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

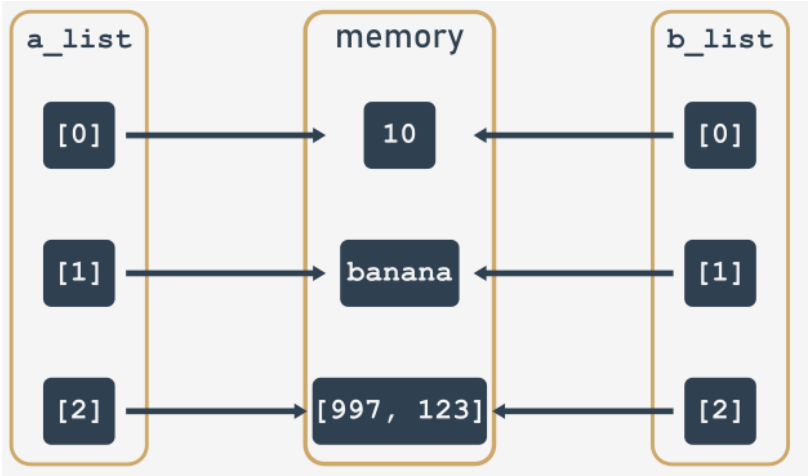
- Podczas przetwarzania danych dojdziesz do punktu, w którym możesz chcieć mieć odrębne kopie obiektów, które możesz modyfikować bez jednoczesnej automatycznej modyfikacji oryginału.
- Spoglądając na kod widzimy że jego intencją jest:
  - zrobić prawdziwą, niezależną kopię a_list (nie tylko odniesienie 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

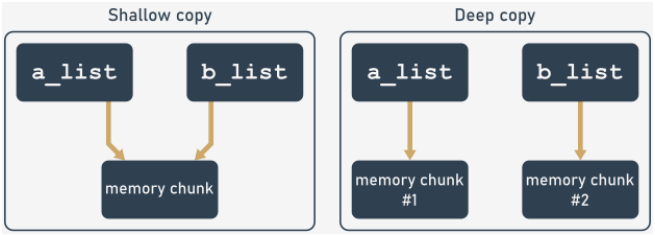
- W poniższym przykładzie utworzymy nową zagnieżdżoną listę, a następnie powierzchownie ją skopiujemy za pomocą funkcji fabrycznej list()
- Oznacza to, że ys będzie teraz nowym i niezależnym obiektem o tej samej zawartości co xs. Możesz to sprawdzić, sprawdzając oba obiekty
- Aby potwierdzić, że ys naprawdę 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)
```

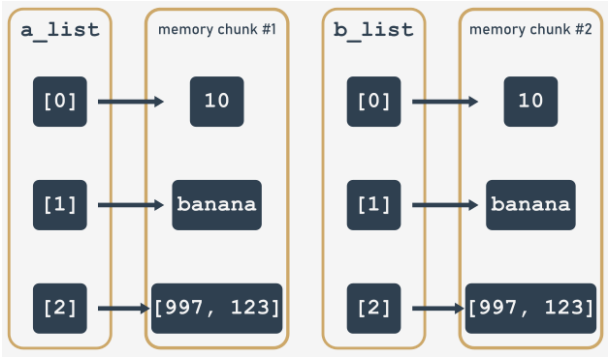
Shallow and deep copy – głęboka kopia

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.

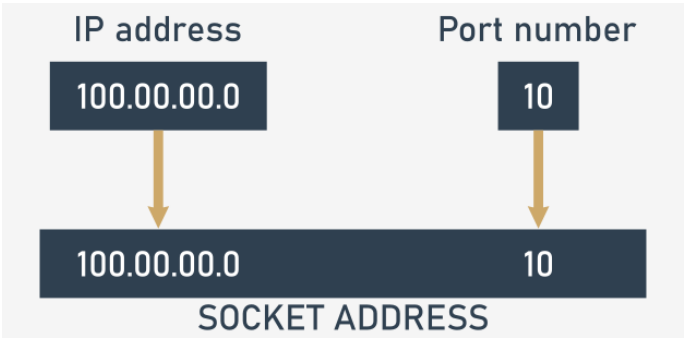
Domains, addresses, ports, protocols and services - wstęp

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:

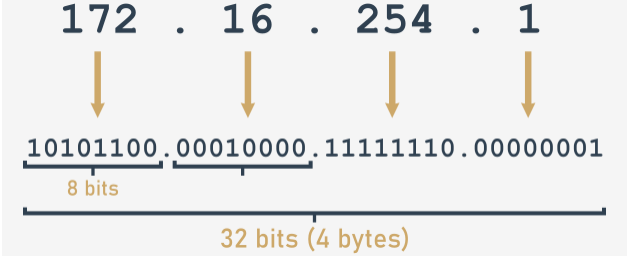
- Adres IP (dokładniej: adres IP4) to 32-bitowa wartość służąca do identyfikacji komputerów podłączonych do dowolnej sieci TCP/IP.
- Wartość jest zwykle przedstawiana jako cztery liczby z zakresu 0..255 (czyli o długości ośmiu bitów) sprzężone z kropkami (np. 87.98.239.87).
- Istnieje również nowszy 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:

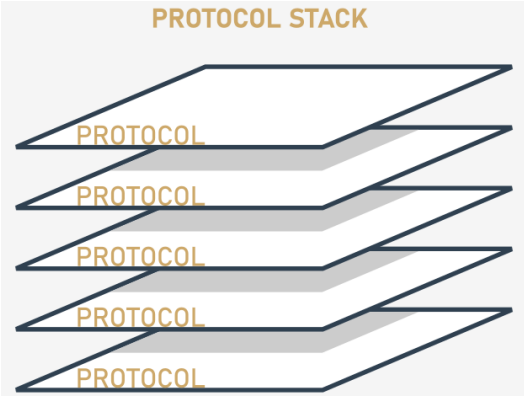
Protokół to znormalizowany zestaw reguł umożliwiający procesom wzajemną komunikację. Można powiedzieć, że protokół to swego rodzaju sieciowy savoir-vivre określający zasady zachowania wszystkich uczestników.



Domains, addresses, ports, protocols and services - wstęp

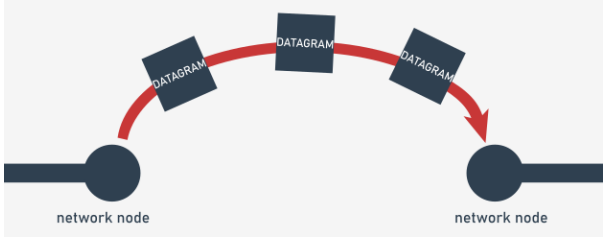
Stos protokołów:

- 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:

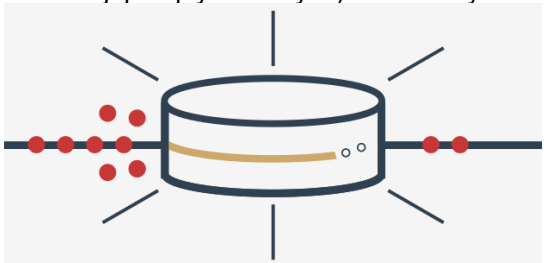
- IP (Internetwork Protocol) jest jedną z najniższych części stosu protokołów TCP/IP. Jego funkcjonalność jest bardzo prosta - jest w stanie przesłać pakiet danych (datagram) pomiędzy dwoma węzłami sieci.
- IP jest bardzo zawodnym protokołem. Nie gwarantuje, że:
    - dowolny z wysłanych datagramów dotrze 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, DHCP itp.)

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.

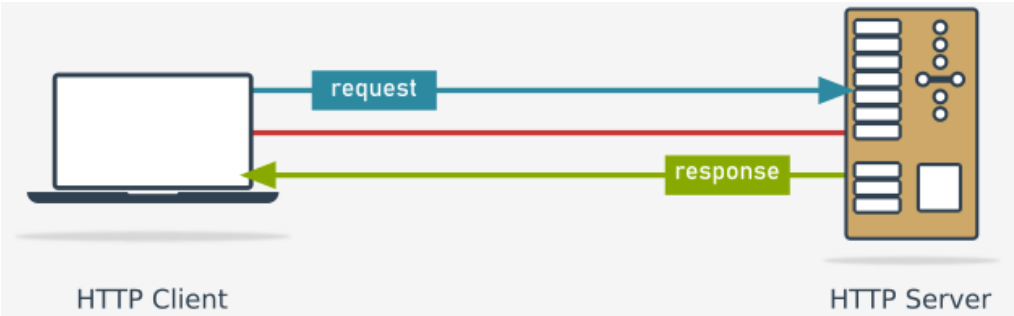
Sockets in Python - wstęp

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

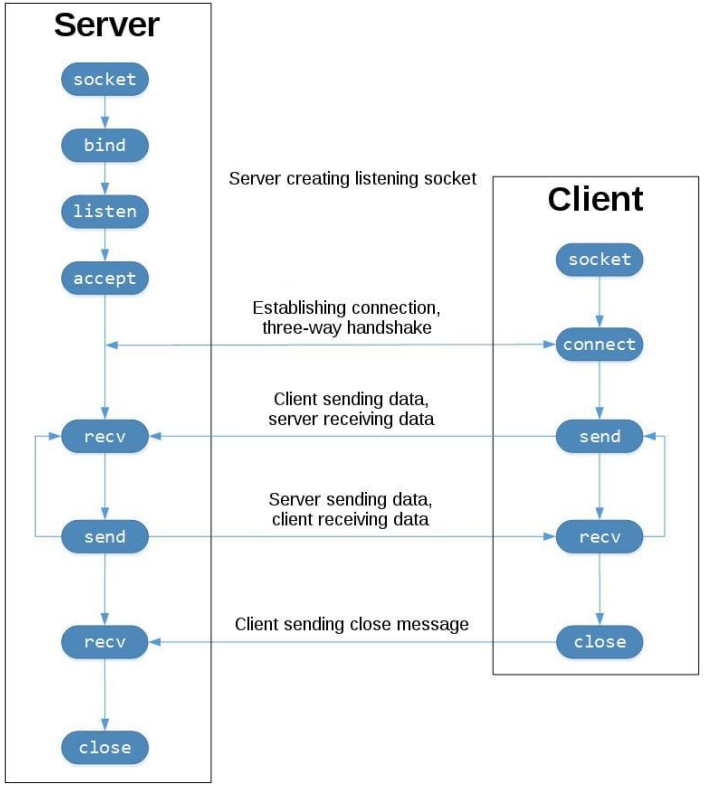
Podstawowe funkcje i metody interfejsu API gniazda w tym module to:

- 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)
```

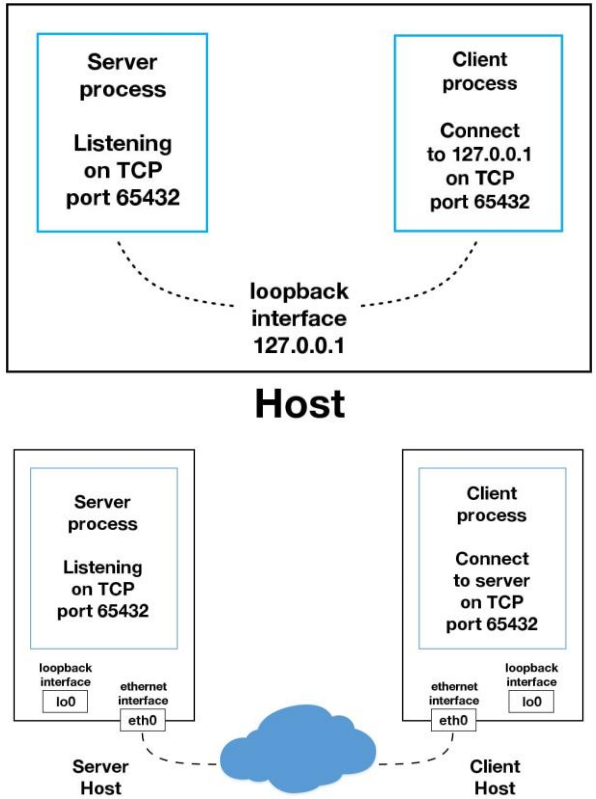
Client

```
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, który udostępniliśmy w edytorze.
- 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 .

```
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, z)


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

4