

```
from os import strerror

try:
    ccnt = lcnt = 0
    s = open('text.txt', 'rt')
    lines = s.readlines(20)
    while len(lines) != 0:
        for line in lines:
            lcnt += 1
            for ch in line:
                print(ch, end='')
                ccnt += 1
        lines = s.readlines(10)
    s.close()
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:      ", lcnt)
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

File Processing – Przetwarzanie pliku tekstowego - write

- Metoda nazywa się write() i oczekuje tylko jednego argumentu - napisu, który zostanie przesłany do otwartego pliku.
- Do argumentu metody write() nie jest dodawany znak nowej linii, więc musisz go dodać samodzielnie, jeśli chcesz, aby plik był wypełniony pewną liczbą wierszy.
- Przykład w edytorze pokazuje bardzo prosty kod, który tworzy plik o nazwie newtext.txt, a następnie umieszcza w nim dziesięć linii .
- String, który ma zostać zapisany, składa się z wiersza słowa, po którym następuje numer wiersza.

```
from os import strerror

try:
    ccnt = lcnt = 0
    for line in open('text.txt', 'rt'):
        lcnt += 1
        for ch in line:
            print(ch, end='')
            ccnt += 1
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:      ", lcnt)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

```
from os import strerror

try:
    fo = open('newtext.txt', 'wt')
    for i in range(10):
        fo.write("line #" + str(i+1) + "\n")
    fo.close()
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

File Processing – Przetwarzanie pliku bitowego

- Dane amorficzne to dane, które nie mają określonego kształtu ani formy - są po prostu ciągiem bajtów.
- Danych amorficznych nie da się zapisać żadnym z przedstawionych wcześniej sposobów – nie są to ani stringi, ani listy.
- Powinien istnieć specjalny kontener, który będzie w stanie obsłużyć takie dane.
- Python ma więcej niż jeden taki kontener - jednym z nich jest wyspecjalizowana nazwa klasy bytearray - jak sama nazwa wskazuje, jest to tablica zawierająca (amorficzne) bajty.
- Jeśli chcesz mieć taki kontener np. aby wczytać obraz bitmapowy i w jakikolwiek sposób go przetworzyć, musisz go jawnie utworzyć, używając jednego z dostępnych konstruktorów.
- `data = bytearray(10)`
- Tablice bajtowe pod wieloma względami przypominają listy. Na przykład są zmienne, podlegają funkcji len() i można uzyskać dostęp do dowolnego z ich elementów przy użyciu konwencjonalnego indeksowania.
- Jest jedno ważne ograniczenie - nie wolno ustawiać żadnych elementów tablicy bajtów z wartością, która nie jest liczbą całkowitą (naruszenie tej reguły spowoduje wyjątek TypeError) oraz nie wolno przypisywać wartości, która nie pochodzi z zakres od 0 do 255 włącznie (chyba że chcesz wywołać wyjątek ValueError).
- Możesz traktować dowolne elementy tablicy bajtów jako wartości całkowite.

```
data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 - i

for b in data:
    print(hex(b))
```

File Processing – Przetwarzanie pliku bitowego

- Najpierw inicjujemy bytearray kolejnymi wartościami zaczynając od 10; jeśli chcesz, aby zawartość pliku była czytelna, zastąp 10 czymś w rodzaju ord('a') - spowoduje to utworzenie bajtów zawierających wartości odpowiadające alfabetycznej części kodu ASCII (nie myśl, że uczyni to plik tekstowym plik - nadal jest binarny, ponieważ został utworzony z flagą wb);
- następnie tworzymy plik za pomocą funkcji open() - jedyną różnicą w stosunku do poprzednich wariantów jest tryb open zawierający flagę b;
- metoda write() pobiera jej argument (bytearray) i wysyła go (w całości) do pliku;
- strumień jest następnie zamykany w rutynowy sposób.
- Metoda write() zwraca liczbę pomyślnie zapisanych bajtów.
- Jeśli wartości różnią się od długości argumentów metody, może ona ogłosić błędy zapisu.

```
from os import strerror

data = bytearray(10)

try:
    bf = open('file.bin', 'rb')
    bf.readinto(data)
    bf.close()

    for b in data:
        print(hex(b), end=' ')
except IOError as e:
    print("I/O error occurred:",
strerror(e.errno))
```

```
data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 - i

for b in data:
    print(hex(b))
```

```
from os import strerror

data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 + i

try:
    bf = open('file.bin', 'wb')
    bf.write(data)
    bf.close()
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

File Processing – Przetwarzanie pliku bitowego – czytanie strumienia

- Alternatywnym sposobem odczytania zawartości pliku binarnego jest metoda o nazwie read().
- Wywoływana bez argumentów, próbuje wczytać całą zawartość pliku do pamięci, czyniąc ją częścią nowo utworzonego obiektu klasy bytes.
- Ta klasa ma pewne podobieństwa do bytearray, z wyjątkiem jednej znaczącej różnicy - jest niezmienna.

```
from os import strerror

data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 + i

try:
    bf = open('file.bin', 'wb')
    bf.write(data)
    bf.close()
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))

try:
    bf = open('file.bin', 'rb')
    data = bytearray(bf.read())
    bf.close()

    for b in data:
        print(hex(b), end=' ')

except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

File Processing – Przetwarzanie pliku bitowego – czytanie strumienia

- Jeśli metoda read() jest wywoływana z argumentem, określa ona maksymalną liczbę bajtów do odczytania.
- Metoda próbuje odczytać żądaną liczbę bajtów z pliku, a długość zwróconego obiektu może zostać wykorzystana do określenia liczby faktycznie odczytanych bajtów.

```
from os import strerror

data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 + i

try:
    bf = open('file.bin', 'wb')
    bf.write(data)
    bf.close()
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))

try:
    bf = open('file.bin', 'rb')
    data = bytearray(bf.read(5))
    bf.close()

    for b in data:
        print(hex(b), end=' ')

except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

File Processing – Kopiowanie plików - proste i funkcjonalne narzędzie

- linie od 3 do 8: poproś użytkownika o nazwę pliku do skopiowania i spróbuj otworzyć go do przeczytania; zakończyć wykonywanie programu, jeśli otwarcie się nie powiedzie; użyta została funkcji exit(), aby zatrzymać wykonywanie programu i przekazać kod zakończenia do systemu operacyjnego; każdy kod zakończenia inny niż 0 oznacza, że program napotkał pewne problemy; użyj wartości errno, aby określić charakter problemu;
- linie od 10 do 16: powtórz prawie tę samą akcję, ale tym razem dla pliku wyjściowego;
- linia 18: przygotuj fragment pamięci do przeniesienia danych z pliku źródłowego do docelowego; taki obszar transferu jest często nazywany buforem, stąd nazwa zmiennej; wielkość bufora jest dowolna - w tym przypadku zdecydowaliśmy się na 64 kilobajty; technicznie rzecz biorąc, większy bufor jest szybszy w kopiowaniu elementów, ponieważ większy bufor oznacza mniej operacji we/wy; w rzeczywistości zawsze istnieje granica, której przekroczenie nie przynosi dalszych ulepszeń; przetestuj sam, jeśli chcesz.
- linia 19: policz skopiowane bajty - to jest licznik i jego wartość początkowa;
- linia 21: spróbuj zapełnić bufor po raz pierwszy;
- linia 22: dopóki otrzymasz niezerową liczbę bajtów, powtarzaj te same czynności;
- linia 23: zapisz zawartość bufora do pliku wyjściowego (uwaga: użyliśmy wycinka, aby ograniczyć liczbę zapisywanych bajtów, ponieważ write() zawsze woli zapisywać cały bufor)
- linia 24: zaktualizuj licznik;
- linia 25: przeczytaj następny fragment pliku;
- linie od 30 do 32: trochę końcowego sprzątania - zadanie wykonane.

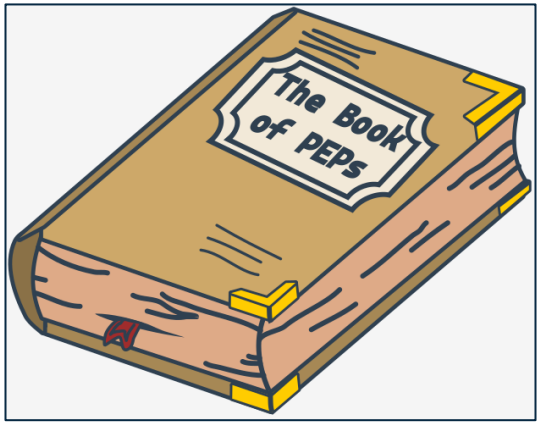
```
from os import strerror

srcname = input("Enter the source file name: ")
try:
    src = open(srcname, 'rb')
except IOError as e:
    print("Cannot open the source file: ", strerror(e.errno))
    exit(e.errno)

dstname = input("Enter the destination file name: ")
try:
    dst = open(dstname, 'wb')
except Exception as e:
    print("Cannot create the destination file: ", strerror(e.errno))
    src.close()
    exit(e.errno)

buffer = bytearray(65536)
total = 0
try:
    readin = src.readinto(buffer)
    while readin > 0:
        written = dst.write(buffer[:readin])
        total += written
        readin = src.readinto(buffer)
except IOError as e:
    print("Cannot create the destination file: ", strerror(e.errno))
    exit(e.errno)

print(total, 'byte(s) succesfully written')
src.close()
dst.close()
```

6 Standardy Pythona

Python Standards

Python standard – PEP

- PEP to zestaw standardów językowych oraz dostarcza informacji o wielu zmianach i procesach związanych z Pythonem.
- PEP 0 — Index of Python Enhancement Proposals (PEPs) (https://peps.python.org)
- PEP 1 – PEP Purpose and Guidelines - który zawiera informacje o przeznaczeniu PEP, ich rodzajach oraz wprowadza ogólne wytyczne
- PEP 8 – Style Guide for Python Code - który podaje konwencje i przedstawia najlepsze praktyki dotyczące kodowania w języku Python
- PEP 20 – The Zen of Python - który przedstawia listę zasad projektowania Pythona
- PEP 257 – Docstring Conventions - który zawiera wytyczne dotyczące konwencji i semantyki związanych z docstringami języka Python



Python standard – PEP 1

- PEP to akronim oznaczający Python Enhancement Proposals, który w rzeczywistości jest zbiorem wytycznych, najlepszych praktyk, opisów (nowych) funkcji i implementacji, a także procesów, mechanizmów i ważnych informacji dotyczących Pythona.
- Mówiąc najprościej, jeśli planowane jest dodanie nowej funkcji do Pythona, zostanie ona szczegółowo opisana w PEP wraz ze specyfikacjami technicznymi i uzasadnieniem jej wdrożenia.
- Istnieją trzy różne typy PEP:
  - Standardy śledź PEP, które opisują nowe funkcje i implementacje języka;
  - Informacyjne PEP, które opisują problemy projektowe w Pythonie, a także dostarczają wskazówek i informacji społeczności Pythona;
  - PEP procesów, które opisują różne procesy związane z Pythonem (np. proponowanie zmian, przedstawianie zaleceń, określanie pewnych procedur).
- PEP są skierowane przede wszystkim do programistów Pythona i członków społeczności Pythona. Są one przechowywane jako pliki tekstowe w repozytorium i można uzyskać do nich dostęp online pod adresem https://www.python.org/dev/peps/.
- Formaty PEP, szablony i proces przesyłania (w tym zgłaszanie błędów i przesyłanie aktualizacji), a także kolejne etapy: przegląd, rozwiązanie i konserwacja, zostały szczegółowo opisane w PEP 1 – Cel i wytyczne PEP.
- PEP 1 definiuje:
  - Rada Sterująca Pythona, czyli pięcioosobowy komitet i ostateczne władze, które akceptują lub odrzucają PEP;
  - Python’s Core Developers, czyli grupa wolontariuszy zarządzających Pythonem;
  - BDFL Pythona, czyli Guido van Rossum, pierwotny twórca Pythona, który pełnił funkcję Benevolent Dictator For Life do 2018 roku, kiedy to zrezygnował z procesu decyzyjnego.

Python standard – PEP 20 - Zen of Python

- Zen Pythona to zbiór 19 aforyzmów, które odzwierciedlają filozofię Pythona, jego zasady przewodnie i projekt.
- Tim Peters, od dawna główny współpracownik języka programowania Python i społeczności Pythona, napisał ten 19-wierszowy wiersz na liście mailingowej Pythona w 1999 roku, aw 2004 roku znalazł się on na pozycji nr 20 w Python Enhancement Proposals.
- Można uyć to za pomocą import this
- To, co widzisz, to zbiór pewnych ogólnych prawd dotyczących zasad projektowania i podejmowania decyzji w Pythonie.

Beautiful is better than ugly

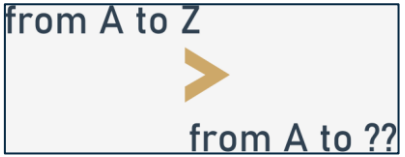
```
from math import sqrt
sidea = float(input("The length of the 'a' side:"))
sideb = float(input("The length of the 'b' side:"))
sidec = sqrt(a**2+b**2)
print("The length of the hypotenuse is", sidec )
```

```
from math import sqrt

side_a = float(input("The length of the 'a' side: "))
side_b = float(input("The length of the 'b' side: "))
hypotenuse = sqrt(a**2 + b**2)

print("The length of the hypotenuse is", hypotenuse)
```

Explicit is better than implicit

```
from fruit import *

apples(2, 3.45)
```

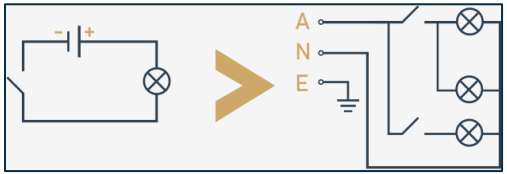
```
from fruit import apples, bananas

apples(quantity=2, price=3.45)
```



Simple is better than complex

```
import heapq

numbers = [-1, 12, -5, 0, 7, 21, 15, 1]
heapq.heapify(numbers)

sorted_numbers = []

while numbers:

sorted_numbers.append(heapq.heappop(numbers))

print(sorted_numbers)
```

```
numbers = [-1, 12, -5, 0, 7, 21, 15, 1]
numbers.sort()

print(numbers)
```



Python standard – PEP 20 - Zen of Python

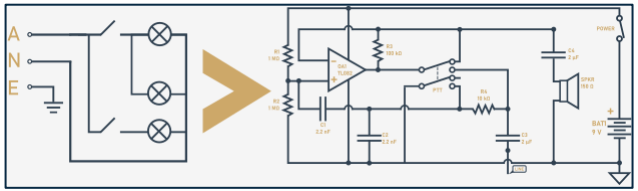
Complex is better than complicated

−

```
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
first_number = int(input("Enter the first number: "))
second_number = int(input("Enter the second number: "))
addition_result = first_number + second_number
print(first_number, "+", second_number, "=", addition_result)
```

+

```
def addition(x, y):
    print(x, "+", y, "=", x+y)

for i in range(5):
    first_number = int(input("Enter the first number: "))
    second_number = int(input("Enter the second number: "))
    addition(first_number, second_number)
```



Flat is better than nested

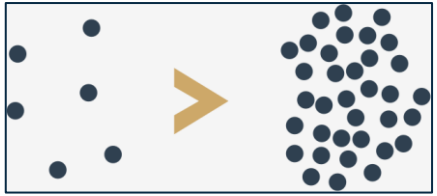
```
x = float(input("Enter a number: "))

if x > 0:
    if x > 1:
        if x > 2:
            if x > 3:
                if x >= 4:
                    if x <= 6:
                        print("x is a number between 4 and 6.")
else:
    print("x is not a number between 4 and 6.")
```

```
x = float(input("Enter a number: "))

if x >= 4 and x <=6:
    print("x is a number between 4 and 6.")
else:
    print("x is not a number between 4 and 6.")
```

Sparse is better than dense

```
x = 1
if x == 1 : print("Hello, World!")
```

```
x = 1
if x == 1:
    print("Hello, World!")
```



Readability counts

```
def f(i):
    I = i + (0.08 * i)
    return I
```

```
def calculate_gross_price(net_price):
    gross_price = net_price + (0.08 * net_price)
    return gross_price
```

Special cases aren't special enough to break the rules...

```python
def multiply_two_numbers(first_number, second_number):
    return first_number * second_number

print(multiply_two_numbers(7, 9))


def addingTwoNumbers(firstNumber, secondNumber):
    return firstNumber + secondNumber

print(addingTwoNumbers(7, 9))
```

```python
def multiply_two_numbers(first_number, second_number):
    return first_number * second_number

print(multiply_two_numbers(7, 9))


def add_two_numbers(first_number, second_number):
    return first_number + second_number

print(add_two_numbers(7, 9))
```

...Although, practicality beats purity

Cóż, musimy pamiętać, że ostatecznym celem jest rozwiązanie rzeczywistych problemów i napisanie kodu, który wykonuje określone (oczekiwane) zadanie. Jeśli Twój kod jest elegancki, czytelny i zgodny ze wszystkimi ważnymi konwencjami stylistycznymi, ale nie działa tak, jak powinien, to nie ma to większego sensu

Errors should never pass silently...

"...Unless explicitly silenced."

```
try:
    print(1/0)
except Exception as e:
    pass
```

```
try:
    print(1/0)
except ZeroDivisionError:
    print("Don't divide by zero!")
```

In the face of ambiguity, refuse the temptation to guess.

- Pierwszą rzeczą, o której należy pamiętać, jest zawsze testowanie kodu przed udostępnieniem go do produkcji i wdrożeniem u klientów.
- Ważną rzeczą, o której należy pamiętać, jest to, że testowanie kodu pozwala zaoszczędzić czas
- Inną rzeczą jest to, że powinieneś unikać pisania niejednoznacznego kodu, co oznacza, że nie powinieneś pozostawiać miejsca na zgadywanie.
- Nadaj zmiennym samokomentujące nazwy i zostaw komentarze tam, gdzie to konieczne. Jeśli importujesz moduł, spraw, aby import był jawny. Jeśli dany fragment jest złożony lub skomplikowany, wyjaśnij jego działanie. Nigdy nie zostawiaj komentarzy ani nie używaj nazw, które są błędne, mylące lub wprowadzające w błąd

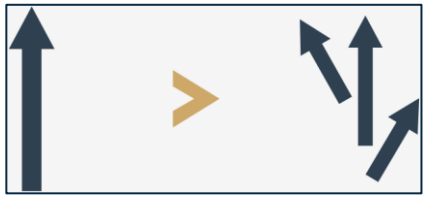
Python standard – PEP 20 - Zen of Python

There should be one – and preferably only one – obvious way to do it

“Although that way may not be obvious at first unless you're Dutch.”

- Ten sam cel można osiągnąć na wiele sposobów. Na przykład, jeśli chcesz pobrać imię i nazwisko użytkownika i wyświetlić je na ekranie, możesz to zrobić na jeden z następujących sposobów
- Wytyczne przypominają nam również, że dobrze jest przestrzegać standardów i konwencji dotyczących używania języka. Na przykład, jeśli do tej pory używałeś snake_case do nazywania zmiennych w swoim kodzie, rozpoczęcie używania CamelCase dla reszty kodu w jednym i tym samym programie może być złym pomysłem. No chyba, że robisz to w konkretnym celu, a plusy takiego podejścia są większe niż minusy.
- Na koniec aforyzm działa jako delikatne wskazanie jeszcze jednej ważnej rady: tam, gdzie to możliwe, dobrze jest pamiętać, że każda funkcja, każda klasa, każda metoda – każda jednostka – powinna mieć jedną spójną odpowiedzialność. Czemu? Ponieważ takie podejście pomaga uzyskać większą przejrzystość i tworzyć czystszy kod, sprawia, że jego utrzymanie jest łatwiejsze i tańsze oraz mniej podatne na błędy.
- Jeśli chodzi o drugą część aforyzmu, to z jednej strony to żart: Holendrzy na pewno mają inny sposób myślenia, inny światopogląd i inny sposób zabierania się do działania (na pewno pamiętacie, że Guido van Rossum też jest Holendrem).

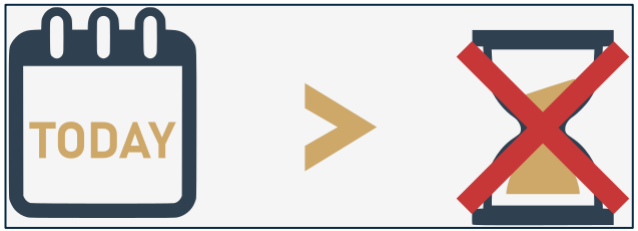
```
first_name = input("Enter your first name: ")
last_name = input("Enter your last name: ")

print("Your name is:", first_name, last_name)
print("Your name is:" + " " + first_name + " " + last_name)
print("Your name is: {} {}".format(first_name, last_name))
```



Now is better than never

“Although never is often better than *right* now.”

- Nie odkładaj do jutra tego, co możesz zrobić dzisiaj. To znane przysłowie. Zanim faktycznie przystąpisz do wykonywania tych czynności — pisania kodu — być może zapomniałeś pomysłów lub informacji potrzebnych do wykonania tego dobrze.
- Python pozwala szybko przełożyć Twoje pomysły na działający kod. Ilekroć doświadczysz efektu eureki lub masz chwilę inspiracji, zapisz swoje myśli i zakoduj je w Pythonie (lub przynajmniej użyj jakiejś formy pseudokodu) – nawet jeśli Twój kod jest daleki od doskonałości. Możesz go później bardzo łatwo udoskonalić, rozwinąć lub przeprojektować.
- Jeśli ulegniesz pokusie ukończenia programu i wypuszczenia go dopiero wtedy, gdy będzie doskonały, istnieje duże prawdopodobieństwo, że nigdy tego nie zrobisz.
- Z drugiej strony aforyzm każe nam nie zapominać o właściwej równowadze. Tak jak doskonałe jest wrogiem dobrego, tak często okazuje się, że szybszy jest wrogiem wolniejszego. Są sytuacje, w których nie należy się spieszyć.



If the implementation is hard to explain, it's a bad idea

“If the implementation is easy to explain, it may be a good idea.”

- Wszystko i wszystko, co można wyjaśnić słowami, można przetłumaczyć na kod, a ostatecznie przekształcić w dobrze działający program komputerowy.
- Jeśli potrafisz wyjaśnić, czego oczekujesz od programu, co chcesz, żeby robił – taki program da się zaprojektować.
- Zachowaj prostotę; im prościej, tym lepiej.
- Jednak nawet jeśli coś jest łatwe do wyjaśnienia, nie oznacza to, że jest dobre. Po prostu łatwiej ocenić, czy tak jest, czy nie.

```
from instruments.guitars import fender, ibanez

fender(page)
ibanez(vai)
```

```
from instruments import guitars

guitars.fender(page)
guitars.ibanez(vai)
```

Namespaces are one honking great idea – let's do more of those!

- Python zapewnia dobry, dobrze zorganizowany mechanizm przestrzeni nazw do zarządzania dostępnością identyfikatorów, których chcesz używać, i unikania konfliktów z już istniejącymi nazwami w różnych zakresach.
- Mówiąc prościej, oznacza to, że za każdym razem, gdy definiujesz zmienną, Python „zapamiętuje” dwie rzeczy: identyfikator zmiennej i wartość, którą do niej przekazujesz
- Funkcje, klasy, obiekty, moduły, pakiety… to wszystko są przestrzenie nazw. Z tego faktu wynika, że bardziej specyficzna przestrzeń nazw nie może zostać zmieniona przez mniej specyficzną przestrzeń nazw, ponieważ znajdują się one w dwóch różnych zakresach

Python standard – PEP 8

- Jak wspomniano wcześniej, PEP 8 to dokument, który zapewnia konwencje kodowania (przewodnik po stylach kodu) dla kodu Pythona.
- PEP 8 jest uważany za jeden z najważniejszych PEP i obowiązkową lekturę dla każdego profesjonalnego programisty Pythona, ponieważ pomaga uczynić kod bardziej spójnym, bardziej czytelnym i wydajnym.
- Mimo że niektóre projekty programistyczne mogą przyjmować własne wytyczne dotyczące stylu (w takim przypadku takie wytyczne specyficzne dla projektu mogą być preferowane w stosunku do konwencji przewidzianych przez PEP 8, zwłaszcza w przypadku jakichkolwiek konfliktów lub problemów ze wsteczną kompatybilnością), PEP 8 nadal zaleca się przeczytanie najlepszych praktyk, ponieważ pomagają one lepiej zrozumieć filozofię Pythona i stać się bardziej świadomym i biegłym programistą.
- PEP 8 wciąż ewoluuje, ponieważ nowe konwencje są identyfikowane i uwzględniane, a jednocześnie niektóre stare konwencje są identyfikowane jako przestarzałe i zniechęcane do ich przestrzegania.
- „Głupia konsekwencja jest chochlikiem małych umysłów". To cytat z eseju Ralpha Waldo Emersona „Samopoleganie", w którym Emerson zachęca czytelników do bycia konsekwentnymi w swoich przekonaniach i praktykach. W naszym przypadku oznacza to, że nie możemy zapomnieć o jednym prostym, ale ważnym spostrzeżeniu: nasz kod będzie znacznie częściej czytany niż pisany.
- Z jednej strony spójność jest kluczowym czynnikiem decydującym o czytelności kodu. Z drugiej strony, niezgodność z PEP 8 może być czasami lepszym rozwiązaniem. Jeśli przewodniki po stylach nie mają zastosowania do twojego projektu, lepiej je zignorować i samemu zdecydować, co jest najlepsze
- Kiedy należy zignorować określone wytyczne PEP 8:
    - Jeśli przestrzeganie ich będzie oznaczać, że złamiesz kompatybilność wsteczną.
    - Jeśli przestrzeganie ich będzie miało negatywny wpływ na czytelność kodu.
    - Jeśli podążanie za nimi spowoduje niespójność z resztą kodu. (Jednak może to być dobra okazja do przepisania kodu i dostosowania go do PEP 8).
    - Jeśli nie ma dobrego powodu, aby kod był zgodny z PEP 8 lub kod jest starszy niż PEP 8.

PEP 8 ma na celu poprawę czytelności kodu i „ujednolicenie go w szerokim spektrum kodu Pythona". Utrzymywanie zgodności kodu Pythona z PEP 8 jest zatem dobrym pomysłem, ale nigdy nie należy ślepo stosować się do tych zaleceń. Zawsze należy kierować się najlepszą oceną sytuacji.

Python standard – PEP 8 compliant checkers

- Istnieje wiele przydatnych narzędzi, które mogą pomóc w sprawdzeniu poprawności stylu kodu i porównaniu go z konwencjami stylu PEP 8. Narzędzia te można zainstalować i uruchomić lokalnie lub uzyskać do nich dostęp online.
- pycodestyle (wcześniej nazywany pep8, ale nazwa została zmieniona, aby uniknąć nieporozumień) - sprawdzanie przewodnika po stylu Pythona; pozwala sprawdzić zgodność kodu Pythona z konwencjami stylu w PEP 8.
- pycodestyle (wcześniej nazywany pep8, ale nazwa została zmieniona, aby uniknąć nieporozumień) - sprawdzanie przewodnika po stylu Pythona; pozwala sprawdzić zgodność kodu Pythona z konwencjami stylu w PEP 8. Możesz zainstalować narzędzie za pomocą następującego polecenia w terminalu: pip install pycodestyle https://github.com/PyCQA/pycodestyle
- Możesz także zainstalować autopep8, aby automatycznie sformatować kod Pythona, aby był zgodny z wytycznymi PEP 8. Aby móc z niego korzystać, potrzebujesz instalacji pycodestyle na swoim komputerze aby wskazać te części kodu, które wymagają poprawek formatowania. https://pypi.org/project/autopep8/
- PEP 8 online to internetowe narzędzie do sprawdzania PEP 8 stworzone przez Valentina Bryukhanova, które umożliwia wklejenie kodu lub przesłanie pliku i zweryfikowanie go pod kątem wytycznych dotyczących stylu PEP 8. Narzędzie online jest zbudowane przy użyciu Flask, Twitter Bootstrap i modułu PEP8 http://pep8online.com/about
- PEP 8 ma sprawić, że twoje doświadczenie z kodowaniem będzie lepsze, a twoje życie o wiele łatwiejsze. Jak wspomniano wcześniej, sposób pisania kodu ma duży wpływ na jego czytelność. Nie należy jednak zapominać, że może to również określać jego legalność składniową.
- W tej sekcji skupimy się na zaleceniach dotyczących stylu związanych z takimi rzeczami jak:
    - wcięcia, używanie tabulatorów i spacji;
    - długość linii, podziały linii i puste linie;
    - kodowanie plików źródłowych i import modułów.

Python standard – PEP 8

Wcięcia:

- Poziom wcięcia, rozumiany jako wiodące białe znaki (tj. spacje i tabulatory) na początku każdego wiersza logicznego, służy do grupowania instrukcji.
- Pisząc kod w Pythonie, należy pamiętać o przestrzeganiu tych dwóch prostych zasad:
  - Użyj czterech spacji na poziom wcięcia i;
  - Używaj spacji zamiast tabulacji.

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

def my_function(x, y):
    return x * y
```

Kontynuacja linii:

- Linie kontynuacji (tj. logiczne linie kodu, które chcesz podzielić, ponieważ są za długie lub chcesz poprawić czytelność) są dozwolone, jeśli używasz nawiasów/nawiasów klamrowych:

```
my_list_one = [1, 2, 3,
    4, 5, 6
]

a = my_function_name(a, b, c,
    d, e, f)

########################################

my_list_one = [
    1, 2, 3,
    4, 5, 6,
    ]

a = my_function_name(a, b, c,
                     d, e, f)
```

Python standard – PEP 8

Maksymalna długość linii i podziały linii:

- Jeśli to możliwe, powinieneś ograniczyć wszystkie linie do maksymalnie 79 znaków, ponieważ pomoże to uniknąć zawijania kilku linii kodu.
- Jeśli zawijanie linii jest nieuniknione, użyj domniemanej kontynuacji linii Pythona z poprzedniej strony.
- Biblioteka Pythona jest w tej kwestii konserwatywna i wymaga użycia nie więcej niż 79 znaków w wierszu (72 w przypadku komentarzy/ciągów dokumentów).

Podziały linii i operatory:

- Mimo że w Pythonie możesz łamać linie kodu przed operatorami binarnymi lub po nich (pod warunkiem, że robisz to konsekwentnie i ta konwencja była wcześniej używana w twoim kodzie), zaleca się stosowanie się do sugestii stylu Donalda Knutha i łamanie przed operatorami binarnymi operatorów, ponieważ skutkuje to bardziej czytelnym, przyjaznym dla oka kodem.

```
total_fruits = (apples
                + pears
                + grapes
                - (black currants - red currants)
                - bananas
                + oranges)
```

Puste linie:

- Puste linie, zwane pionowymi spacjami, poprawiają czytelność kodu.
- Pozwalają osobie czytającej twój kod zobaczyć podział kodu na sekcje, pomagają lepiej zrozumieć relacje między sekcjami i łatwiej uchwycić logikę danych bloków kodu.
- W ten sam sposób użycie zbyt wielu pustych wierszy w kodzie sprawi, że będzie wyglądał na rzadki i trudniejszy do śledzenia, dlatego zawsze musisz uważać, aby ich nie nadużywać.

Python standard – PEP 8

Puste linie:

- Puste linie, zwane pionowymi spacjami, poprawiają czytelność kodu.
- Pozwalają osobie czytającej twój kod zobaczyć podział kodu na sekcje, pomagają lepiej zrozumieć relacje między sekcjami i łatwiej uchwycić logikę danych bloków kodu.
- W ten sam sposób użycie zbyt wielu pustych wierszy w kodzie sprawi, że będzie wyglądał na rzadki i trudniejszy do śledzenia, dlatego zawsze musisz uważać, aby ich nie nadużywać.

```
class ClassOne:
    pass


Class ClassTwo:
    pass


def my_top_level_function():
    return None
```

dwie puste linie otaczające funkcje najwyższego poziomu i definicje klas

pojedyncza pusta linia otaczająca definicje metod wewnątrz klasy

```
class MyClass:
    def method_one(self):
        return None

    def method_two(self):
        return None
```

puste wiersze w funkcjach w celu wskazania sekcji logicznych

```
def calculate_average():
    how_many_numbers = int(input("How many
numbers? "))

    if how_many_numbers > 0:
        sum_numbers = 0
        for i in range(0, how_many_numbers):
            number = float(input("Enter a number: "))
            sum_numbers += number

        average = 0
        average = sum_numbers /
how_many_numbers

        return average
    else:
        return "Nothing happens."
```

Python standard – PEP 8

Domyślne kodowania:

- Zaleca się używanie domyślnego kodowania Pythona (Python 3 — UTF-8, Python 2 — ASCII). Kodowanie inne niż domyślne jest odradzane i powinno być używane wyłącznie do celów testowych lub w sytuacjach, gdy w komentarzach lub dokumentach używane jest nazwisko (np. nazwisko autora) zawierające znak inny niż ASCII.
- PEP 8 stwierdza, że „wszystkie identyfikatory w standardowej bibliotece Pythona MUSZĄ używać identyfikatorów tylko ASCII i POWINNY używać angielskich słów, gdy tylko jest to możliwe”.

Importowania:

- Importy należy zawsze umieszczać na początku skryptu, pomiędzy komentarzami/dokumentami modułu a wartościami globalnymi i stałymi modułu, przestrzegając następującej kolejności:
  - Importy bibliotek standardowych;
  - Powiązany import stron trzecich;
  - Lokalne importy specyficzne dla aplikacji/bibliotek.
- Upewnij się, że wstawiłeś pusty wiersz, aby oddzielić każdą z powyższych grup importów.

```
import sys, os

from animals import *
```

```
import os
import sys

from subprocess import Popen, PIPE

import
animals.mammals.dogs.puppies
```

Zalecenia dotyczące cudzysłowów, białych znaków i końcowych przecinków:

- Skupimy się na zaleceniach dotyczących stylu związanych z takimi rzeczami jak:
  - cudzysłowy;
  - odstępy w wyrażeniach i instrukcjach oraz użycie końcowych przecinków.

Python standard – PEP 8

Cytowanie ciągów:

- Python pozwala nam używać ciągów znaków w pojedynczym cudzysłowie (np. „łańcuch znaków”) i podwójnie cudzysłowach (np. „łańcuch znaków”). Są takie same i nie ma specjalnych zaleceń w PEP mówiących, jaki styl powinieneś przyjąć w swoim kodzie. Ponownie najważniejsza zasada brzmi: bądź konsekwentny w swoim wyborze.
- Jednak w celu poprawy czytelności PEP 8 zaleca unikanie stosowania w ciągach znaków ukośników odwrotnych (znaków ucieczki). To znaczy że:
- jeśli twój ciąg znaków zawiera pojedyncze cudzysłowy, zaleca się użycie ciągów ujętych w podwójne cudzysłowy;
- jeśli Twój ciąg zawiera znaki cudzysłowu, zaleca się użycie ciągów ujętych w pojedyncze cudzysłowy.
- W przypadku łańcuchów ujętych w potrójne cudzysłowy PEP 8 zaleca, aby zawsze używać znaków cudzysłowu, aby zachować spójność z konwencją docstring wyszczególnioną w PEP 257

Białe znaki w wyrażeniach i instrukcjach:

- PEP 8 zawiera długą sekcję, która pokazuje przykłady poprawnego i niepoprawnego użycia spacji w kodzie.
- Ogólnie rzecz biorąc, należy unikać używania zbyt dużej ilości białych znaków, ponieważ utrudnia to śledzenie kodu.

```
my_list = ( dog[ 2 ] , 5 , { "year": 1980 } , "string" )
if 5 in my_list : print( "Hello!" ) ; print( "Goodbye!" )
```

```
my_list = (dog[2], 5, {"year": 1980}, "string")
if 5 in my_list: print("Hello!"); print("Goodbye!")
```

Python standard – PEP 8

Końcowe przecinki:

- po końcowym przecinku, po którym następuje nawias zamykający,
- bezpośrednio przed nawiasem otwierającym, który oznacza początek listy argumentów wywołania funkcji,
- bezpośrednio przed nawiasem otwierającym, który oznacza początek indeksowania/wycinania.

```
my_tuple = (0, 1, 2, )
my_function (5)
my_dictionary ['key'] = my_list [index]


my_tuple = (0, 1, 2,)
my_function(5)
my_dictionary['key'] = my_list[index]
```

Zalecenia dotyczące korzystania z komentarzy

- Komentarze mają na celu poprawę czytelności kodu bez wpływu na wynik działania programu.
- Istnieje kilka zasad, których należy przestrzegać, zostawiając komentarze w kodzie:
  - Pisz komentarze, które nie będą sprzeczne z kodem ani nie wprowadzają czytelnika w błąd. Są znacznie gorsze niż całkowity brak komentarza.
  - Zaktualizuj swoje komentarze, gdy Twój program zostanie zaktualizowany.
  - Pisz komentarze jako pełne zdania (pierwsze słowo pisz wielką literą, jeśli nie jest to identyfikator, i zakończ zdanie kropką)
  - Pisząc komentarze blokowe z komentarzami wielozdaniowymi, używaj dwóch spacji po każdej kropce kończącej zdanie, z wyjątkiem ostatniego zdania.
  - Pisz komentarze w języku angielskim (chyba, że masz 100% pewność, że kod nigdy nie zostanie odczytany przez osoby nieznające Twojego języka).
  - Komentarze powinny mieć nie więcej niż 72 znaki w linii (ale to już wiesz)

```
# Program that calculates body mass index (BMI).

height = float(input("Your height (in meters): "))
weight = float(input("Your weight (in kilograms): "))
bmi = round(weight / (height*height), 2)

print("Your BMI: {}".format(bmi))
```

Python standard – PEP 8

Komentarze w tekście:

- Komentarze blokowe są zwykle dłuższe i należy ich używać do wyjaśniania fragmentów kodu, a nie poszczególnych wierszy.
- Pozwalają zostawić informacje dla czytelnika w wielu wierszach (i wielu zdaniach). Generalnie blokuj komentarze:
  - powinny odnosić się do następującego po nich kodu;
  - powinny być wcięte na tym samym poziomie, co kod, który opisują.
- Pisząc komentarze blokowe, zaczynaj każdy wiersz znakiem #, po którym następuje pojedyncza spacja, a akapity oddzielaj wierszem zawierającym tylko symbol #.

Zalecenia dotyczące korzystania z komentarzy

- Komentarze wbudowane to komentarze napisane w tym samym wierszu, co Twoje wypowiedzi.
- Powinny odnosić się lub dostarczać dalszych wyjaśnień do pojedynczej linii kodu lub pojedynczej instrukcji. Nie należy ich nadużywać.
- Ogólnie rzecz biorąc, komentarze śródliniowe powinny wyglądać następująco:
  - oddzielone dwiema (lub więcej) spacjami od wypowiedzi, do której się odnoszą;
  - używany oszczędnie.

```
def calculate_product():
    # Calculate the average of three numbers obtained from the user.
Then
    # multiply the result by 4.17, and assign it to the product variable.
    #
    # Return the value passed to the product variable and use it
    # for the subsequent x to y calculations to speed up the process.
    sum_numbers = 0

    for number in range(0, 3):
        number = float(input("Enter a number: "))
        sum_numbers += number

    average = (sum_numbers / 3) * 4.17
    product = average
    return product


x = product * 1.73
y = x ** 2
x_to_y = (x * y) / 1.05
```

Python standard – PEP 8

Konwencje nazewnictwa:

- Podczas programowania często musisz nazwać identyfikatory i inne jednostki w swoim kodzie. Nadanie odpowiednich nazw i unikanie niewłaściwych z pewnością zwiększy czytelność kodu i zaoszczędzi Tobie (i innym programistom czytającym Twój kod) dużo czasu i wysiłku.
- Z pewnością przestrzegasz już pewnych konwencji dotyczących nadawania nazw zmiennym, funkcjom i klasom w swoim kodzie; niektóre z nich mogą wynikać z Twojego wcześniejszego doświadczenia w programowaniu w innych językach, inne mogą być czysto praktycznym wyborem, a jeszcze inne mogą wynikać z wymagań projektowych lub praktyk przyjętych przez Twoją firmę lub zespół.

Style nazewnictwa

- Nazywając zmienną, należy używać małych liter lub słów i rozdzielać słowa podkreśleniami, np. x, var, moja_zmienna. Ta sama konwencja dotyczy zmiennych globalnych.
- Funkcje podlegają tym samym zasadom, co zmienne, tj. podczas nadawania funkcji funkcji należy używać małych liter lub słów oddzielonych podkreśleniami, np. zabawa, moja_funkcja.
- Nadając nazwę klasie, powinieneś przyjąć styl CamelCase, np. MySampleClass, lub jeśli jest tylko jedno słowo, zacznij je wielką literą, np. Sample.
- Nadając nazwę metodzie, należy użyć słowa pisanego małą literą lub słów oddzielonych podkreśleniami, np. metoda, moja_klasa_metoda. Powinieneś zawsze używać self jako pierwszego argumentu do metod instancji i cls jako pierwszego argumentu do metod klasy.
- Podczas nadawania nazwy stałej należy używać wielkich liter, a słowa rozdzielać podkreślnikami, np. TOTAL, MOJA_STAŁA.

```
# A multi-line docstring:

def fun(x, y):
    """Convert x and y to strings,
    and return a list of strings.
    """
    ...

# A single-line docstring:

def fun(x):
    """Return the square root of x."""
    ...
```

- Nadając nazwę modułowi, należy użyć słowa lub słów pisanych małą literą, najlepiej krótkich, i oddzielić je podkreślnikami, np. sample.py, my_samples..
- Nazywając paczkę należy używać słowa lub słów pisanych małą literą, najlepiej krótkich. Nie należy oddzielać słów, np. pakiet, moja paczka.

Python standard – PEP 257

- PEP 257 to dokument stworzony w ramach Python Developer's Guide, który stanowi próbę ujednolicenia wysokopoziomowej struktury docstringów. Przedstawia konwencje, najlepsze praktyki i semantykę (nie prawa ani przepisy!) związane z dokumentowaniem kodu Pythona za pomocą docstringów.
- Krótko mówiąc, próbuje odpowiedzieć na dwa następujące pytania: Co powinny zawierać dokumenty Pythona? Jak należy używać docstringów Pythona?
- Docstring to „literał łańcuchowy, który występuje jako pierwsza instrukcja w module, funkcji, klasie lub definicji metody. Taki ciąg dokumentacyjny staje się specjalnym atrybutem __doc__ tego obiektu”. (PEP 257)
- Innymi słowy, docstrings to łańcuchy dokumentacji Pythona, które są używane w definicjach klas, modułów, funkcji i metod w celu dostarczenia informacji o funkcjonalności większego fragmentu kodu w sposób normatywny.
- Pomagają programistom (w tym tobie) zapamiętać i zrozumieć cel, działanie i możliwości poszczególnych bloków lub sekcji kodu.

Docstring a komentarze

- komentarze służą do komentowania twojego kodu, podczas gdy docstrings są używane do dokumentowania twojego kodu

Komentarze:

- Komentarze to niewykonalne instrukcje w Pythonie, co oznacza, że są ignorowane przez interpreter Pythona; nie są one przechowywane w pamięci i nie można uzyskać do nich dostępu podczas wykonywania programu
- Głównym celem komentarzy jest zwiększenie czytelności i zrozumiałości kodu oraz wytłumaczenie kodu użytkownikowi w zrozumiały sposób. Użytkownik oznacza tutaj zarówno innych programistów, jak i Ciebie (np. gdy po jakimś czasie wrócisz do swojego kodu) – kogoś, kto będzie chciał lub musiał zmodyfikować, rozszerzyć lub utrzymać kod.
- Komentarze nie mogą zostać przekształcone w dokumentację; ich celem jest uproszczenie kodu, dostarczenie precyzyjnych informacji i pomoc w zrozumieniu intencji konkretnego fragmentu/linii.

Docstrings:

- Dostęp do dokumentów można uzyskać, czytając kod źródłowy i używając atrybutu __doc__ lub funkcji help().
- Głównym celem docstringów jest dokumentowanie twojego kodu – opisywanie jego użycia, funkcjonalności i możliwości użytkownikom, którzy niekoniecznie muszą wiedzieć, jak to działa.
- Docstringi można łatwo przekształcić w rzeczywistą dokumentację, która opisuje zachowanie modułu lub funkcji, znaczenie parametrów lub przeznaczenie określonego pakietu.

Python standard – PEP 257

- Dokumentowanie kodu pomaga zachować czystszy, bardziej czytelny i zrównoważony kod, co oznacza, że jest to jedna z najlepszych praktyk, które dobry, odpowiedzialny programista powinien przyjąć jako część zestawu narzędzi codziennych warsztatów programistycznych
- Jeśli chcesz zawrzeć dłuższy komentarz w swoim kodzie, możesz użyć komentarza wielowierszowego, w takim przypadku powinieneś użyć znaku krzyżyka na początku każdego wiersza komentarza.
- Komentarze należy umieszczać blisko kodu, który opisujesz, aby czytelnicy wiedzieli, do której części kodu się odnosisz. Powinieneś być precyzyjny – nie dołączaj nieistotnych lub zbędnych informacji; a przede wszystkim – staraj się projektować i pisać swój kod w taki sposób, aby łatwo i zrozumiale się komentował (np. nadawał samokomentujące nazwy zmiennym).

Kiedy używamy komentarzy, Poza najbardziej oczywistymi przypadkami, takimi jak opisy kodu i algorytmów, komentarze mogą służyć jeszcze kilku innym przydatnym celom

- mogą pomóc Ci otagować te fragmenty kodu, które mają być wykonane w przyszłości lub pozostawione do dalszej poprawy,

```
# TODO: Add a function that takes the val and prc arguments.
```

- mogą pomóc Ci skomentować (i odkomentować) te fragmenty kodu, które chcesz przetestować

```
def fun(val):
    return val * 2

user_value = int(input("Enter the value: "))
# fun(user_value)
# user_value = user_value + "foo"

print(fun(user_value))
```

- mogą pomóc Ci zaplanować pracę i nakreślić pewne sekcje kodu, które będziesz projektować

```
# Step 1: Ask the user for the value.
# Step 2: Change the value to an int and handle possible exceptions.
# Step 3: Print the value multiplied by 0.7.
```

Python standard – PEP 484

- Typ podpowiedzi to mechanizm wprowadzony w Pythonie 3.5 i opisany w PEP 484, który pozwala wyposażyć kod w dodatkowe informacje bez użycia komentarzy.
- Jest to opcjonalna, ale bardziej sformalizowana funkcja, która umożliwia użycie wbudowanego modułu pisania w Pythonie w celu podania wskazówek dotyczących typu w kodzie w celu pozostawienia pewnych sugestii, zaznaczenia pewnych możliwych problemów, które mogą pojawić się podczas programowania przetwarzać i etykietować określone nazwy informacjami o typie.
- W skrócie, podpowiadanie typu pozwala statycznie wskazać informacje o typie obiektów Pythona, co oznacza, że możesz np. dodać do funkcji informację o typie – wskazać typ argumentu, jaki funkcja akceptuje, lub wartość zwróci.
- Podpowiedź typu jest opcjonalna, co oznacza, że PEP 484 nie zobowiązuje do pozostawienia w kodzie żadnych statycznych informacji związanych z typowaniem. Pierwszy przykład jest wolny od jakichkolwiek wskazówek typu.
- podpowiedzi typu mogą pomóc w udokumentowaniu kodu. Zamiast pozostawiać informacje związane z argumentami i odpowiedziami w docstringach, możesz użyć do tego celu samego języka. Może to być elegancki i użyteczny sposób na podkreślenie niektórych ważniejszych informacji o kodzie, zwłaszcza podczas publikowania kodu w projekcie, dzielenia się nim z innymi programistami lub pozostawiania sobie wskazówek, kiedy będziesz musiał wrócić do kodu źródłowego w przyszłości . W niektórych większych projektach programistycznych podpowiedzi typu są zalecaną praktyką, która pomaga zespołom lepiej zrozumieć, w jaki sposób typy przechodzą przez kod.
- Podpowiedzi typu pozwalają skuteczniej zauważać pewne rodzaje błędów i pisać piękniejszy, a przede wszystkim czystszy kod. Korzystając ze wskazówek dotyczących typów, dokładniej myślisz o typach w swoim kodzie, co pomaga zapobiegać lub wykrywać niektóre błędy, które mogą wynikać z dynamicznej natury Pythona. (Jednak nie jesteśmy zwolennikami zmuszania Pythona do statycznego pisania).
- Musisz pamiętać, że podpowiedzi typu w Pythonie nie są używane w czasie wykonywania, co oznacza, że wszystkie informacje o typie pozostawione w kodzie w postaci adnotacji są usuwane podczas wykonywania programu. Innymi słowy, podpowiadanie typu nie ma żadnego wpływu na działanie twojego kodu. Z drugiej strony, gdy jest używany wraz z systemem sprawdzania typu lub narzędziami podobnymi do kłaczków, które można podłączyć do edytora lub IDE, może wspierać pisanie kodu poprzez automatyczne uzupełnianie pisania oraz wykrywanie i wyróżnianie błędów przed wykonaniem kodu .
- Ponieważ wskazówki typu nie mają wpływu na kod źródłowy, oznacza to, że nie mają one wpływu na czas działania (znaki są ignorowane przez Pythona w czasie wykonywania, co nie ma wpływu na przyspieszenie interpretacji/kompilacji).

```
# No type information added:
def hello(name):
    return "Hello, " + name


# Type information added to a function:
def hello(name: str) -> str:
    return "Hello, " + name
```

Python standard – PEP 257

- Powiedzieliśmy już, że docstringi mogą być używane w definicjach klas, modułów, funkcji i metod. Teraz chcemy to rozwinąć: istnieją przypadki, w których nie tylko można je uwzględnić, ale należy je uwzględnić. Mówiąc dokładniej – wszystkie publiczne moduły, funkcje, klasy i metody, które są eksportowane przez dany moduł, powinny mieć docstringi.
- Metody niepubliczne nie muszą zawierać docstringów. Zaleca się jednak pozostawienie komentarza zaraz po linii def opisującej, co faktycznie robi metoda. W przypadku pakietów należy je również udokumentować, a dokumentację pakietu można zapisać w dokumentacji modułu pliku __init__.py w folderze pakietu.
- Jak powiedzieliśmy wcześniej, ciągi dokumentów to literały łańcuchowe występujące jako pierwsza instrukcja w module, funkcji, klasie lub metodzie. Jednak ważne (i uczciwe) jest dodanie, że literały łańcuchowe mogą również występować w wielu innych miejscach kodu Pythona i nadal służyć jako dokumentacja. I chociaż mogą nie być już dostępne jako atrybuty obiektów środowiska wykonawczego, nadal można je wyodrębnić za pomocą niektórych określonych narzędzi programowych (więcej informacji na ten temat można znaleźć w PEP 256, który zawiera informacje o Docutils, systemie przetwarzania dokumentów w języku Python).

Wyróżniamy dwa rodzaje takich „dodatkowych dokumentów”:

- docstringi atrybutów, które znajdują się bezpośrednio po instrukcji przypisania na najwyższym poziomie modułu (atrybuty modułu), klasy (atrybuty klasy) lub definicji metody __init__ klasy (atrybuty instancji). Są one interpretowane przez narzędzia do wyodrębniania, takie jak Docutils, jako „dokumenty celu instrukcji przypisania”. (Jeśli chcesz dowiedzieć się więcej o ciągach dokumentów atrybutów, zapraszamy do samodzielnego zapoznania się z PEP 224. W tym momencie chcemy tylko, abyś był ich świadomy).
- dodatkowe dokumenty, które znajdują się bezpośrednio po innym dokumencie. (Pierwotny pomysł na dodatkowe ciągi dokumentów został zaczerpnięty z PEP 216, który z kolei został później zastąpiony przez PEP 287 – ponownie zapraszamy do zapoznania się z tymi PEP.)
- Dokumenty powinny być ujęte w potrójne podwójne cudzysłowy ("""potrójne podwójne cudzysłowy""")

Python standard – PEP 257

Docstringi jednowierszowe powinny być używane do raczej prostych, oczywistych i krótkich opisów. Powinny zajmować tylko jedną linię i być otoczone potrójnymi podwójnymi cudzysłowami (cytaty zamykające powinny znajdować się w tym samym wierszu, co otwierające, ponieważ pomaga to zachować przejrzystość i elegancję kodu).

- dokumentacja powinna zaczynać się od dużej litery (chyba że identyfikator rozpoczyna zdanie) i kończyć się kropką;
- dokumentacja powinna opisywać efekt segmentu kodu, a nie go opisywać. Innymi słowy, powinien mieć formę rozkazu (np. „Zrób to”, „Zwróć tamto”, „Oblicz to”, „Zamień tamto” itp.), A nie opisu (np. „Czy to”, „Zwraca że”, „Formuje to”, „Rozszerza tamto” itp.).

```
def greeting(name):
    """Take a name and return its replicated form."""
    return name * 2
```

- dokumentacja nie powinna po prostu powtarzać parametrów funkcji lub metody.

```
def my_function(x, y):
    """my_function(x, y) -> list"""
...
```

```
def my_function(x, y):
    """Compute the angles and return a list of coordinates."""
...
```

- Nie używaj pustej linii nad ani pod jednowierszowym dokumentacją, chyba że dokumentujesz klasę, w takim przypadku powinieneś umieścić pustą linię po wszystkich dokumentach, które ją dokumentują.

```
def calculate_tax(x, y):

    """I am a one-line docstring."""

    return (x + y) * 0.25
```

```
def calculate_tax(x, y):
    """I am a one-line docstring."""
    return (x+y) * 0.25
```

Python standard – PEP 257

Docstringi wielowierszowe powinny być stosowane w przypadkach nieoczywistych i bardziej szczegółowych opisach segmentów kodu. Powinny mieć wiersz podsumowania, podobny do tego, jak wygląda jednowierszowy ciąg dokumentów, po którym następuje pusta linia i bardziej rozbudowany opis. Wiersz podsumowania może znajdować się w tym samym wierszu co otwarte potrójne podwójne cudzysłowy lub w następnym wierszu. Apostrofy końcowe należy umieścić w osobnej linii.

- wielowierszowy ciąg dokumentów powinien mieć wcięcie na tym samym poziomie co otwarte cudzysłowy,
- powinieneś wstawić pustą linię po wszystkich wielowierszowych docstringach dokumentujących klasę
- skrypty docstringowe (w sensie samodzielnych programów/pojedynczych plików wykonywalnych) powinny dokumentować funkcję skryptu, składnię wiersza poleceń, zmienne środowiskowe i pliki. Opis powinien być wyważony w taki sposób, aby pomóc nowym użytkownikom zrozumieć użycie skryptu, a także zapewnić szybkie zapoznanie się ze wszystkimi funkcjami programu dla bardziej doświadczonego użytkownika;
- dokumentacja modułu powinna zawierać listę klas, wyjątków i funkcji eksportowanych przez moduł;
- docstrings pakietu (rozumiane jako docstring modułu __init__.py pakietu) powinny zawierać listę modułów i podpakietów wyeksportowanych przez pakiet;
- ciągi dokumentów dla funkcji i metod klas powinny podsumowywać ich zachowanie i dostarczać informacji o argumentach (w tym argumentach opcjonalnych), wartościach, wyjątkach, ograniczeniach itp.
- class docstrings powinny również podsumować jego zachowanie, a także udokumentować publiczne metody i zmienne instancji.

```
def king_creator(name="Greg", ordinal="I", country="Neverland"):
    """Create a king following the article title naming convention.

    Keyword arguments:
    :arg name: the king's name (default: Greg)
    :type name: str
    :arg ordinal: Roman ordinal number (default: I)
    :type ordinal: str
    :arg country: the country ruled (default: Neverland)
    :type country: str
    """
    if name == "Voldemort":
        return "Voldemort is a reserved name."
    ...
```

Python standard – PEP 257

```
class Vehicle:
    """A class to represent a Vehicle.

    Attributes:
    -----------
    vehicle_type: str
        The type of the vehicle, e.g. a car.
    id_number: int
        The vehicle identification number.
    is_autonomous: bool
        self-driving -> True, not self-driving -> False


    Methods:
    --------
    report_location(lon=45.00, lat=90.00)
        Print the vehicle id number and its current location.
        (default longitude=45.00, default latitude=90.00)
    """

    def __init__(self, vehicle_type, id_number, is_autonomous=True):
        """
        Parameters:
        -----------
        vehicle_type: str
            The type of the vehicle, e.g. a car.
        id_number: int
            The vehicle identification number.
        is_autonomous: bool, optional
            self-driving -> True (default), not self-driving -> False
        """

        self.vehicle_type = vehicle_type
        self.id_number = id_number
        self.is_autonomous = is_autonomous
```

```
def report_location(self, id_number, lon=45.00, lat=90.00):
    """
    Print the vehicle id number and its current location.

    Parameters:
    -----------
    id_number: int
        The vehicle identification number.
    lon: float, optional
        The vehicle's current longitude (default is 45.00)
    lat: float, optional
        The vehicle's current latitude (default is 90.00)
    """


...
...
...
```

Python standard – PEP 257

Typy formatowania dokumentów:

- Użyliśmy dwóch różnych formatów docstringu do dokumentowania funkcji king_creator() i klasy Vehicle.
- Pierwszy typ formatowania nazywa się reStructuredText i jest to oficjalny standard dokumentacji Pythona wyjaśniony i opisany w PEP 287.
- Drugi przykład wykorzystuje format NumPy/SciPy docstrings (szczegóły, kliknij tutaj, który jest połączeniem formatu Google docstrings i format reStructuredText.
- Oba typy formatowania są dobre do tworzenia dokumentacji formalnej i oba są obsługiwane przez Sphinx, jeden z najpopularniejszych generatorów dokumentacji Pythona.
- Sphinx to świetne narzędzie do tworzenia dokumentacji dla projektów programistycznych. Używa reStructuredText jako języka znaczników i ma wiele przydatnych funkcji, takich jak obsługa formatu wyjściowego HTML, automatyczne testowanie fragmentów kodu, obszerne odsyłacze i hierarchiczna struktura, która pozwala zdefiniować drzewo dokumentu.

Python standard – PEP 257

Jak udokumentować projekt:

- Dokumentując projekt w Pythonie, w zależności od charakteru projektu (tj. prywatny, współdzielony, publiczny, open source/domena publiczna), należy przede wszystkim zdefiniować jego użytkowników i pomyśleć o ich potrzebach.
- Stworzenie persony użytkownika może się tutaj przydać, ponieważ pomoże Ci określić, w jaki sposób użytkownicy będą korzystać z Twojego projektu.
- Oznacza to, że możesz łatwo poprawić ich wrażenia, zastanawiając się, w jaki sposób zamierzają wykorzystać Twój kod i próbując przewidzieć najczęstsze problemy, na które mogą się natknąć.
- Generalnie projekt powinien zawierać następujące elementy dokumentacji:
    - plik readme, który zawiera krótkie podsumowanie projektu, jego cel i ewentualnie wskazówki dotyczące instalacji;
    - plik example.py, który jest skryptem demonstrującym kilka przykładów wykorzystania projektu;
    - licencja w postaci pliku txt (szczególnie ważna przy projektach Open Source i Public Domain)
    - plik jak wnieść wkład, który zawiera informacje o możliwych sposobach wniesienia wkładu w projekt (projekty współdzielone, open source i projekty należące do domeny publicznej).
- Ponieważ dokumentowanie kodu może być dość wyczerpującym i czasochłonnym zajęciem, zdecydowanie zachęcamy do korzystania z niektórych narzędzi, które mogą pomóc w automatycznym generowaniu dokumentacji w pożądanym formacie oraz radzeniu sobie z aktualizacjami dokumentacji i wersjonowaniem w efektywny i efektywny sposób droga.
- Dostępnych jest wiele narzędzi i zasobów dokumentacji, takich jak Sphinx, o którym już wspominaliśmy, lub bardzo popularny pdoc i wiele innych..

Python standard – PEP 257

Linters:

- Jest to narzędzie, które pomaga pisać kod, ponieważ analizuje go pod kątem wszelkich anomalii stylistycznych i błędów programistycznych na podstawie zestawu predefiniowanych reguł. Innymi słowy, jest to program, który analizuje Twój kod i zgłasza takie problemy, jak błędy strukturalne i składniowe, brak spójności i brak zgodności z najlepszymi praktykami lub wytycznymi dotyczącymi stylu kodu, takimi jak PEP 8. Najpopularniejsze lintery to: Flake8, Pylint , Pyflakes, Pychecker, Mypy i Pycodestyle (wcześniej Pep8) – oficjalne narzędzie Lintera do sprawdzania kodu Pythona pod kątem zgodności z konwencjami PEP 8.
- Fixer to program, który pomaga rozwiązać te problemy i sformatować kod, aby był zgodny z przyjętymi standardami. Najpopularniejsze fixery to: Black, YAPF i autopep8
- Większość edytorów i IDE (np. PyCharm, Spyder, Atom, Sublime Text, Visual Studio, Eclipse + PyDev, VIM lub Thonny) obsługuje lintery, co oznacza, że możesz je uruchamiać w tle podczas pisania kodu. Umożliwia to wykrywanie, podkreślanie i identyfikowanie wielu problematycznych obszarów w kodzie, takich jak literówki, nieprawidłowe tabulatory i wcięcia, wywołania funkcji z niewłaściwą liczbą argumentów, niespójności stylistyczne, niebezpieczne wzorce kodu i wiele innych, a także automatycznie sformatuj swój kod zgodnie z predefiniowaną specyfikacją.

Python standard – PEP 257

Style Guides:

- Przewodnik po stylu służy do zdefiniowania spójnego sposobu pisania kodu. Zazwyczaj jest to wszystko kosmetyczne, co oznacza, że nie zmienia logicznego wyniku kodu. Chociaż niektóre wybory stylistyczne pozwalają uniknąć typowych błędów logicznych.
- Przewodniki po stylach ułatwiają czytanie, utrzymywanie i rozszerzanie kodu.
- PEP 8 zapewnia konwencje kodowania dla kodu Python. Kod Python często stosuje siędo tego przewodnika po stylach. To świetne miejsce na rozpoczęcie, ponieważ jest już dobrze zdefiniowane.
- PEP 257 opisuje konwencje dla dokumentów w języku Python, które są łańcuchami przeznaczonymi do dokumentowania modułów, klas, funkcji i metod. Dodatkową zaletą jest to, że jeśli dokumenty są spójne, istnieją narzędzia do generowania dokumentacji bezpośrednio z kodu.

Linters:

- Lint – historyczny program oryginalnego Uniksa analizujący kod w C pod kątem podejrzanych lub nieprzenośnych instrukcji, pomagający wykryć błędy
- Małe błędy, niespójności stylistyczne i niebezpieczne odwołania występują w kodzie i są rzeczą normalną.
- Lintery pomagają zidentyfikować te obszary problemowe takie jak błędnie wpisane nazwy zmiennych, zapominanie o nawiasie zamykającym, niepoprawne tabulacje w Pythonie, wywoływanie funkcji z niewłaściwą liczbą argumentów itp.
- Większość edytorów i IDE ma możliwość uruchamiania w tych programów w tle podczas pisania.
- Linters analizują kod w celu wykrycia różnych kategorii tzw. Kłaczków (Lint)

Python standard – PEP 257

Popularne kombo linterów:

- **Flake8**: Może wykrywać błędy logiczne i stylistyczne. Dodaje kontrolę stylu i złożoności pycode style do logicznego wykrywania kłaczków PyFlakes. Łączy w sobie następujące lintery:
  - PyFlakes
  - pycodestyle (pep8)
  - Mccabe
- **Pylama**: narzędzie do audytu kodu złożone z dużej liczby linterów i innych narzędzi do analizy kodu. Łączy następujące elementy:
  - pycodestyle (pep8)
  - pydocstyle (pep257)
  - PyFlakes
  - Mccabe
  - Pylint
  - Radon
  - gjslint

Linter	Category	Description
Pylint	Logiczny i Stylistyczny	Sprawdza błędy, próbuje egzekwowaćstandard kodowania, szuka błedówkodu
PyFlakes	Logiczny	Analizuje programy i wykrywa różne błędy
pycodestyle	Stylistyczny	Sprawdza zgodnośćz niektórymi konwencjami stylów w PEP 8
pydocstyle	Stylistyczny	Sprawdza zgodnośćz konwencjami dokumentacyjnymi Python
Bandit	Logiczny	Analizuje kod w celu znalezienia typowych problemów bezpieczeństwa
MyPy	Logiczny	Sprawdza opcjonalnie wymuszone typy statyczne

Python standard – PEP 257

Dostęp do docstringu:

- Robimy to za pomocą atrybutu Pythona __doc__ – jeśli po zdefiniowaniu funkcji/modułu/klasy/metody występują jakieś literały łańcuchowe, to są one powiązane z obiektem jako jego atrybut __doc__, a ten atrybut dostarcza dokumentację tego obiektu .
- Ale jest też inny sposób na dostęp do ciągów dokumentacji – możesz użyć funkcji help().

```
def my_fun(a, b):
    """The summary line goes here.

    A more elaborate description of the function.

    Parameters:
    a: int (description)
    b: int (description)

    Returns:
    int: Description of the return value.
    """
    return a*b

print(my_fun.__doc__)
```

6