**50 zadań egzaminacyjnych z Pythona**

1. Lista niepowtarzalnych elementów.

Korzystając z podanej listy A stwórz listę B zawierającą tylko unikalne elementy z listy A.

Sposób 1:

A = [ 1,2,3,3,2,1,1,2,3]

 B = []

for i in A:

    if i not in B:

        B.append(i)

Wyjaśnienie:

Przy użyciu pętli FOR bierzemy po kolei każdy element z listy A. Jeżeli nie znajduje się w liście B, to wykonywana If i not in B, co powoduje, że wykonana jest linijka B.append(i).

Proste, łatwe i przyjemne:

Sposób 2:

A = [ 1,2,3,3,2,1,1,2,3]

B = list(set(A))

print(B)

Set jest strukturą zawartą w Pythonie tylko unikalnych wystąpień elementów. Co tłumaczy się samo przez się. Na końcu musimy pamiętać o zamianie SETA na listę, ponieważ jest inną strukturą danych, ale dla nas to zostanie załatwione poprzez prostą linijkę kodu.

Celem takiego zadania jest sprawdzenie czy potrafimy to rozwiązać na dwa sposoby. 1 🡺 algorytmiczny, oraz 2 🡺 wykorzystujący właściwości SETu, pozwalający na rozwiązanie tego zadania sprytnie, w jednej linijce.

Zadanie domowe:

# Sprawdź i wypisz ile unikatowych elementów znajduje się w liście A.

A = [1, 2, 3, 4, 3, 2, 3, 4, 5, 6, 7, 5, 43, 4, 6]

B = list(set(A))

print(len(B))

1. Co się stanie po wykonaniu poniższego kodu?

Rozchodzi się o to, że stringi są niemutowalne, tj. nie da się ich modyfikować, tak jak list i tupli, dlatego kod, który wrzuciliśmy do bębna zwróci nam jako błąd:

a = "abcdefg"

print(a[1])

a[1] = 'X'

Jedyny sposób, aby wykoanć to ćwiczenie to zamiana stringa na listę, a następnie podmianka elementu a1 na nas interesujący:

a = "abcdefg"

print(a[1])

#a[1] = 'X'

a\_lista = lista(a) # worzymy nową listę, która będzie zawierała dane ze stringa A

a\_lista[1]= 'X'

a = "".join(a\_lista) # metoda .join na pustym stringu powoduje, że elementy podanego ciągu stają się częścią tej listy

print(a)

Jeżeli pomiędzy cudzysłowami damy jakiś separator, np. myślnik, to nasz string będzie wyglądał tak:

a-X-c-d-e-f-g

**3. Różnica między listą a tuplą**

Napisz kod, który zaprezentuje najważniejsze różnice między list a tuple.

L = [1,2,3,4,True,(1,2)] # Lista jak to w Pythonie może przechowywać różne wartości, liczby, wartości logiczne, albo Tuple właśnie

T = ( 3,4,5,False, ['x','y']) # Tuple tak samo jak listy mogą również przechowywać różne typy wartości tak jak listy.

 L[2] = 'trzy' # w tej linijce chcę zamienić element listy pod numerem drugi, uda się

print(L)

==>[1, 2, 'trzy', 4, True, (1, 2)]

T[2] = 'sześć' # type error, że Tupla nie wspiera przypisywania elementów.

Tak więc:

Różnica pierwsza 🡺 Listy zapisujemy w nawiasach kwadratowych, zaś krotki(tuple) w okrągłych

Różnica druga 🡺 Listy są mutowalne, zaś krotki nie.

**Pytanie 4. Jakiej struktury danych użyłbyś do zamodelowania: szafki, która ma 3 szuflady, a w każdej z nich 3 przegórdki**

Stwórz taki model i umieść stringa "długopis"

W środkowej przegródce środkowej szuflady

# w teorii nadają się i tupla i lista, ale umieszczenie długopisu w szafce to tak naprawdę modyfikacja, więc zostaje nam lista.

szafka = [[[],[],[]],[[],[],[]],[[],[],[]]] # --> ale to skomplikowane :U

szafka[1][1] = 'długopis'# ==> Teraz dajemy znać Pythonowi, że chcemy sięgnąć do szuflady nr 2 [1],

                      #a także do drugiej przegródki w niej[1]

Stwórz listę A, w której znajdują się dwie listy: w pierwszej umieść integery: 1 i 2, w drugiej 3 i 4.

Zmodyfikuj listę A tak, aby w zagnieżdżonej liście zamiast integera 3 był string 'trzy'.

Wydrukuj listę A.

A = [[1,2],[3,4]]

A[1][0] = 'trzy'

print(A)

**5. Zagnieżdżone struktury w kodzie.**

Takie zadania najlepiej rozwiązywać po kolei, krok po kroku. Wtedy staną się banalnie proste.

Z poniższej listy wypisz stringa „Schowany”.

L = [[34, False],[0],[('abc', 123),{'a': 1, 'x':(True, 'schowany', 5)}]]

# mamy listę L w której znajduje się kilka list w której znajdują się ciekawe rzeczy. W jednej z list znajduje się tupla i słownik,

# wewnątrz słownika znajduje się tupla, w której jest interesujący nas element.

print(L[2]) # wydzialamy sobie dużą listę w której jest interesujący nas plik.

🡺[('abc', 123), {'a': 1, 'x': (True, 'schowany', 5)}]

print(L[2][1]) 🡺 {'a': 1, 'x': (True, 'schowany', 5)} 🡺 mamy teraz dwie tuple, więc musimy wybrać tą, w której jest nasza zguba, a więc ‘x’

print(L[2][1]['x']) 🡺 (True, 'schowany', 5)

print(L[2][1]['x'][1]) 🡺 schowany

# Z poniższej tupli wypisz słowo: 'wróbel'.

K = (('król', {2:'królewna', 1: ['córka', 'wróbel']},'5'),('żółw', 'wiewiórka'))

print(K[0]) 🡺 ('król', {2: 'królewna', 1: ['córka', 'wróbel']}, '5')

print(K[0][1]) 🡺 {2: 'królewna', 1: ['córka', 'wróbel']}

print(K[0][1][1]) 🡺  ['córka', 'wróbel']

print(K[0][1][1][1]) 🡺 wróbel

**Pytanie 6.**

Jakiej struktury danych użyłbyś do zapisania numerów telefonów wszystkich klientów firmy i odpowiadających im nazwisk. Wybierz strukturę tak, aby sprawdzenie właściciela numeru telefonu nie zajmowało dużo czasu.

Następnie stwórz przykładową strukturę przechowywującą poniższe informacje:

123456789 Jan Kot

999888777 Anna Lis

777744445 Zbyszek Bobkowski

123455543 Jan Kot

Musimy więc stworzyć strukturę, która powiąże unikalne dane, z ich nie unikalnymi odpowiednikami.

Dobrze myślałem, słownik!

D = {123456789:'Jan Kot', 999888777:'Anna Lis', 777744445:'Zbyszek Bobkowski', 123455543:'Jan Kot'}

print(D) 🡺 {123456789: 'Jan Kot', 999888777: 'Anna Lis', 777744445: 'Zbyszek Bobkowski', 123455543: 'Jan Kot'}

print (D[123456789]) 🡺 Jan Kot

Pytanie 7.

Które z poniższych słowników zostały utworzone niepoprawnie:

Kluczem słownika może być liczba, wartość liczbowa, tupla lub string. Słownik nie może być tak samo jak lista. Kluczem jest zrozumienie tablic haszujących.

A = {1: 1, 2: 4, 3: 9}

B = {'imie': 'Anna', 'nazwisko': 'Kowalska'}

# C = {[4, 5]: [16, 25]}    # lista jako element mutowalny nie może byc kluczem słownika!

D = {(4, 5): [16, 25]}

# E = {{1:2}: 'jeden\_dwa'}  # słownik jako element mutowalny równiez nie może być kluczem!

**Pytanie 8.**

Co zostanie wypisane w wyniku wykonania poniższego kodu?

D={1:'Ala',2:'ma',3:'kota.'} # słownik, wartości liczbowe a kluczami są ciągi

for key in D:

    print(D[key])

Ala

ma

kota.

W Pythonie od 3.7 w górę wynik będzie zawsze taki sam. Natomiast w starszych Pythonach wynik będzie za każdym razem inny, ponieważ tablice haszujące. W dawnych pythonach słowniki działy losowo.

W pythonach w wersji 2 – do 3,5 słownik nie trzyma kolejności

W 3,6 trzyma, ale twórcy tego nie gwarantują,

W starszych wersjach Pythona, jeżeli chcieliśmy mieć słownik, który zawsze przekaże dane w takiej samej kolejności, stosowało się **OrderedDict**(musiała być ona zaimplementowana).

**Pytanie 9.**

Dla podanego ciągu stwórz słownik przechowujący informację ile razy dana litera wystąpiła w stringu.

x = "myszydokazujągdykotanieczują"

D = {}

for litera in x:

    if litera not in D.keys(): #keys jest konieczne, jeżeli je usuniemy, to Python będzie sprawdzał czy litera jest jednym z kluczy

        D[litera] = 1

    else:

        D[litera] += 1 #  D[litera] = D[litera] + 1

print(D) ==>{'m': 1, 'y': 3, 's': 1, 'z': 3, 'd': 2, 'o': 2, 'k': 2, 'a': 2, 'u': 2, 'j': 2, 'ą': 2, 'g': 1, 't': 1, 'n': 1, 'i': 1, 'e': 1, 'c': 1}

Celem tego zadania jest sprawdzenie czy potrafimy radzić sobie ze słownikiem i czy umiemy sprawdzić czy jest już kluczem słownika, czy znamy metodę Keys, i czy wiemy co się stanie jak ją usuniemy

Ćwiczenie:

Do podanego słownika S napisz kod sprawdzający czy liczba 7430 znajduje się wśród kluczy słownika

Jeśli tak to True,

Jeśli nie to False.

S = {x:x+1 for x in range(10000) if x%23 == 0}

if 7430 in S.keys():

    print(True)

else:

    print(False)

**Pytanie 10. Wartości słownika**

Korzystając ze słownika stworzonego w poprzednim zadaniu sprawdź czy któraś z liter wystąpiła w stringu dokładnie 4 razy. Jeśli tak – wypisz True, jeśli nie - False.

litery = {'m': 1, 'y': 3, 's': 1, 'z': 3, 'd': 2, 'o': 2, 'k': 2, 'a': 2, 'u': 2, 'j': 2, 'ą': 2, 'g': 1, 't': 1, 'n': 1, 'i': 1, 'e': 1, 'c': 1}

print(list(litery.values())) # dlaczego znalazło się tu list? A zobaczmy

==> [1, 3, 1, 3, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1]

print(litery.values()) ==> dict\_values([1, 3, 1, 3, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1])

Widać to bezsprzecznie,ale co jeżeli słownik miałby 1000 elementów albo więcej? TO już takie łatwe by nie było Ale i na to jest rozwiązanie

if 4 in litery.values():

    print(True)

else:

    print(False)

print(True if 4 in litery.values()else False)

Zadanie:

Korzystając ze słownika pensje, zawierającego informację o wynagrodzeniach w firmie, wydrukuj ich sumę:

pensje = {'ksiegowa': 5000, 'kierowca': 4500, 'recepcjonistka': 4000}

# twój kod tutaj

print(sum(pensje.values()))

**Pytanie 11. Odwracanie listy**

Na podstawie listy języki stwórz listę języki odwrócone zawierającą elementy listy w odwróconej kolejności.

jezyki = ['Python', 'Java', 'C#', 'Ruby']

Sposób 1

jezyki\_odwrócone = jezyki[::-1]

print(jezyki\_odwrócone)

Sposób 2

jezyki.reverse() # ==> reverse() modyfikuje obecną listę i nie tworzy nowej

jezyki\_odwrocone = jezyki

print(jezyki\_odwrocone)

Sposób 3

jezyki\_odwrocone = list(reversed(jezyki)) # modyfikuje obecną listę i tworzy nową.

print(jezyki\_odwrocone)

Sposób 4

jezyki\_odwrocone = []

for jezyk in jezyki:

    jezyki\_odwrocone.insert(0,jezyk) # Jak działa metoda insert? Na podany indeks danej liczny wsadza podaną wartość, tak więc, każdy przeczytany będzie umieszczany na liście,

        #każdy przeczytany język będzie umieszczony na początku listy ==> tak odwróci się w ten sposób

print(jezyki\_odwrocone)

\

Pytanie 12. Palindrom.

Napisz funkcję sprawdzającą czy podane słowo jest palindromem.

Można to zrobić na 3 sposoby.

1.

def palindrom(slowo):

    slowo\_odwrocone = slowo[::-1]

    if slowo == slowo\_odwrocone:

        return True

    else:

        return False

2. To samo ale w jednej linijce.

def palindrom(slowo):

    return True if slowo == slowo[::-1] else False

3.Najbardziej złożony

def palindrom(slowo):

    poczatek = 0

    koniec = len(slowo) -1

    while poczatek <= koniec:

        if slowo[poczatek] != slowo[koniec]:

            return False

        else:

            początek += 1

            koniec -= 1

    return True

**Pytanie 13. Range**

Stwórz dwie listy. A zawierającą liczby od 1 do 10 i B zawierającą liczbę od 100 do 1. W obu przypadkach możesz napisać tylko jedną linijkę kodu

W tym zadaniu kluczowa jest funkcja Range oraz jej trzy parametry: start, stop, krok.

**Start** to miejsce gdzie zaczynamy, jeżeli go nie podamy to domyślnie będzie to 0

**Stop** to liczba, która jako pierwsza nie wchodzi w wydrukowany zakres(dla 10 będzie to 9 itd.)

Krok – mówi nam o ile liczb chcemy przeskakiwać do przodu lub do tyłu w naszej sekwencji

Tak więc:

    A = list(range(1,11))

    print(A) 🡺[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

    B = list(range(1,101,3))[::-1]

    print(B) 🡺[100, 97,[...] 4, 1]

Lub:

 C = list(range(100,0,-3))

    print(C) 🡺[100, 97,[...] 4, 1]

Czym jest Range i jaką zwraca wartość?

To zależy od generacji Pythona.

W drugiej generacji Range był funkcją i zwracał listę.

W trzeciej generacji Range nie jest już funkcją, a typem danych. W nawiasach nie są to argumenty funkcji, a parametry konstruktora danego typu danych, a Range zwraca sekwencję liczb.

Jak istotne to różnice?

Wydajność. W 2.x jeżeli tworzyliśmy funkcję, to musiała ona powstać i pozostać w pamięci komputera. W trójce zwraca nam konkretną liczbę, a lista nie siedzi w pamięci, a jest na bieżąco generowana.

**14. Slicy.**

Wypisz pierwsze 5 elementów z listy L

Wypisz co drugą literę stringa S zaczynając od ostatniej o cofając się do początku

L = [11,22,33,44,55,66,77,88,99,1010]

s = 'Roztropniez to waszmosc wykoncypowal'

L[0:5] 🡺 [11, 22, 33, 44, 55]

s[-1::-2] 🡺 lwpcoy smswo enoto'

Tworzenie slajsów to bardzo ważna umiejętność każdego programisty PYTHONA!

Wypisz co trzecią literę stringa a, zaczynając od przedostatniej i cofając się do początku.

a = '!ooe&sj7?czaa()lmxuo,t2fa^4rtngk'

print(a[-2::-3])

**15. Slicy praktycznie. Sprawdzanie nazwy pliku.**

Napisz funkcje która sprawdzi czy podany string zaczyna się łowem python i kończy rozszerzeniem”.py”

a = "python\_moj\_kod.py"

b = "python\_notatki.docx"

def x(nazwa\_pliku):

    if nazwa\_pliku[0:6] == 'python' and nazwa\_pliku[-3:] == '.py':

        return True

    else:

        return False

Napisz kod, który przy użyciu sliceów wypisze SZEŚĆ ostatnich znaków ze stringa z.

z = "In the face of ambiguity, refuse the temptation to guess."

print(z[-6:])

**16. Enumerate.**

Wypisz podaną listę imion przed każdym dodając kolejny numer. Zacznij od 1.

Zrobimy to na dwa sposoby:

imiona = ['Adam', 'Bob', 'Stanisław', 'Maria', 'Mikojaj']

#1

num = 1 # bo mamy zacząć od numeru jeden

for imie in imiona:

    print(num, imie)

    num += 1

#2 Użyjemy wbudowanej w Pythona funkcji enumerate.

for num, imie in enumerate(imiona,1): # jedynka po przecinku to informacja dla Pythona, od jakiej liczby ma zacząć

    print(num, imie)

# enumerate to funkcja, która przypisuje kolejne numery poszczególnym elementom funkcji

1 Adam 2 Bob 3 Stanisław 4 Maria 5 Mikojaj

# Wypisz zawartość listy A dodając przed każdym elementem kolejny numer.

# Zacznij numerowanie od 0.

A = [1, 1, 4, 9]

for num, new in enumerate(A):

    print(num,new)

**17. Min i Max.**

Znajdź różnicę między największą, a najmniejszą wartością na poniższej liście. Zadbaj o to aby rozwiązanie było efektywne.

Znamy 3 sposoby. Od najlepszego do najgorszego.

#1

A = [4,5,7,-3,2,8,- 10,15]

print(max(A) - min(A))

#2

A = [4,5,7,-3,2,8, -10,15]

najmniejsza = A[0]

najwieksza = A[0]

for liczba in A:

    if liczba < najmniejsza:

        najmniejsza = liczba

    elif liczba > najwieksza:

        najwieksza = liczba

print(najwieksza - najmniejsza)

#3

A = [4,5,7,-3,2,8, -10,15]

A = sorted(A)

print(A[-1] - A[0])

# Sprawdź i wypisz (True lub False) czy największy element na liście A jest większy niż liczba 99.

A = [x\*\*2 + 3 for x in range(10)]

if max(A) > 99:

    print(True)

else:

    print(False)

**18. Podzielność.**

Napisz funkcję, która będzie pobierała dwie liczby i będzie sprawdzać czy pierwsza z nich jest podzielna przez drugą

Możńa to zrobić na dwa razy.

Przez modulo, oraz klasycznie:

Modulo:

def dzielenie(a,b):

    return(a % b == 0)

i klasycznie:

def dzielenie(a,b):

    if a / b:

        return True

    else:

        return False

**19. Podzielność.**

Wyjaśnij jak działa poniższa funkcja.

Wyjaśnij skąd wzięły się wyniki przez poszczególne wywołania funkcji.

def dodaj\_do\_listy(n, lista=[]):  # lista=[] - argument domyślny funkcji

    lista.append(n)               # dodaj n do końca listy lista

    print(lista)

dodaj\_do\_listy(1)

dodaj\_do\_listy(2,[4,5])

dodaj\_do\_listy(3)

Co to oznacza, że funkcja jest argumentem domyślnym funkcji?

… , lista=[]):  🡪 Ten zapis oznacza, że możemy wywołać tą funkcje podając jeden parametr, jak i obydwa parametry. Jeżeli wywołamy ją podając tylko jeden parametr, wywołane zostanie to co zostało zdefiniowane jako domyślne.

Prościej?:

def drukuj\_imie(imie = "Bob"):

    print(imie)

Mamy bardzo prostą funkcję, która pobiera imię, a następnie je drukuje. Jeżeli natomiast nie podamy parametru imie, zamiast niego zostanie podane imię „Bob”

drukuj\_imie("Wojtek") 🡺 tu dostaniemy Wojtka

drukuj\_imie() 🡺 A tutaj Boba

A więc rozwiążmy nasze zadanie:

def dodaj\_do\_listy(n, lista=[]):

    lista.append(n)

    print(lista)

dodaj\_do\_listy(1) 🡪

Tutaj dostaniemy 1 ponieważ nie podaliśmy parametru drugiego, to dostaniemy domyślny

dodaj\_do\_listy(2,[4,5]) 🡪

Tutaj wychodzi na to, że na koniec listy 4,5 dostaniemy jeszcze parametr 2,

A więc [4,5,2]

dodaj\_do\_listy(3) 🡪 Tutaj dostaniemy tylko 3?

Niee.

Zamiast samej trójki dostaniemy [1,3]

Dlaczego?

Ten rezultat jest wynikiem działania bardzo specyficznej właściwości – argumentów domyślnych w Pythonie.

Argumenty domyślne tworzone są tylko RAZ! A więc, za pierwszym razem gdzieś w pamięci zachowała się informacja o powstaniu pustej listy.Skoro ta lista tworzona jest tylko raz, to w momencie kiedy wykonaliśmy naszą funkcję pierwszy raz, ta rzeczona jedynka już w niej pozostała. Dlatego kiedy wywołaliśmy (pustą) funkcję po raz trzeci, otrzymaliśmy wynik [1,3] ponieważ nasza jedynka była już wcześniej zachowana w pamięci.

Dlaczego nie otrzymaliśmy jej też w drugim wywołaniu funkcji?

Ponieważ w drugim wywołaniu funkcji podaliśmy już argument listy, który spowodował, że ów domyślny argument się nie zaktywizował.

Właściwość ta jest dość podstępna, dlatego należy ją bezwzględnie znać.

**20. List Comprahension(wyrażenia listowe):**

LC to bardzo popularny mechanizm Pythona, pozwalający tworzyć listy w jednej linijce. Są one świetną alternatywą dla tworzenia list przy pomocy pętli FOR, która to wymaga przecież kilku linijek tekstu.

# Pytanie 20 - co otrzymamy w wyniku wydrukowania zawartości poniższych zmiennych?

L = [1,2,3,4,5,6]

L1 = [x for x in range(5)]

L2 = [x\*\*2 for x in L]

L3 = [x for x in L if x % 2 == 0]

L4 = ['Parzysta' if x%2 == 0 else 'Nieparzysta' for x in range(5)]

L5 = [(x, x+10) for x in L]

D1 = {x:x % 2 == 0 for x in L}

print(L1)

print(L2)

print(L3)

print(L4)

print(L5)

print(D1)

Zacznijmy od omówienia pierwszego wyrażenia:

L1: Nawiasy kwadratowe na początku i końcu mówią nam, że będziemy tworzyć listę. Aby je lepiej zrozumieć warto czytać te listy od tyłu. W tym przypadku widzimy, że dostaniemy listę pięcioznakową o takim też zasięgu. Jako, że w IT zaczynamy od zera będzie to : [0,1,2,3,4]

L2: Może to być też taka sekwencja. W tym wypadku otrzymamy każdy argument z listy podniesiony do potęgi.

L3: W tym przypadku otrzymamy listę liczb, które są podzielne przez dwa, a więc [2,4,6]

L4: Tutaj otrzymamy, aż dwie listy. Jeżeli liczba będzie parzysta to trafi do 'Parzysta' , jeżeli nie to trafi do tej drugiej grupy.

L5: Tutaj otrzymamy dwuelementowe tuple. Pierwszą liczbą w tupli będzie liczba z listy, a drugą zaś liczba ta powiększona o 10

D1: W przypadku słownika otrzymamy pary klucz:wartość, gdzie kluczem są oczywiście liczby z listy L, a wartościami True or False w zależności od tego czy liczba ta jest podzielna przez 2.

**21. ‘is’ vs == ( IS vs Operator porównawczy)**

# Pytanie 21: co wydrukuje się w wyniku wykonania poniższego kodu?

print(1 == True) # == to operator porównania wartości

print(1 is True) # is to operator porównania identyczności/tożsamości

**W pierwszym przypadku będzie True, ponieważ w języka programowania boolean TRUE jest równoznaczny z jedynką, a FALSE jest równoznaczne z zerem.**

**Druga linijka to oczywiście False, ponieważ jedynka NIE JEST tym samym co True. Drobna, acz istotna różnica. Jedynka jest INTEM, a True jest Booleanem.**

print(id(1), id(1), id(True))

**Tutaj dostaniemy wyjaśnienie**

print(2 \*\* 3 == 10 - 2)

**To będzie równe, bo dwa do potęgi 3 to 8, tak samo jak 10-2**

**A jak operatory porównania działają na listach?**

A = [1,2,3]                    # stworzenie dwóch list o identycznej zawartośc

B = [1,2,3]                    # i przypisanych do innych zmiennych A i B

print(A == B)

Mamy dwie identyczne listy. One będą sobie równe ponieważ ich zawartość jest taka sama. JEDNOCZEŚNIE NIE SĄ IDENTYCZNE. Są obiektami mutowalnymi, możemy je więc zmieniać.

Więc takie wyrażenie, da nam FALSE.

print(A is B)

A co z ciągami(stringami?)

a = 'kotek'

b = 'kotek'

print(a == b)

To będzie TRUE, bo przecież są takie same.

A co jeżeli porównamy ich identyczność?

print(a is b)

Również otrzymamy TRUE, ponieważ Stringi są niemutowalne. W pamięci jest tylko jeden string o takiej zawartości i możemy do niego przyporządkować do niego setki zmiennych i nic to nie zmieni.

**22. False is False is False.(Porówania łańcuchowe)**

# Pytanie 22 - co zostanie wydrukowane w wyniku wykonania poniższego kodu?

print(False is False)

print(True is False)

print(False is False is False) # (False is False) and (False is False) -> (True) and (True) -> True

print(1 < 3 == 5)              # (1 < 3) and (3 == 5) -> (True) and (False) -> False

Wszystkie porównania łańcuchowe w Pythonie traktowane są wg tego samego schematu:

 porównanie rozbijane jest na dwuelementowe podgrupy połączone operatorem 'and'. Przykładowo, porównanie czteroelementowe zostanie potraktowane następująco:

**A is B == C > D  -> (A is B) and (B == C) and (C > D)**

Analogicznie postępujemy dla pięciu i więcej elementów.

**23. Lambda.**

Czym jest Lambda? Napisz kod ja wykorzystuący.

Nazywana również funkcją anonimową. Są to krótkie funkcje, które nie potrzebują i nie posiadają nazwy. Są wykorzystywane w miejscach gdzie potrzebujemy krótkiej, jednorazowej funkcji. Są często stosowane jako argumenty, przekazujące innym funkcjom dane.

lambda argumnent : wyrażenie(które coś robi z argumentem)

lambda x : x+2 🡪 Taka lambda będzie pobierała x i zwracała X+2

A teraz to samo, tylko klasycznie:

def iks(x):

    return x+2

L = [('Anna',82), ('Robert',33), ('Artur',40), ('Feliks',56)]

# W poniższej linijce funkcja sorted pobiera sekwencję danych do posortowania i klucz, po którym będzie sortować.

# Sekwencją jest lista L, a kluczem lambda, która dla kolejnego elementu listy L (czyli tupli) zwraca drugi element danej tupli.

L\_posortowana = sorted(L, key = lambda x:x[1]) # ==> Jak wpiszemy tam zero, to nasza funkcja będzie porównywać od stringów

print(L\_posortowana)

🡺 [('Robert', 33), ('Artur', 40), ('Feliks', 56), ('Anna', 82)]

# Posortuj podaną listę stringów jako kryterium sortowania przyjmując ostatnią literę każdego stringa.

S = ['Anna', 'Robert', 'Artur', 'Feliks']

S\_posortowana = sorted(S, key = lambda x:x[-1])

**24. Kopiowanie list.**

Co zostanie wypisane w wyniku wykonania poniższego kodu?

A = [1,2,3,4,5]

B = A

C = A[:]

B[0] = 111

print(B) # [111,2,3,4,5]

print(A) # [111,2,3,4,5] A dlaczemu? Lista się już zdążyła zmienić

print(C) # [1,2,3,4,5]

A dlaczemu? Pod zmienną A nie ma listy, a jedynie referencje do niej. Wskaźnik, że gdzieś w pamięci jest taka lista i możemy do niej sięgnąć przy użyciu zmiennej A. W drugim przypadku więc przypisujemy referencję z A do B,a więc wskazują na nią teraz dwie zmienne, A i B. Dlatego więc wydrukowanie B i A dało identyczny rezultat.

Co się więc stało z listą C? do zmiennej C nie przypisaliśmy listy. Użycie slice w środku, którego mamy dwukropek, oznacza, że do zmiennej C, chcemy przypisać ZAWARTOŚĆ listy A. Dlatego więc wynik jest taki, a nie inny. Podobny efekt można by uzyskać,wrzucając kod **C = list(A)**

**Zadanie**:

Stwórz kopię **wartości**poniższej listy L i przypisz ją do zmiennej Ka następnie:

- zmodyfikuj wartość pod indeksem **1** w liście L tak, aby wynosiła **'jeden'**.

- zmodyfikuj wartość pod indeksem **1** w liście K tak, aby wynosiła **'JEDEN'**.

Wydrukuj listę L, a w kolejnej linijce listę K.

L = [0, 1, 2, 3]

**Rozwiązanie**:

K = L[:] - tworzenie kopii **wartości**listy L. Można by również napisać K = list(L) - zadziałałoby tak samo.

L[1] = 'jeden'    - wpisanie stringa '**jeden**' pod indeks **1** na liście L

K[1] = 'JEDEN'  - wpisanie stringa '**JEDEN**' pod indeks **1** na liście K

Wydrukowanie obu list w osobnych linijkach. 🡺 print(L) print(K)

**25. Zmienne lokalne i globalne.**

Co zostanie wypisane w wyniku wykonania poniższego kodu?

x = 10

def f():

    x = 11

    print(x)

     oraz

f()

print(x)

W pierwszym przypadku będzie to 11, natomiast w drugim 10. Dlaczego?

Ponieważ pierwsza zmienna x = 10, to zmienna globalna, znajdująca się poza ciałem naszej funkcji. Oznacza to, że jest dostępna dla nas z poziomu całego pliku Pythona.

Druga natomiast jest zmienną lokalną i działa tylko wewnątrz tej funkcji. Zaczyna działać w momencie startu funkcji, a kiedy ona się skończy, to ta funkcja znika.

Jest również ciekawy sposób, aby pokazać jak to działa. Modyfikujemy nasze funkcyję:

x = 10

def f():

    x = 11

y = 12

    print(x,y)

     oraz

f()

print(x,y)

W tym wypadku dostaniemy informację, że Y nie jest zdefiniowana. To oznacza, że została usunięta z pamięci, i dla tego printa poniżej funkcji ona już nie istnieje.

A teraz podnieśmy poziom trudności i stwórzmy funkcję, która zmodyfikuje zmienną globalną. Można to zrobić całkiem łatwo, z użyciem słowa kluczowego global

x = 10

def f():

global x

    x = 111

y = 12

    print(x,y)

     oraz

f()

print(x)

I teraz dostaniemy w drugim przypadku 111.

Należy pamiętać, że modyfikowanie zmiennych globalnych z funkcji lokalnych jest działaniem dość niebezpiecznym i trzeba je wykonywać z uwagą. Łatwo się pogubić i narobić błędów.

Różnice między globalnymi i zmiennymi należy znać i kiedy są dostępne, oraz czy potrafimy zrobić to co powyżej.

Pytanie 1:

**Co zostanie wydrukowane w wyniku wykonania poniższego kodu?**

1. **g = 'jestem globalna'**
3. **def f():**
4. **global g**
5. **g = 'teraz jestem lokalna'**
6. **print(g)**
8. **f()**
9. **print(g)**

Odpowiedź:

Teraz jestem lokalna x2

**26. Operacje na plikach**

Stwórz plik o nazwie „moj\_plik.txt”, a następnie wpisz do niego liczby od 1 do 100, każdą w nowej linijce. A następnie otwórz ten plik i zapisz jego zawartość do listy z\_pliku.

with open ('moj\_lik.txt', 'w') as f:

    pass

I mamy plik. Czym jest blok with? W starszych Pytonach, po każdej operacji na pliku należało ten plik zamknąć pisząc na końcu *f.close()* . Było to raczej problematyczne, bo jak nie zamknęliśmy pliku to plik był otwarty w pamięci. Można tak było skutecznie zapchać pamięć. Po to jest więc blok *with,* który automatycznie zamyka nam plik po jego użyciu.

Jak tworzymy plik? Używamy *open* która również służy do tworzenia plików, następnie w pojedynczym cudzysłowie podajemy nazwę pliku, oraz parametr ‘w’ co oznacza, że będziemy chcieli coś w tym pliku napisać. Następnie używamy słowa kluczowego **AS** oraz nazwę tego pliku, jakim będziemy go se wywoływać. W Pythonie jest przyjęte **F.**

Jeżeli wewnątrz bloku with, wypiszemy słowo kluczowe **pass** , co oznacza, że nie chcemy tu wykonywać żadnych operacji i nie chcemy, aby z tego powodu nam tu wypadł błąd, to te dwie linijki spowodują nam utworzenie nowego pustego pliku.

2/3

with open ('moj\_plik.txt', 'w') as f:

    for n in range(1,101):

        f.write(str(n)) # jeżeli tego nie zrobimy i zostawimy zwykłe write(n) to wyleci nam błądv

# jeżeli zostawimy to tak, to dostaniemy ciągły string od 1 do 100 (12345...)

    # więc?

with open ('moj\_plik.txt', 'w') as f:

    for n in range(1,101):

        f.write(str(n) + '\n')

# i tera będzie dobrze

A teraz trzeba jeszcze otworzyć plik i zapisać jego zawartość do nowej listy

with open('moj\_plik,txt', 'r') as f: # zamiast W mamy R, czyli że będziemy czytać

    z\_pliku = f.readlines() # metoda readlines służy do odczytania tego co mamy w pliku, linijka po linijce i wrzuca je do listy

    print(z\_pliku) # mamy owszem, ale string

Można by jeszcze usunąć /n i zamienić na liczby, ale nie jest to wymagane

**Zadanie**:

**Otwórz**, **przeczytaj**i **wydrukuj**zawartość pliku **"przeczytaj\_mnie.txt"** (plik dołączony jest do zadania i będzie dostępny z poziomu kodu Pythona).

Aby rozkodować poprawnie polskie znaki w funkcji open(), po parametrze 'r' należy dodać jeszcze jeden parametr: 'encoding=utf-8'.

Do przeczytania zawartości pliku użyj metody .read() (odczytuje ona cały plik na raz).

**Rozwiązanie**:

Aby rozwiązać to zadanie należy użyć bloku with i funkcji open(), której:

- jako **pierwszy**parametr podajemy nazwę pliku, czyli "**przeczytaj\_mnie.txt**"

- jako **drugi**parametr podajemy 'r' - informuje on Pythona, że zamierzamy czytać plik.

- **na końcu** dodajemy parametr informujący o kodowaniu pliku  - encoding='utf-8' - pozwoli nam to odczytać polskie znaki.

Następnie czytamy tekst umieszczony w pliku przy użyciu metody .read() i drukujemy go.

1. with open('przeczytaj\_mnie.txt', 'r', encoding='utf-8') as f:
2. print(f.read())

**27. Zagnieżdżone pętle.**

|  |
| --- |
|  |
|  | Objętość graniastosłupa oblicza się na podstawie wzoru: V = a \* b \* h. |
|  | a i b to długości boków jego podstawy, a h to wysokość. |
|  | Poniższy kod znajduje największy graniastosłup jaki możemy utworzyć |
|  | z elementów list A, B i H. |
|  | Ile operacji zostane wykonane w wyniku uruchomienia tego kodu? |
|  | W jaki sposób można by to zadanie rozwiązać bardziej efektywnie? |

    import random                                     # biblioteka random służy do generowania liczb losowych

A = [random.randint(0,100) for i in range(5)]     # tworzenie pięcioelementowej listy losowych integerów z zakresu od 0 do 100

B = [random.randint(0,100) for i in range(5)]

H = [random.randint(0,100) for i in range(5)]

# każdy element z listy A testujemy wraz z każdym elementem z B i H. Taki zapis powoduje, że dla kolejnych liczb z listy A będziemy brali kolejne liczby z B i H. Dzięki temu policzymy każdą możliwą objętość jaką da się stworzyć i porównamy  ją z maksymalną. Jeżeli będzie większa to przypiszemy ją do tego co tam jest poniżej. W ten sposób wyszukamy największą

max\_v = 0

for a in A:

    for b in B:

        for h in H:

            if a \* b \* h > max\_v: # jeżeli to jest prawda, to przypiszemy ten wynik

                max\_v = a \* b \* h # a potem go wydrkujemy

print(max\_v)

Zastanówmy się ile wykonaliśmy operacji. Każdy element z A sprawdzimy z B, a więc 5x5, a następnie każdy z operacji H. W sumie 5x5x5, a więc 125. Możemy też sprawdzić to w inny sposób, delikatnie modyfikując kod.

import random                                     # biblioteka random służy do generowania liczb losowych

A = [random.randint(0,100) for i in range(5)]     # tworzenie pięcioelementowej listy losowych integerów z zakresu od 0 do 100

B = [random.randint(0,100) for i in range(5)]

H = [random.randint(0,100) for i in range(5)]

# każdy element z listy A testujemy wraz z każdym elementem z B i H. Taki zapis powoduje, że dla kolejnych liczb z listy A będziemy brali kolejne liczby z B i H. Dzięki temu policzymy każdą możliwą objętość jaką da się stworzyć i porównamy ją z maksymalną. Jeżeli będzie większa to przypiszemy ją do tego co tam jest poniżej. W ten sposób wyszukamy największą

max\_v = 0

licznik = 0

for a in A:

    for b in B:

        for h in H:

            licznik += 1

            if a \* b \* h > max\_v: # jeżeli to jest prawda, to przypiszemy ten wynik

                max\_v = a \* b \* h # a potem go wydrkujemy

print(max\_v)

print(licznik)

Za każdym razem będzie 125.

Ale ?

Czy da się to zrobić bardziej efektywnie?

No pewno!

print(max(A)\* max(B) \* max(H))

Mnożymy każdy największy wynik z pętelek i otrzymamy na pewno taki sam wynik. Jest ono znacznie znacznie lepsze. Zamiast wykonać 125 wykonamy 15 operacji i całość zrobi się znacznie szybciej.

Na rozmowie musimy wiedzieć jak działają zagnieżdżone pętle i czy wiemy jak go uprościć i bardziej efektywnie

**Pytanie**:

Jaką liczbę wydrukuje poniższy kod?

1. licznik = 0
2. for number in [1,2,3,4,5]:
3. for letter in 'anakonda':
4. licznik += 1
5. print(licznik)

**Odpowiedź**:

W kodzie widzimy dwie pętle: wewnątrz pętli iterującej po liście [1,2,3,4,5] znajduje się druga pętla iterująca po literach stringa 'anakonda'.

Pierwsza pętla wykona się **5** razy (tyle, ile jest elementów w liście).

Dla każdego z tych pięciu obiegów pętli druga pętla wykona się **8** razy (tyle, ile liter jest w stringu 'anakonda').

W rezultacie zmienna licznik zostanie zwiększona 5 \* 8 = **40**razy.

**28. Moduły.**

Jaki błąd popełniono w poniższym kodzie? Co zrobić, aby go uniknąć? Stwórz moduł xero zawierający funkcję, która podany string wydrukuje dwa razy, użyj ten funkcji w kodzie poniżej:

from drukarka import wydrukuj

def wydrukuj(imie):

    print(imie)

wydrukuj("Zbynio")

W tym zadaniu chodzi o to, że moduł nazywał się tak samo jak funkcja, więc aby wszystko działało należy zmienić nazwę funkcji na inną. Wtedy będzie lala

Drugą opcją jest użycie aliasu zapisywanego jako as:

from drukarka import wydrukuj as wydrukuj\_mnie\_z\_druksy

Czym jest moduł?

To każdy plik z rozszerzeniem .py, czyli każdy plik Pythona. Każdy taki plik można zaimportować do innego pliku i używać sobie go dowoli.

# tworzymy plik xsero.py, a w nim funkcję.

def xero(napis):

    print(napis \* 2)

# tera import do naszego plyku.

from drukarka import wydrukuj as wydrukuj\_mnie\_z\_drukarki

from xero import zrob\_ksero

def wydrukuj(imie):

    print(imie)

wydrukuj("Zbynio")

xero("No elo bejbson")

**Pytanie:**

W katalogu roboczym znajdują się dwa pliki:

**matematyka.py** zawiera funkcję dodawanie:

1. def dodawanie(a,b):
2. print("Dodawanie nie jest do niczego w życiu potrzebne!")

**testme.py**, zawiera następujący kod:

1. from matematyka import dodawanie
3. def dodawanie\_lokalne(a,b):
4. print(a + b)
6. dodawanie(2,3)

Co zostanie wypisane w wyniku uruchomienia pliku **testme.py**?

**Rozwiązanie:**

Funkcja, która zostaje uruchomiona w pliku **testme.py** to funkcja dodawanie, która została zaimportowana z modułu matematyka. Funkcja dodawanie\_lokalne została tylko zdefiniowana, ale nie uruchomiona.

Zatem uruchomienie pliku spowoduje wypisanie efektu działania funkcji dodawanie czyli:

**"Dodawanie nie jest do niczego w życiu potrzebne!".**

Zwróć uwagę, że choć funkcja dodawanie pobiera dwa parametry, to nic z nimi nie robi.

Niezależnie więc jakie argumenty podamy przy jej uruchomieniu, zawsze wypisze tego samego stringa.

**29. Pakiety.**

Do czego w Pythonie służy plik \_\_init\_\_.py?

Stwórz kod w krótki sposób pokazujący jego zastosowanie.

Celem tego pliku jest oznaczenie w którym przechowujemy pythonowe moduły. Pakiet to folder zawierający wiele modułów, oraz zawierający pusty lub nie pusty plik \_\_init\_\_.py. Jeżeli takiego pliku nie będzie, to Python nie uzna tego pliku za pakiet.

Aby sprawdzić jak to działa tworzymy folder, a w nim trzy pliki. \_\_init\_.py, trabka.py, bebny.py.

Teraz możemy już zaimportować nasz pakiet. NA 3 sposoby:

import muzyka.trabka

import muzyka.bebny

# uruchamiamy

muzyka.trabka.graj()

muzyka.bebny.graj()

from muzyka import trabka

from muzyka import bebny

trabka.graj()

bebny.graj()

from muzyka.trabka import graj # te działają bezpośrednio bez uruchamiana

from muzyka.bebny import graj # ale jest problem, bo jeżeli uruchomimy tą, to nadpisze ona tą powyżej

# jest to do rozwiązania za pomocą aliasów as graj\_trabka. Ale nie jest to najlepsze rozwiązanie

**Pytanie**:

Wyobraźmy sobie, że mamy pakiet superbohaterzy, w nim moduł batman, a w nim funkcję zaloz\_kostium\_nietoperza. Chcemy użyć tej funkcji w naszym kodzie. Który z poniższych sposobów importu i uruchomienia jest **NIEPOPRAWNY**?

**Rozwiązanie**:

Nie możemy importować do kodu całego pakietu, bez wyszczególnienia jakie moduły nas interesują.

(A jeśli bardzo chcemy to zrobić musimy napisać from pakiet import \*).

Dlatego też kod, w którym importowany jest **tylko**pakiet:

1. import superbohaterzy
2. superbohaterzy.batman.zaloz\_kostium\_nietoperza()

jest **NIEPOPRAWNY**.

**30. IS, Not, In.**

Jaką wartość przyjmie poniższe zdanie logiczne?

Wyjaśnij proces jego ewaluacji i znaczenie poszczególnych słów: Is, not, in.

print(1 is not True in [1,2,3])

 # (1 is not True) and (True in [1,2,3])

Tutaj podobną zasadę jak z zadania 22. Python będzie to robił na dwa dwuelementowe zdania logiczne połączone AND, a potem porówna obydwie strony.

Mimo tego, że dla Pythona 1 i True mają taką samą wartość, to w tym przypadku nie będzie to True, ponieważ liczba nie będzie równa wartości logicznej.

Czyli w tym przypadku gitarka. 1 nie jest równe True, czyli **TRUE.**

Podobnie będzie to wyglądało w drugim przypadku Python będzie to tak naprawdę widział tak:

(1 in [1,2,3])

A więc to dla Pytka prawda, czyli też TRUE, a więc całe działanie będzie **TRUE.**

Dostaniemy Synthax Warning, ale w tym przypadku pytek się myli.

print(1 is not True) 🡺 True (Jedynka nie jest TRUE)

print( 1 != True) 🡺 False (Jedynka jest różna od TRUE)

Takie coś da nam dwa różne wyniki.

IS – jest, NOT nie, IN w jakimś zbiorze, np. w liście lub tupli

**Pytanie**:

Jaka jest wartość logiczna poniższego zdania?

not 2 is 2.0 in [0,2]

**Rozwiązanie**:

Powyższe zdanie zostanie przez Pythona przekształcone w: (not 2 is 2.0) and (2.0 in [0,2]).

(2 is 2.0) to False, bo integer **nie** **jest**floatem. Słowo not powoduje zamianę tego wyrażenia na True.

(2.0 in [0,2]) to True, bo **wartość** floata 2.0 równa jest **wartości**integera 2, a zatem 2.0 znajduje się w liście [0,2].

Mamy więc: True and True czyli True.

**31. MAP I FILTER.**

Otrzymujesz listę nazwisk jakie klienci wprowadzili w formularz na stronie internetowej. Użyj funkcji **filter(),**  aby usunąć z niego wszystkie wpisy, które nie są stringami. Użyj funkcji **map()** aby przerobić nazwiska tak, aby wszystkie zostały zapisane poprawnie, z wielkimi literami tylko na początku imienia i nazwiska.

Filter(funkcja,sekwencja) 🡪 służy do filtrowania danych sekwencji. Jako pierwszy parametr pobiera inną funkcję, a jako drugi sekwencją. Każdy element sekwencji jest przekazywany do funkcji, a ona ma za zadanie zwrócić TRUE lub FALSE. Jeżeli jest on TRUE, to zostanie zwrócony do nowej funkcji.

A więc służy ona do filtrowania jakichś sekwencji przy użyciu funkcji

nazwiska = ['jan kot', 18, 'ANNA KRÓL', 'jÓzef BYK', ['nie', 'wasza','sprawa'], 'ROBERT wąŻ']

nazwiska\_1 = list(filter(lambda x:type(x) is str, nazwiska))

Funkcja filter() jako pierwszy parametr pobiera funkcję lambda. Nasza lambda dla danego X sprawdza czy jest stringiem dla kolejnych sekwencji z nazwiska. Jeżeli jest stringem zostanie dopisany do nowej listy nazwiska\_1. Ważne jest aby dopisać list, ponieważ filter zwraca puste obiekty, więc dodając listę organizujemy je w nową listę

print(nazwiska\_1) 🡺 ['jan kot', 'ANNA KRÓL', 'jÓzef BYK', 'ROBERT wąŻ']

Drugą częścią zadania jest użycie funkcji **map**, aby poprawić ich formatowanie. Działa on podobnie do **filter**, natomiast w przeciwieństwie do poprzedniej, nie tworzy nowej funkcji, tylko pozwala funkcji na przerobienie parametru.

A więc:

nazwiska\_2 = list(map(lambda x: x.lower().title(), nazwiska\_1))

print(nazwiska\_2) 🡺 ['Jan Kot', 'Anna Król', 'Józef Byk', 'Robert Wąż']

Metoda title działa w taki sposób, że każde elementy stringa oddzielone spacją rozpoczyna z dużej litery. Także ten.

**Pytanie**:

Co otrzymamy w wyniku wykonania poniższego kodu?

1. A = [1,2,3,4,5,6,7,8,9]
3. B = list(filter(lambda x: x % 2 == 0, A))
4. C = list(map(lambda x: x \* 10, B))
6. print(C)

**Rozwiązanie**:

* lista B stworzona zostanie w efekcie **przefiltrowania**listy A przy użyciu funkcji, która zwraca True tylko dla **wartości parzystych**
* lista C stworzona zostanie z elementów listy B **zmapowanych**(przetworzonych) przez funkcję, która **mnoży je razy 10**

W **rezultacie**:

lista B = [2, 4, 6, 8]

lista C = **[20, 40, 60, 80] -> poprawna odpowiedź w quizie**

**32. DEKORATORY**

Do czego w Pythonie służą dekoratory. Napisz takowy, który będzie dodawał 3 gwiazdki przed i po efekcie działania udekorowanej funkcyji.

Dekoratory są to funkcje, które jako argumenty pobierają inne funkcje, i ich celem jest zmiana działania tej funkcji pobranej, lub dodaniem czegoś do niej. Można więc „udekorować” funkcję i stąd też nazwa.

def dekorator(funkcja):

    def funk\_udekorowana():

        print("\*\*\*")

        funkcja()

        print("\*\*\*")

    return funk\_udekorowana

@dekorator # nie może tu być żadnego odstępu!

def f():

    print("Czołem")

f()

\*\*\*

Czołem

\*\*\*

**Pytanie**:

1. Co zostanie wypisane w wyniku uruchomienia poniższego kodu?

2. Co zostanie wypisane jeśli usuniemy linijkę 7, w której znajduje się @wypisz\_efekt\_dzialania

1. def wypisz\_efekt\_dzialania(funkcja):
2. def funkcja\_udekorowana():
3. a = funkcja()
4. print(a)
5. return funkcja\_udekorowana
7. @wypisz\_efekt\_dzialania
8. def zwroc\_czesc():
9. return "cześć"

**Rozwiązanie**:

1. Uruchomienie powyższego kodu spowoduje wypisanie stringa **"cześć"**. Funkcja zwroc\_czesc została udekorowana dekoratorem, który powoduje wypisanie (przy użyciu print) zwracanej przez funkcję wartości.

2. Po usunięciu nazwy dekoratora nad definicją funkcji zwroc\_czesc nie wypisze się nic. Funkcja zwroc\_czesc **zwraca**(przy użyciu return), ale nie drukuje stringa, i bez "pomocy" dekoratora ten string nie zostanie wypisany.

Zatem:

1. "cześć"

2. nic

**33. GENERATORY**

Do czego w Pythonie służy słowo kluczowe **yield**? Napisz przykładowy kod je wykorzystujący.

Yield w Pythonie stosowane jest razem z generatorami i służy do zwracania ich wartości

A czym są generatory? Są rodzajem funkcji, będący tak zwanym iteratorem. Iterator jest to obiekt, po którym możemy iterować, czyli odbierać po jednej wartości za każdym razem. Generator możemy więc porównać do funkcji, która wypisuje listę. Zwykła funkcja od razu wyrzucimy nam całość, natomiast generator za każdym uruchomieniem będzie zwracał po jednym elemencie tej listy, a kiedy wszystkie elementy wylecą dostaniemy błąd 🡺 stop iteration error, co oznaczać będzie, że wypisał wszystko co miał do wypisania.

Zatem **yield** będzie dla Pythona sygnałem, że ta funkcja jest generatorem i ma zwracać dane w taki sposób jak generator

def kolejne\_parzyste():

    for n in range (2,21,2):

        yield n

z = kolejne\_parzyste()

print(z) # w ten sposób tego nigdy nie dostaniemy należy zmodyfikować nasz kod.

==> <generator object kolejne\_parzyste at 0x000001A0D7EE4900>

print(next(z))

==> 2, 4 itd...

Aby wypisać wszystkie, należy użyć pętli for.

for i in range(11):

    print(next(z))

# generator expressions podobny do LIST COMPRAHENSIONS

y = ('a' \* n for n in range(5))

Mamy generator, który będzie zwracał kolejne wielokrotności liczby A, począwszy od zera, do czterech. Aby wypisać, znów należy użyć pętli for.

for i in range(5):

    print(next(y)) 🡺 a, aa, aaa, aaaa

Po co tworzy się generatory? Pierwszy powód to oszczędzanie pamięci. Generator jest znacznie bardziej ekonomiczny. Drugi powód to sekwencje bez końca. Listy muszą mieć koniec, a generator może być bez końca.

Napisz generator x, który zwróci 10 kolejnych wielokrotności liczby 5 - zaczynając od 5, kończąc na 50 (włącznie).

Następnie napisz pętlę for, która wypisze kolejne wartości zwracane przez generator (i nie spowoduje wypisania StopIteration Error).

**Rozwiązanie**:

Generator o jaki jesteśmy proszeni w zadaniu nie jest skomplikowany, więc możemy stworzyć go przy użyciu **generator expression**, odpowiednika **list comprehension** dla generatorów.

Poniższy zapis oznacza, że do zmiennej x przypiszemy obiekt generatora, który będzie zwracał liczbę 5 pomnożoną przez liczby z zakresu od 1 do 10.

Następnie przy użyciu pętli for, która wykona się **dokładnie 10 razy** (range(10)) i funkcji next() wypiszemy kolejne liczby przechowywane w generatorze.

1. x = (n \* 5 for n in range(1,11))
3. for \_ in range(10):
4. print(next(x))

**34. ASSERT.**

Napisz fragment kodu, w którym zobrazujesz użycie słowa kluczowego **ASSERT.**  Wyjaśnij jaka jest rola testów jednostkowych i czym charakteryzuje się dobry test jednostkowy.

Asercja w logice oznacza uznanie pewnego zdania za prawdziwe. Taka jest również rola słowa kluczowego **ASSERT**  w Pythonie

assert 1 == 1

Tak więc zakładamy, że to powyżej jest prawdziwe. Jeżeli uruchomimy ten kod, to nic się nie wydarzy. Ale, jeżeli napiszemy np. Assert 1== 2 to dostaniemy we zwrocie ASSERTION ERROR 🡺 Python poinformuje nas, że nasze założenie jest błędne.

Taka więc jest rola **ASSERT** w Pythonie. Sprawdza, czy założenie, które wrzuciliśmy jest prawdą? Do czego więc nam się to może przydać? Główną rolą **ASSERTów** jest pisanie testów.

Np. jeżeli oczekujemy pewnego działania od funkcji, to aby się upewnić czy ona działa możemy napisać inną funkcję, która sprawdzi co jest grane.

def dodawanie(a,b):

    return a + b

def test\_dodawanie\_int():

    assert dodawanie(2,3) == 5

def test\_dodawanie\_str():

    assert dodawanie ('a', 'b') == 'ab'

# mamy funkcje, więc wywołajmy, przetestujmy.

test\_dodawanie\_int()

test\_dodawanie\_str()

Nic się nie wypisało, to znaczy, że funkcje są prawidłowe.

A teraz pozmieniamy naszą funkcję, aby zobaczyć co się stanie jak wyskoczą błędy.

def dodawanie(a,b):

    return a + b + 1

Błąd wyskoczy nam już na etapie pierwszej funkcji, wypadnie nam assertion error. Dostaliśmy informację, ze nie zwróciło nam pięciu, jak powinno. Kod się zatrzymał, więc druga assercja nawet się nie wykonała. Gdyby się wykonała, to byśmy dostali informację, że też nie działa.

Opisana powyżej metoda, choć poprawna, nie jest już stosowana. Jest inna, lepsza, bardziej automatyczna. Aby jej użyć konieczne jest zainstalowanie modułu Pythona pytest.

A jak go użyć? W linii komend wrzucamy pytest, a następnie nazwę pliku, który akurat chcemy przetestować.

Pytest uruchomił nam framework testowy. Dostaniemy podsumowanie, że dwa testy się nie powiodły, zarówno inty jak i stringi. Dostaliśmy też szczegółowe informacje co, gdzie i jak niepasuje.

Ważne!

Aby Pytest znalazł wszystkie funkcje testowe w kodzie, muszą być napisane według określonego formatu. Na początku muszą mieć słowo test na początku. Bez tego nie zostaną zauważone.

Pisanie testów jest bardzo ważnym elementem programowania, również w Pythonie. Dobry test jednostkowy sprawdza wszystkie możliwe przypadki zastosowania danej funkcji. My przetestowaliśmy ją tylko pod kątem dodawania intów i stringów. Gdybyśmy chcieli napisać porządny test jednostkowy, w naszych testach musielibyśmy uwzględnić, listy, tuple i inne obiekty Pythonka.

Czym się charakteryzują dobre testy?

Charakteryzują się tym, że testów mamy znacznie więcej niż funkcji. Na każdą funkcję przypada kilka testów. Pełne pokrycie kodu testami. Kolejnym ważnym punktem, jest testowanie jednej rezczy na raz. Jedna funkcja do intów, jedna do stringów, jedna do tupli. Jak coś się wykrzaczy to nie będziemy wiedzieli dokładnie co i jak się popsiuło.

Musimy znać pytest, jak napisać testy i jakie są cechy dobrych testów.

**35. \_\_INIT\_\_**

Do czego służy w pythonie \_\_init\_ ? Czym różni się od \_\_init\_\_.py .

Napisz fragment kodu wykorzystujący \_\_init\_\_.

Czym \_\_init\_\_ różni się od \_\_init\_\_.py ?

Mówiliśmy już wcześniej o **init.py.** Ten służy do korzystania z pakietów, a konkretniej do oznaczenia folderu, aby powiedzieć Pythonowi, że dany folder będzie pakietem.

PO ZA PODOBIEŃSTWEM WYGLĄDU, NIE MA ON NIC WSPÓLNEGO Z \_\_init\_\_ !

\_\_init\_\_ to konstruktor klasy w Pythonie. Konstruktor klasy jest to metoda klasy, która pozwala tworzyć obiekty danej klasy.

Jeżeli np. będziemy mieli klasę Pies, to konstruktor będzie nam pozwalał tworzyć kolejne obiekty psów.

Nasza klasa pies, a w niej konstruktor. Na tej podstawie będzie nam łatwiej zorientować się how it works.

class Pies:

    def \_\_init\_\_(self, imie, rasa):

        self.imie = imie

        self.rasa = rasa

maly\_pies = Pies("Pikuś", "Bobi")

duzy\_pies = Pies("Zbynio", "Edward")

Stworzyliśmy klasę Pies. Mamy w niej jedną metodę, będąca konstruktorem tej klasy. INIT pobiera 3 parametry, self i dwa pozostałe. SELF to typowe słówko dla OOP (objected oriented programming) i mówi nam, że ta metoda będzie działała na obiekcie danej klasy. Kolejne parametry to już skolko ugodno.

Powyżej stworzyliśmy dwa obiekty klasy pies. Stworzenie obiektu klasy polega na stworzeniu zmiennej. Następnie informujemy, że będą to obiekty klasy pies, a w nawiasach wrzucamy parametry, które trafią do konstruktora. Zatem dla przykładu:

**Do zmiennej maly\_pies przypisz obiekt klasy pies, który w konstruktorze posiada Pikusia jako imię, oraz Bobi jako rasę.** Parametr self po prostu pomijamy.

# Stwórz klasę Kot, a w niej konstruktor, który będzie pobierał imie i kolor. Do zmiennej k przypisz obiekt klasy Kot, daj mu na imię "Mruczek" i dowolny kolor. Dalsza część kodu zajmie się wypisaniem imienia obiektu przechowywanego pod zmienną k.

class Kot:

    def \_\_init\_\_(self,imie,kolor):

        self.imie = imie

        self.kolor = kolor

k = Kot("Mruczek","Szylkretowy")

print(k.imie)

**36. \_\_STR\_\_**

Wykorzystamy kod z poprzedniego ćwiczenia. Co się stanie gdy wypiszemy **print(maly\_pies**) ? Co zrobiłbyś, aby wydrukowana w ten sposób informacja zawierała imię i rasę psa?

class Pies:

    def \_\_init\_\_(self, imie, rasa):

        self.imie = imie

        self.rasa = rasa

maly\_pies = Pies("Pikuś", "Bobi")

print(maly\_pies)

<\_\_main\_\_.Pies object at 0x00000202456EC640>

Jak tak uczynimy to dostaniemy informację, że to obiekt klasy pies i znajduje się w jakimś miejscu pamięci naszego komputera. A jak zrobić, żeby było to tak jak oczekujemy? Musimy użyć metody \_\_STR\_\_

class Pies:

    def \_\_init\_\_(self, imie, rasa):

        self.imie = imie

        self.rasa = rasa

    def \_\_str\_\_(self):

        return f"Pies rasy {self.rasa} wabi się {self.imie}"

maly\_pies = Pies("Pikuś", "Bobi")

Do naszego kodu dodaliśmy metodę STR i łańcuch F. Metoda \_\_STR\_\_ służy do tworzenia reprezentacji tekstowej obiektów klasy. Metoda ta pobiera jako parametr słowo kluczowe self,a więc będzie działała na obiekcie klasy i zwraca nam stringa, gdzie możemy umieścić dowolne informacje.

print(maly\_pies)

Pies rasy Bobi wabi się Pikuś.

I teraz otrzymaliśmy to o co nam chodziło.

**Zadanie**:

Uzupełnij klasę Kot o metodę \_\_str\_\_, która po wydrukowaniu obiektu k wypisze "**Kot Mruczek jest rudy**".

1. class Kot:
3. def \_\_init\_\_(self, imie, kolor):
4. self.imie = imie
5. self.kolor = kolor

8. k = Kot("Mruczek", "rudy")
10. print(k)

**Rozwiązanie**:

W klasie Kot, poniżej metody \_\_init\_\_ tworzymy metodę \_\_str\_\_, w której określamy w jakim formacie ma być wypisana informacja o obiekcie klasy:

1. class Kot:
3. def \_\_init\_\_(self, imie, kolor):
4. self.imie = imie
5. self.kolor = kolor
7. def \_\_str\_\_(self):
8. return f"Kot {self.imie} jest {self.kolor}"

11. k = Kot("Mruczek", "rudy")

14. print(k)

**37. Co zostanie wypisane w wyniku uruchomienia poniższego kodu?**

class ZwierzeLadowe:  # klasa ma dwie metody: przedstaw\_sie i biegaj

    def przedstaw\_sie(self):

        print("Jestem zwierzęciem lądowym.")

    def biegaj(self):

        print("Biegam tu i tam.")

class ZwierzeMorskie:          # klasa ma dwie metody: przedstaw\_sie i plywaj

    def przedstaw\_sie(self):

        print("Jestem zwierzęciem morskim.")

    def plywaj(self):

        print("Pływam tu i tam.")

class SwinkaMorska(ZwierzeLadowe, ZwierzeMorskie): # klasa SwinkaMorska dziecziczy z klas: ZwierzeLadowe i ZwierzeMorskie

    pass  # klasa ZwierzeLadowe lądowe ma priorytet nad klasą ZwierzeMorskie,  bo jest podana jako pierwszas

Dziedziczenie na naszym przypadku oznacza, że świnka morska będzie posiadała cechy klas swoich przodków. Zwróćmy uwagę, że wewnątrz klasy świnka morska nie znajduje się żaden kod, a jedynie słowo kluczowe PASS. To mówi nam o tym, że nie chcemy żadnego kodu tutaj, ani żeby interpretator Pythona zwracał uwagę, że coś tu jest nie tak. Nasza świnka dziedziczy z klas **ZwierzętaLądowe** jak i **ZwierzętaMorskie.**

Poniżej tworzymy obiekt klasy SwinkaMorska, nie przekazujemy żadnych parametrów.

Następnie chcemy, aby świnka nam się przedstawiła, a potem pobiegała i popływała

swinka = SwinkaMorska()      # tworzenie obiektu klasy SwinkaMorska

swinka.przedstaw\_sie()       # wywołanie metody przedstaw\_sie odziedziczonej z klasy ZwierzeLadowe

swinka.biegaj()              # wywołanie metody biegaj odziedziczonej z klasy ZwierzeLadowe

swinka.plywaj()              # wywołanie metody pływaj odziedziczonej z klasy ZwierzeMorskie

Dziedziczenie działa w ten sposób, że nawet jeżeli wewnątrz naszej klasy świnka morska, nie mamy żadnych metod, to może ona używać klas swoich rodziców. Więc, dla przykładu, jeżeli każemy jej biegać, to użyje tej metody z klasy zwierzęta lądowe, a jeżeli każemy jej pływać, to użyje tej metody z klasy zwierzęta morskie.

A teraz pytanie, która z nich zostanie użyta gdy poprosimy, aby nam się przedstawiła? Jeżeli uruchomimy ten kod, to dostaniemy informację, że jest zwierzęciem lądowym, czyli metoda z klasy zwierzęta lądowe. Dlaczego? Jeżeli w Pythonie występuje multidziedziczenie, to priorytet zyskuje klasa, którą wpisaliśmy jako pierwszą. Jeżeli zamienilibyśmy kolejność, to było by na odwrót. A więc ta klasa, która jest po najbardziej po lewej stronie jest z punktu dziedziczenia ważniejsza.

**Pytanie**:

Co zostanie wypisane w wyniku uruchomienia poniższego kodu?

1. class Zwierze:
2. def przedstaw\_sie(self):
3. print("Jestem jakimś zwierzeciem.")
4. def daj\_glos(self):
5. print("Nie wiem czym jestem i jaki dźwięk wydaję.")
7. class Gawron(Zwierze):
8. def daj\_glos(self):
9. print("Kraaa kraaa!")
11. g = Gawron()
12. g.przedstaw\_sie()
13. g.daj\_glos()

**Rozwiązanie**:

Klasa Gawron dziedziczy z klasy Zwierze.

Po wywołaniu na g - obiekcie klasy Gawron metody przedstaw\_sie(), zostanie wywołana metoda pochodząca z klasy Zwierze, ponieważ klasa Gawron nie ma tej metody zdefiniowanej. Zatem wypisane zostanie:  **"Jestem jakimś zwierzeciem."**

Wywołanie metody daj\_glos() na obiekcie g spowoduje użycie metody daj\_glos(), która zdefiniowana jest w klasie Gawron, a zatem wypisanie: **"Kraaa kraaa!".**

**38. Do czego służą dekoratory @statichmethod i @classmethod ?**

class Matematyka:

    def \_\_init\_\_(self):

        self.pi = 3.14

    def policz\_obwod(self,r):

        return 2 \* self.pi \* r

    @staticmethod 🡺 Dzięki temu oznaczeniu możemu naszą metodę wywołać bez obiektu klasy

    def dodaj (a,b):

        return a + b

m = Matematyka()

print(m.policz\_obwod(5)) 31.400000000000002

print(Matematyka.dodaj(2,3)) 5

Uwaga:

print(Matematyka.policz\_obwod(5))

Taki zapis spowoduje błąd. Piątka którą tu podaliśmy została wpisana na miejsce tego selfa, przez co system myśli, że brakuje nam jednego argumentu

Utworzymy kolejną metodę:

@staticmethod

def dodaj\_i\_pomnoz(a,b):

        return dodaj(a + b) \* 2

Kiedy spróbujemy ją uruchomić będzie błąd. Dlaczego? To pewna istotna cecha metod statycznych. Nie są one świadome tego, że znajdują się w klasie. Nie widzą innych elementów tej klasy. Dalej jesteśmy w klasie **Matematyka**, a pomimo to metoda **dodaj\_i\_pomnoz** nie potrafi sięgnąć do metody dodaj, bo nie jest jej świadoma.

Metody statyczne działają więc jak zwykłe metody w Pythonie, jakby nie były powiązane żadną klasą. Ich powiązanie to raczej kwestia organizacji kodu. Czyli fakt, że wywołujemy te metody przy użyciu klasy Matematyka, mówi nam o tym, że one tam się organizacyjnie znajdują, ale moglibyśmy umieścić je gdzieś indziej i zadziałają sobie tak samo. Więc nasza metoda nie ma prawa zadziałać, jeżeli będzie próbowała odwołać się do innej metody w tej klasie.

Żeby stworzyć metodę, która nie wymaga obiektu klasy, ale jest w stanie sięgnąć do innej metody w klasie, musimy użyć innego dekoratora, który nazywa się **classmethod.**

class Matematyka:

    def \_\_init\_\_(self):

        self.pi = 3.14

    def policz\_obwod(self,r):

        return 2 \* self.pi \* r

    def dodaj (a,b):

        return a + b

    @classmethod

    def dodaj\_i\_pomnoz(cls, a, b):

return cls.dodaj(a,b) \* 2

 # CLS jest konieczne, aby informować, że jest to metoda classowa. Zapis ten też oznacza, że

# chcemy sięgnąć do metody DODAJ, która znajduje się w tej samej klasie, na której będzie nasza metoda klasowa.

# W skrócie: Ten zapis informuje Pythona, że ta metoda może być wywołana bezpośrendio na klasie, nie potrzebuje obiektu,

#czyli instancji klasy do działania, ale jest świadoma bycia w klasie i potrafi sięgnąć do innych metod w niej się znajdujących

I teraz działa!

Podsumowując. Mamy w Pythonie 3 rodzaje metod.

🡺 Metody działające na obiektach klasy, które posiadają słowo kluczowe SELF.

🡺 Metody statyczne, które nie mają żadnego słowa kluczowego, wyglądają jak każda inna metoda w Pythonie i oznaczamy je dekoratorem **@statticmethod**

**🡺**Metody klasowe, które używają słowa kluczowego CLS i są świadome bycia wewnątrz klasy. Nie potrzebują obiektu klasy, żeby działać, ale wiedzą, że znajdują się w klasie. I te metody oznaczamy dekoratorem **@classmethod**

**Pytanie**:

Wewnątrz klasy możemy tworzyć metody:

- działające na **obiekcie**klasy

- **klasowe**- działające na klasie samej w sobie

- **statyczne**- które nie mają świadomości bycia w klasie.

Dwie z tych metod wymagają użycia określonego parametru na pierwszym miejscu listy parametrów metody, jedna nie.

Zaznacz, w którym punkcie poprawnie powiązano **rodzaj metody**ze stosowanym w niej **parametrem**.

**Rozwiązanie**:

Metody działające na **obiekcie klas** wymagają słowa kluczowego self.

Metody **statyczne**nie potrzebują **żadnego słowa kluczowego.**

Metody **klasowe**wymagają słowa kluczowego cls.

Zatem :

* metoda działająca na obiekcie klasy - self
* metoda statyczna - **brak**
* metoda klasowa - cls

„

Istnieją jednak takie metody, które nie operują na konkretnej instancji klasy. Typowo nazywa się je **statycznymi**. W Pythonie nie posiadają one parametru self, lecz są opatrzone dekoratorem @staticmethod:

Statyczną metodę można wywołać zarówno przy pomocy nazwy klasy (Counter.format\_string()), jak i jej obiektu (Counter().format\_string()), ale w obu przypadkach rezultat będzie ten sam. Technicznie jest to bowiem zwyczajna funkcja umieszczona po prostu w zasięgu klasy zamiast zasięgu globalnego.

Mamy wreszcie trzeci typ, mieszczący się w pewnym sensie pomiędzy dwoma opisanymi powyżej. Nie wydaje mi się jednak, żeby występował on w żadnym innym, popularnym języku. Chodzi o **metody klasowe** (*class methods*). Nazywają się tak, bo są wywoływane na rzecz całej klasy (a nie jakiejś jej instancji) i przyjmują ową klasę jako swój pierwszy parametr. (Argument ten jest często nazywany cls, ale jest to o wiele słabsza konwencja niż ta dotycząca self).  
W celu odróżnienia od innych rodzajów, metody klasowe oznaczone są dekoratorem @classmethod:

Podobnie jak metody statyczne, można je wywoływać na dwa sposoby – przy pomocy klasy lub obiektu – ale w obu przypadkach do cls trafi wyłącznie klasa. Tutaj akurat będzie to Counter, lecz w ogólności może to być także klasa pochodna:

„

**UWAGA. A TERAZ W ROZDZIAŁACH 39-42 wykonamy 4 najpopularniejsze zadania na rozmowach. Fizz Buzza, ciąg Fibonacciego, Rekurencyjne wypisywanie zawartości katalogu oraz wyszukiwanie binarne**

**39. FIZZ BUZZ**

Pytanie 39 - napisz program, który dla kolejnych liczb z zakresu od 1 do n: wypisze: "Fizz" - jeśli liczba będzie podzielna przez 3 wypisze: "Buzz" - jeśli liczba będzie podzielna przez 5 wypisze: "FizzBuzz" - jeśli liczba będzie podzielna przez 3 i 5 jeśli nie zajdzie żaden z tych przypadków, to wypisz po prostu liczbę. Zadanie to znane jest pod nazwą: FizzBuzz.

FizzBuzz to klasyk. Łatwiutki

Ver1.

def fizzbuzz(n):

    for num in range (1, n+1):

        if num % 3 == 0:

            print('Fizz')

        if num % 5 == 0:

            print('Buzz')

        elif num % 3 == 0 and num % 5 == 0:

            print('FizzBuzz')

        else:

            print (num)

fizzbuzz(30)

Ver2.

def fizzbuzz(n):

    for num in range(1, n+1):

        wynik = ''              # w każdym obiegu pętli tworzymy pusty string do przechowywania wyniku

        if num % 3 == 0:

            wynik += "Fizz"     # wynik = wynik + "Fizz"

        if num % 5 == 0:

            wynik += "Buzz"

        if num % 3 != 0 and num % 5 != 0:

            wynik = str(num)    # jeśli liczba nie jest podzielna ani przez 3 ani przez 5

        print(wynik)            # to do zmiennej wynik dodajemy stringa utworzonego z tej liczby

fizzbuzz(30)

**40. Ciąg Fibonacciego.**

Ciąg Fibonacciego to ciąg liczb, którego:

- zerowy element wynosi 0, pierwszy element wynosi 1,a każdy kolejny element jest sumą dwóch poprzedzających go elementów.

Napisz funkcję, która zwróci n-ty element ciągu Fibonacciego.

# kolejny element ciągu: 0 1 2 3 4 5 6 7 8 9 10

# wartość dla elementu: 0 1 1 2 3 5 8 13 21 34 55

**Zadanie to można rozwiązać na wiele sposobów. W tym zadaniu zrobimy to na 2.**

1. Rekurencyjne

def fibo\_r(n):

    if n <= 1:

        return n

    return fibo\_r(n - 1) + fibo\_r(n - 2)

print(fibo\_r(10))

Jego wadą jest jego złożoność obliczeniowa. Stworzone w rezultacie drzewo wywołań będzie bardzo duża i wynosi: o od 2 do N, a więc jeżeli chcemy obliczyć ile wynosi wartość 10, to naszą funkcję trzeba wywołać 1024 razy.

Za każdym razem funkcja wywoła poprzednią wartość, a ta będzie obliczała poprzednią i poprzednią i poprzednią.

Nawet mój mighty Lenovo Legion potrzebuje kilkunastu chwil, aby obliczyć nam ile wynosi wartość setna ciągu Fibonnaciego.

1. Liniowe.

# rozwiązanie liniowe 🡪 Będzie wymagało tyle operacji ile elementów ciągu będziemy chcieli obliczyć. Wadą jest to, że jest trudniejsze do zrozumienia.

def fibo\_l(n):

    p,d = 0,1 # P to skród od pierwsza, D od druga

    for \_ in range(n): # podkreślenie oznacza, że nie będziemy używać zmiennej

        p,d = d, p +d # tutaj do pierwszej P przypisujemy D, a do drugiej sumę, ze wcześniejszego obiegu pętli

    return p

print(fibo\_l(100)) 🡪 354224848179261915075

**Pytanie 41 – Rekurencyjne wyszukiwanie zawartości katalogu.**

Do zabawy z tym konieczny jest import OS, który pozwoli nam się za to zabrać. Tworzymy pętlę for dla katalogu, a następnie korzystamy z trzech podanych przez ucitiela.

#os.listdir - zwraca zawartość danego katalogu

#os.path.join - łączy dwa stringi w ścieżkę czytelną dla danego systemu operacyjnego

#os.path.isdir - sprawdza czy pod nadą ściezką znajduje się katalog

import os

def wypisz(sciezka):

    for element in os.listdir(sciezka): # pętla for dla każdego elementu w katalogu

        sciezka\_ele = os.path.join(sciezka, element)

        if os.path.isdir(sciezka\_ele):

            wypisz(sciezka\_ele)

        else:

            print(sciezka\_ele)

wypisz(r"C:\Users\karol\Desktop\WebDevCourse\Kursy z UDEMY")

**Pytanie 42 – WYSZUKIWANIE BINARNE**

Przy użyciu wyszukiwania binarnego sprawdź czy liczba 341 znajduje się w posortowanej liście P

P = [-10, -7, -5, -3, 0, 3, 5, 21, 68, 341, 500]

Wyszukiwanie binarne to jedna z najbardziej popularnych rzeczy w całym programowaniu. Popularne, bo pozwala przeszukać listy w bardzo wydajny sposób. Możemy to przyrównać do książki telefonicznej. Jeżeli chcemy wyszukać Kowalskiego, to otworzymy ją najpewniej w środku i poszukamy litery K, żeby znaleźć interesującą

P = [-10, -7, -5, -3, 0, 3, 5, 21, 68, 341, 500]

szukana = 341

lewy = 0

prawy = len(P) -1

# teraz czas na pętlę

while lewy <= prawy:

    srodkowy = (lewy + prawy) // 2 # dzielenie bez reszty

    if P[srodkowy] == szukana:

        print(f"Liczba {szukana} znajduje się na tej liście.")

        break

    elif P[srodkowy] < szukana:

        lewy = srodkowy + 1

    else:

        prawy = srodkowy - 1

else:

    print(f"Liczba {szukana} nie znajduje się na tej liście.")

Liczba 341 znajduje się na tej liście.

**Pytanie 43 – Big O**

Uszereguj podane złożoności obliczenieowe algorytmów od najlepszej do najgorszej

**O(1)** - sprawdzenie czy element jest w słowniku (tablicy hashującej)

**O(log(n))** - wyszukiwanie binarne. Ilość operacji to logarytm od liczby mówiącej o rozmiarze zbioru.

**O(n)** - jednokrotne przejrzenie listy o długości n. Też nie jest najgorszy. Jeżeli lista będzie miała 23 elementy to wykona się 23 razy.

**O(n \* log(n))** - (wydajne) sortowanie (timsort, quicksort) 🡪 To tak jakby iloczyn tych dwóch poprzednich

**O(n\*\*2)** - mnożenie każdego z każdym z elementów na liście, lub mało wydajne elementy sortowania. Np. BUBBLESORT 🡪 To już nie jest za dobre. Całkiem to jest złożone.

**O(2\*\*n)** - Fibonacci rekurencyjnie 🡪 Tej krowy nie trzeba przedstawiać. Bardzo zła złożoność obliczeniowa. Jeżeli tylko można jej uniknąć to należy to zrobić. Potrafiła ona nawet utentegować majty Lenovo

Złożoność obliczeniową określa się często jako Big O. O od jakiejś wartości. Mówi nam ona ile operacji będzie musiał wykonać komputer, aby dany algorytm zakończył swoją pracę na zbiorze o rozmiarze N

Tutaj przedstawienie za pomocą MATH:

import math       # poniższy kod wyświetla wykres przedstawiający porównanie różnych złożoności obliczeniowych

                  # przy użyciu popularnej bibliteki matplotlib

import matplotlib.pyplot as plt

seria0 = [1] \* 10

seria1 = list(range(1,11))

seria2 = [math.log(x) for x in seria1]

seria3 = [math.log(x) \* x for x in seria1]

seria4 = [x\*\*2 for x in seria1]

seria5 = [2\*\*x for x in seria1]

plt.plot(seria0, label="O(1)")

plt.plot(seria2, label="O(log(n))")

plt.plot(seria1, label="O(n)")

plt.plot(seria3, label="O(n \* log(n))")

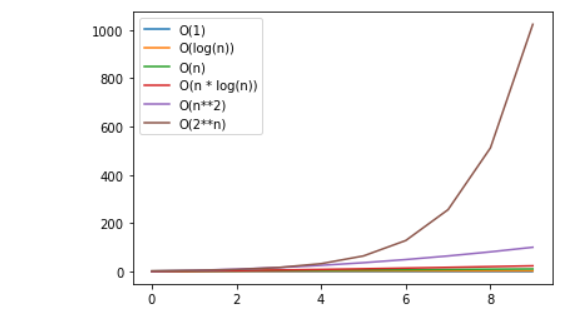
plt.plot(seria4, label="O(n\*\*2)")

plt.plot(seria5, label="O(2\*\*n)")

plt.legend()

plt.show()

I tak to wygląda jako GRAF:



**Pytanie 44. Dynamiczne typowanie.**

Co to znaczy, że Python jest językiem dynamicznie typowanym?

Dybnamicznie typowanie oznacza sytuację, kiedy tworząc zmienną przypisujemy do niej typ dopiero w momencie kiedy zostanie jej przypisana konkretna wartość.

Jeżeli utworzymy nową zmienną i nie przypiszemy jej wartości, to Python nie wie jakiego typu jest to zmienna. Kiedy damy jej nową wartość, np. 5 to Python nam dopiero teraz przysposobi informację, że to liczba całkowita (integer).

a = 5

print(type(a))

Jeżeli jeszcze raz zmienimy typ naszych danych na ciąg, lub jak kto woli stringa i w tym momencie, nasza zmienna A stanie się nowym typem danych.

a = "pięć"

To samo stanie się jak przypiszemy do zmiennej A np. listę.

Istnieje wiele języków programowania, w których taka opcja nie jest możliwa(język statyczny) i wyrzuci nam błąd. W językach statycznych raz przyjęta opcja staje się obowiązującą. To jest zaleta naszego Pythona, że można tak robić.

Pomimo tego, że Python jest dynamicznie typowany to istnieje możliwość, aby zaznaczyć jakich typów należy używać w danych sytuacjach. Rozwiązanie to jest szczególnie użyteczne jeżeli np. konstruujemy w Pythonie duży kod(duża aplikacja na wiele osób).

Więc zapiszemy teraz nasz kod, tak aby stał się on statyczny

def a(word: str) -> List[str]:

    return list(word)

Mamy tutaj prostą funkcję A, pobiera ona stringa i zwraca listę kolejnych liter tego stringa. Jednakże, zwróćmy uwagę co dzieje się po dwukropku. Po dwukropku znajduje się informacja, że zmienna będzie typu STRING. Strzałka mówi nam o tym, że wartość zwracana przez funkcję powinna być typu :” Lista zawierająca stringi” .

Aby użyć tego zapisu musimy najpierw zaimportować moduł typing, a z niego klasę list. Gdybyśmy chcieli użyć TUPLI czy SŁOWNIKA, również należałoby go zaimportować.

    print(a('Python')) --> ['P', 'y', 't', 'h', 'o', 'n']

Jednakże, spróbujmy złamać tą funkcję. Wywołajmy liczbę

def a(word: str) -> List[str]:

    return 5

Funkcja, mimo, że sprzeczna z zasadami zwróciła tak jak jej kazaliśmy liczbę 5.

Jak więc użyć tego zapisu aby kontrolować kod? Musimy użyć biblioteki **mypy**

Żeby jej użyć w linii komend wpisujemy: **mypy** oraz **nazwę pliku**, którego chcemy użyć

Jak klikniemy to MYPY sprawdzi nam poprawność kodu i wskaże co i jak mamy nie tak.

A mechanizm, który pozwala w Pythonie stosować statyczne typowanie nazywa się:

**TYPE HINTS**

**Pytanie**:

Poniższa funkcja ma pobrać dwa argumenty - **integera**i **stringa**, a zwrócić ma **listę booleanów**.

Który z poniższych zapisów jest poprawny?

(Zakładamy, że nad każdym z tych fragmentów kodu znajduje się już linijka: from typing import List)

**Rozwiązanie**:

Stosując **type hints** (wsparcie dla statycznego typowania) w Pythonie:

* typy zmiennych podajemy po **dwukropku**
* typ zwracany przez funkcję podajemy po **strzałce ->**

Zatem poniższy zapis jest poprawny:

1. def jestem\_statyczna(num: int, word: str) -> List[bool]:
2. pass

**Pytanie 45. PEP8. (bardzo często się pojawia)**

W jaki sposób zadbasz o to, aby twój kod był czytelny i łatwy do zrozumienia dla innych programistów.

**PEP8** to dokument opisujący standard opisu kodu w Pythonie. Mówi o tym jak tworzyć nazwy funkcji, nazwy zmiennych, zapisywać klasy oraz nawet jakie odstępy powinniśmy utrzymywać pomiędzy kolejnymi jego komponentami.

Te zalecenia wbrew pozorom są bardzo ważne. W pracy musimy tworzyć kod, który jest czytelny, a PEP8 to po prostu zestaw reguł, który każdy musi znać

Jedna z zasad PEP8 mówi o tym jak powinniśmy zapisywać importy:

Po pierwsze, jeden import w jednej linijce

import os

import modul\_zewnętrzny

Dobrze? Nie! Jeżeli robimy importy różnych typów to powinniśmy oddzielać je jedną linijką pustej przestrzeni.

Więc to będzie OK(ponieważ to są domyślne moduły z Pythona):

import os

import math

A to już nie:

import os

import modul\_zewnętrzny

Kolejna z zasad:

Jeżeli tworzymy klasę, to nad i pod klasą zostawiamy co najmniej dwie linijki wolnej przestrzeni, a dodatkowo klasy zapisujemy tak zwanym CapCasem(czyli każde słowo dużą literą):

class CapCase:

pass

Zmienne?

Zmienne zawsze zapisujemy małymi literami. Nie stosujemy w nich znaków specjalnych ani cyfr. Jeżeli zmienna zawiera więcej niż jedno słowo, używamy snake case.

zmienna\_malymi\_literkami = ‘snake case’

Kolejna zasada mówi o funkcjach. Nad i pod funkcją jedna linijka wolnego miejsca, a nazwy funkcji jak nazwy zmiennych:

def funkcja\_testowa():

    pass

Kolejna zasada pepa dotyczy zapisu sekwencji. Pomiędzy elementami sekwencji jedna spacja!

A=[1, 2, 3, 4] --> TO JE DOBRE

B =['to','nie','jest','właściwy','zapis']

W PyCharmie możemy sobie włączyć przestrzeganie reguł skrótem klawiszowym CTRL + ALT + L

**Pytanie 46. PIP.**

W jaki sposób importujesz zewnętrzne moduły w Pythonie?

Oczywiście, że PIP. W linii komend, albo w CMD: pip install np. colorama

from colorama import init, Fore, Back, Style

init()

print(Fore.RED + "Tekst w czerwieni")

print(Back.BLUE + "Na niebieskim tle")

print(Style.RESRET\_ALL + "... i znów klasycznie")

PIP łączy się z PyPI. Na PyPI jest też cała dokumentacja jednego modułu. Warto znać tą stronę:

[www.pypi.org](http://www.pypi.org)

**Pytanie 47. DIR i HELP.**

Do czego służą funkcje dir() i help().

Obie są zaprojektowane aby użyć ich w pliku lub w edytorze.

Jeżeli wypiszemy print(dir()) TO wyświetli nam wszystkie metody jakich możemy użyć

if \_\_name\_\_ =='\_\_main':

    pass

Często spotykamy się z takim kodem. Jego celem jest sprawdzenie czy plik jest uruchomiony bezpośrednio czy zaimportowany do innego pliku. To ważna rola atrybutu name

Dira możemy uruchomić również w trybie interaktywnym i wrzucić np. taki kod:

dir(5)

Python zwróci nam listę wszystkich możliwych atrybutów intidżera.

**Pytanie 48. Najważniejsze cechy Pythona**

1.Jakie są najważniejsze cechy Pythona?

2.Jakie są wady i zalety?

3.Dlaczego zdecydowałem się na Pythona?

1:

a)

Prosta, logiczna i najbardziej czytelna składnia. 🡪 Nie sprawia problemów. Czytanie kodu jest łatwe.

b)

Brak nawiasów klamrowych! Zamiast nich mamy wcięcia. Jak dla mnie jest to o wiele lepsze.

c)

Język jest to interpretowany, a nie kompilowany. Jest więc czytany przez program linijka po linijce. W przeciwieństwie do kodów kompilowanych, nie musi być kompilowany przez kompilator i uruchamiany dopiero przez cały plik.

Oznacza to, że np. 99 linijek wykona się poprawnie, a dopiero w setnej nam wyrzuci błąd.

d)

Jest dynamicznie typowany. Oznacza to, że nie musimy od razu przypisywać zmiennej jakiegoś typu. Zadeklarujemy ja dopiero w momencie kiedy przypiszemy jej jakąś wartość.

2.To, że jest językiem interpretowanym jest też jego wadą, ponieważ jest wolniejszy niż języki kompilowane.

Możemy to rozwiązać np. wstawiając do naszego kodu elementy kodu C lub C++

Mamy mnóstwo bibliotek! Np. taką jak Antigravity, która importuje nam komiks.

Typowanie dynamiczne może być problemem, jeżeli wcześniej typowano statycznie. Jest to jednak w zasadzie kwestia osobistych preferencji.

**Pytanie 49. Skąd czerpiesz wiedzę?**

Gdzie szukamy informacji?

No oczywiście jego dokumentacja.

Tu jest wszystko i to w dodatku prostym języczkiem

Mamy też modulł 🡪 Python Tutorial

Programiz.com/python 🡪 Świetny tutorial Pythona

Stack Overflow. Tutaj nie trzeba chyba nic mówić

GIT HUB 🡪 Lista najlepszych frameworków, bibliotek i fajnych rzeczy

Hackerrank 🡪 Ogromna baza zadań

Meetup

**Pytanie 50. Zen of Python**

Co się stanie po uruchomieniu tego kodu?

import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

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

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

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

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.

Now is better than never.

Although never is often better than \*right\* now.

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.

Zbiór 19 porad, coś jak przykazania dla programisty.

Piękne jest lepsze niż brzydkie.

Jawne jest lepsze niż niejawne.

Proste jest lepsze niż złożone.

Złożone jest lepsze niż skomplikowane.

Płaskie jest lepsze niż zagnieżdżone.

Rzadkie jest lepsze niż gęste.

Liczy się czytelność.

Specjalne przypadki nie są na tyle wyjątkowe, aby łamać zasady.

Chociaż praktyczność przewyższa czystość.

Błędy nigdy nie powinny przejść bezgłośnie.

Chyba że wyraźnie uciszono.

W obliczu niejasności odrzuć pokusę zgadywania.

Powinien być jeden - a najlepiej tylko jeden - oczywisty sposób na zrobienie tego.

Chociaż na początku może to nie być oczywiste, chyba że jesteś Holendrem.

Teraz jest lepiej niż nigdy.

Chociaż nigdy nie jest często lepsze niż \* teraz \*.

Jeśli implementacja jest trudna do wyjaśnienia, to zły pomysł.

Jeśli implementacja jest łatwa do wyjaśnienia, może to być dobry pomysł.