**SEKCJA 1, 2. KONFIGURACJA ŚRODOWISKA, PODSTAWY**

**Lekcja\_1\_Wprowadzenie**

**Anaconda Navigator → Spyder**. Anaconda to dystrybucja języków programowania Python i R do obliczeń naukowych (nauka o danych, aplikacje do uczenia maszynowego, przetwarzanie danych na dużą skalę, analizy predykcyjne itp.). Anaconda zawiera wiele popularnych, już wbudowanych bibliotek, które w przeciwnym razie musieliśmy samodzielnie doinstalowywać do Pythona. To m.in. NumPy, SciPy, PyQT, Spyder i inne. Spyder (Naukowe środowisko programistyczne języka Python) to jest potężne interaktywne środowisko programistyczne dla języka Python. Ma zaawansowane funkcje edycji, interaktywne testowanie, debugowanie i introspekcję oraz numeryczne środowisko obliczeniowe.

**Środowisko programistyczne** → IDE (od ang. integrated development environment) – program lub zespół programów (środowisko) służących do tworzenia, modyfikowania, testowania i konserwacji oprogramowania. Programy będące zintegrowanymi środowiskami programistycznymi charakteryzują się tym, że udostępniają złożoną, wieloraką funkcjonalność obejmującą edycję kodu źródłowego, kompilowanie kodu źródłowego, tworzenie zasobów programu (tzn. formatek/ekranów/okien dialogowych, menu, raportów, elementów graficznych jak ikony, obrazy), tworzenie baz danych, komponentów i innych.

**Tworzenie osobnych komórek kodu** → # %%. Za pomocą takiej linijki kodu można oddzielić kod od siebie i dzięki temu uruchamiać nie cały kod, ale jego wybrane, oddzielone od siebie fragmenty.

Np.

# %%

print(‘Hello world’)

# %%

print(Hello’)

**Uruchamianie fragmentu kodu** → Shift+Enter

**Uruchamianie całego kodu** → F5

**Layout programu Spyder**: trzy okna. Okno kodu (skryptów), okno informacji dodatkowych (eksplorator zmiennych, plików) oraz okno konsoli wyświetlającej wyniki prac.

W oknie skryptów piszemy nasz kod.

W oknie informacji dodatkowych widać eksplorator zmiennych (widać co się tworzy, jakie obiekty) czy podgląd aktualnych plików w folderze roboczym.

W konsoli ukazują się wyniki pracy z Python.

**Przechowywanie plików roboczych** → utwórz katalog. Najpierw określ jaki jest teraz katalog roboczy wpisz komendę pwd do konsoli wyników prac. Za pomocą ikony folderu zmień adres katalogu roboczego, utwórz sobie jakiś katalog roboczy. Files; new; Python file.

Np.

pwd

**Pwd** → komenda przywołująca katalog roboczy

**Tworzenie plików Pythone-owskich w katalogu roboczym**. Prawym na okno eksploratora plików i utwórz plik z końcówką py. (takie jest zakończenie plików Pythonowskich).

**Lekcja\_2\_Drukowanie wyników do konsoli\_1**

**Nazwa kodowania**: program Spyder automatycznie na początku dodaje nazwę kodowania. Tak ustawia się kodowanie znaków dla danego pliku - domyślnie jest to utf-8.

Np.

# -\*- coding: utf-8 -\*-

**# -\*- coding: utf-8 -\*-** → domyślne kodowanie

**Gwiazdka nad nazwą pliku**: oznacza nie zapisanie → zapis, Crtl+S.

**Uruchom kod** → zielona strzałka (F5).

**Funkcja print** → ona pokaże wynik kodu w konsoli. W przypadku działań arytmetycznych wynikiem kodu będzie wynik działania arytmetycznego.

Np.

print(2 + 2)

**Okno wyników**: ma funkcję testową, eksploracyjną → możesz tam pisać kod, ale on się nie zapisze, gdy go tam piszesz. Dopiero pisanie kodu w oknie skryptu pozwala na zapisanie go na później.

**Zakomentowanie za pomocą #**. Umówmy się, że komentarz piszę nad linijką kodu, którego dotyczy.

Np.

# komentarz

**Dzielenie komórki kodu** → na osobne skrypty.

Np.

# %%

**Uruchamianie działania samej jednej komórki** → Shift+Enter

**Dzielenie w Python**: zawsze zwraca liczby rzeczywiste, zmiennoprzecinkowe.

Np.

6 / 2 → da wynik w konsoli testowej “3.0”

**Operator usuwający z dzielenia wartości po przecinku** → //, po prostu pokaże wynik dzielenia bez wartości po przecinku, np. 14 // 6 = 2.

Np.

7 // 6 → da w konsoli wynik “1”, bez wartości po przecinku

**Potęgowanie, operator** → \*\*

Np.

2 \*\* 5

**Operator dzielenie modulo** → zwraca resztę z dzielenia. Jest to symbol %. Przydaje się do sprawdzenia czy wynik dzielenia to liczba parzysta czy nieparzysta. Pamiętaj, że wynik zawsze będzie o jeden mniejszy niż dzielnik. Na poniższym przykładzie będzie to więc 0, 1, 2.

Np.

10 % 3 → co da wynik 1, bo są trzy trójki w 10 plus 1

**Weryfikacja liczb parzystych i niepatrzystych poprzez modulo** → jak na przykładzie poniżej. Modulo oddaje resztę z dzielenia.

Np.

number = 7

if number % 2 != 0:

    print(f'The number {number} is odd.')

else:

    print(f'The number {number} is even.')

**Kolejność wykonywania działań →**  Python pamięta o kolejności wykonywania działań.

**Deklarowanie zmiennych liczbowych** → za pomocą operatora =. Żeby zmienne się zapisały uruchom kod. Zapiszą się jak zauważysz w eksploratorze zmiennych. Eksplorator pokazuje nazwę zmiennej, jej typ, rozmiar i wartość.

Np.

a = 10 → to zmienna integer

b = 20

**Integer** → zmienna liczbowa

**=** → operator przypisania. Przypisuje wartość do zmiennej.

**Deklarowanie zmiennych cd.**: i wyciąganie zadeklarowanych zmiennych.

Np.

a = 10

b = 20

c = a \* b

print(c)

**Spacja**: dobrą praktyką jest pomiędzy operatorami wstawiać spację. Jest to zabieg, który ma poprawić czytelność kodu.

**Wpisywanie tekstu i symbol ucieczki** → za pomocą podwójnego lub pojedynczego cudzysłowu. Częściej korzystaj z pojedynczego. Czasem jednak może być potrzeba użyć podwójnego, np. w sytuacji gdzie piszesz It’s the best!, gdzie apostrof już jest, w środku zdania. Daj wtedy podwójny cudzysłów albo dodaj symbol ucieczki (backslash) przed apostrofem, dzięki czemu Python zrozumie, że to apostrof a nie koniec pojedynczo napisanego cudzysłowu.

Np.

“I love Python!”

‘I love Python!’

“It’s the best”

‘It\’s the best’

**Przełamanie tekstu (enter) do nowej linii** → za pomocą symbolu \n. Możesz też to zrobić normalnie w tekście, tzn. stawiać w kodzie entery jak chcesz, ale do tego na początku i końcu muszą być trzy cudzysłowy ”””. Przełamywanie tekstu musi odbywać się za pomocą funkcji print. Pisząc funkcje nie musisz używać przełamania tekstu, wystarczy enter

Np.

print(‘Python\n4.1’) → przełamanie poprzez fizyczne wpisanie \n

print(”””Python

4.1”””) → przełamanie poprzez wpisanie na początku i końcu trzech cudzysłowów, dzięki czemu przełamania będą tam, gdzie po prostu postawię entery.

**Invalid syntax** → jest to podstawowy błąd oznaczający niepoprawną składnię kodu.

**Definiowanie zmiennych po przecinku →** możesz definiować zmienne szybko ustawiając je jedna za drugą po przecinku. Po prostu przypisane im obiekty muszą korespondować z ułożeniem zmiennych.

Np.

a, b, c, d = 1, 2, 3, 4

print(a)

print(b)

print(c)

print(d)

**Lekcja\_3\_Drukowanie wyników do konsoli\_2**

**Czyszczenie zdefiniowanych zmiennych z eksploratora zmiennych** → restartuj kernel za pomocą skrótu Crtl+. (Crtl + kropka). Przy każdym uruchomieniu Spydera, wprowadza on “jądro” naszego kodu, czyli zmienne, czyli kernel. Możesz je zrestartować powyższym sposobem jeśli nie chcesz żeby były one zdefiniowane jak dotychczas.

**Czyszczenie konsoli IPython** → komenda clear (poza tym jeśli zrestartujesz kernel - jak wyżej - to nie tylko zrestartują się zmienne, ale też treść konsoli bocznej.

Np.

clear

**Wprowadzenie wcięcia (akapitu) do kodu** → komenda \t. Można je powielać, jedno po drugim.

Np.print(‘\tPython’)

print(‘\t\t\tPython’) → trzy akapity

**Ukryte komendy, a tekst**: co w przypadku kiedy w kodzie są ukryte komendy, a ma być tekst → trzy wyjścia:

Np.

print(‘C:path\to\something\new’) → zawiera \t i \n, co psuje efekt końcowy. W związku z tym można:

print(‘C:path\\to\something\\new’) → wstawić symbole ucieczki do newralgicznych miejsc

print(r’C:path\to\something\new’) → wstawić komendę r, czyli raw tekst i Python potraktuje zawartość jako surowy tekst

print(‘C:path\\to\\something\\new’) → wstawić symbole ucieczki wszędzie. To rekomendowane wyjście w tym przypadku, gdyż w ten sposób Python sam przechowuje nazwy ścieżek

**Przechowywanie ścieżek przez Python**: C:path\\to\\something\\new

**Formatowanie dzięki potrójnemu cudzysłowiu** → stosuje się kiedy nie chcę zwracać uwagę na formatowanie. Np. chcę napisać wstęp do programu, wyświetlaną informację dla użytkownika.

Np.

print(”””

Instrukcja uruchamiania pliku przyklad.py:

Instrukcja…

”””)

**Printowanie zadeklarowanego tekstu oraz powielanie tekstów** → na przykładzie:

Np.

text = ‘I love Python. ’

print(text \* 3)

# Poniżej oddzielenie kodu podkreślnikami, co jest przydatne gdy chcę żeby kod był bardziej czytelny

print(‘\_’ \* 10)

print(‘Nice! ‘ \* 3)

**Sklejanie tekstów w Python** → na przykładzie:

Np.

# Sprawi, że wyprintuje się tekst “Python”

print(‘Py’ ‘thon’)

**Dobra praktyka**: dobrą praktyką jest żeby linijki instrukcji nie były dłuższe niż 79 znaków. Instrukcje nie powinny wychodzić poza pionową linię widoczną w konsoli gdzie piszę kod.

**Url** → zmienna przechowująca adresu stron www

Np.

Url = ‘<https://www.udemy.com/course/programowanie-w-jezyku-python/learn/lecture/15714542?start=945#content>’

**Przełamywanie ciągu znaków przed 79 znakiem**: za pomocą sklejania tekstów. Zwróć uwagę, że sklejanie tekstów odbywa się w nawiasie, a zapis zmiennej url w “zwykły sposób” bez nawiasów.

Np.

url\_2 = ('https://www.udemy.com/course/programowanie-w-jezyku-python/'

        'learn/lecture/15714542?start=945#content')

**Łączenie zmiennych ze zwykłym tekstem** → za pomocą + lub ,. Różnica polega na tym, że jeśli chcę żeby słowa dzieliła spacja, to w wariancie z + muszę ją wstawić ręcznie, a w wariancie z , nie muszę w ogóle tego robić.

Np.

zmienna = 'Python'

print(zmienna + " 4.0")

print(zmienna,"4.0")

**# Powyższe dają taki sam wynik**

**Funkcja str** → zmiana wartości int na tekst (czyli string).

Np.

age = 26

print(str(age))

**Printowanie wielu różnych zmiennych** → za pomocą + lub ,. Jeśli to różne zmienne pod względem typu (int i str) to również można skorzystać z tych sposobów, ale w przypadku + należy skorzystać z funkcji str przy int. To znaczy, że jeśli konsolidujesz różne zmienne za pomocą przecinków to nie musisz używać funkcji, żeby zmienić typ zmiennych, a jeśli używasz plusów to musisz używać funkcji.

Np.

age = 25

imie = "Leszek"

nazwisko = "Dekiert"

print(imie + ‘ ‘ + nazwisko + ‘ = ‘ + str(age) + " lat.")

print(imie, nazwisko, ‘=’, age, ‘lat.’)

**# Powyższe dają taki sam wynik**

**Metoda .format →** metoda za pomocą której można przekazywać nazwy argumentów do kolejnych nawiasów kwadratowych (przykład poniżej).

**Indeksowanie** → przypisywanie zmiennym cyfr. Pamiętaj, że w Python przebiega ono od zera.

**Printowanie za pomocą zmiennych i metody format** → można również printować tekst za pomocą metody .format. Za pomocą nawiasów sześciennych pokaż ile i gdzie chcesz w tekście wstawiać dane zmienne. Do metody .format wstaw w nawiasie kolejność jako argumenty pierwszy, drugi…, a co do nawiasów kwadratowych możesz albo zostawić je puste, ale wtedy printują się tylko według kolejności argumentów czyli pierwszy nawias kwadratowy = pierwszy argument, itd. Albo możesz w nawiasach kwadratowych wpisać cyfrę który argument ma wyprintować - jest dodanie indeksowania (pamiętaj, że idzie od zera). Zwróć uwagę, że dzięki takiemu printowaniu nie musisz korzystać z + ani , kiedy piszesz kod. Wszystko jest już w apostrofie.

Np.

# %%

age = 30

name = "Leszek"

**# Da wynik “Leszek is 30 year's old!”**

print ('{} is {} year\'s old!'.format(name, age))

**# Da wynik “Leszek is 30 year's old!”**

print ('{0} is {1} year\'s old!'.format(name, age))

**# Da wynik “30 is Leszek year's old!”**

print ('{1} is {0} year\'s old!'.format(name, age))

**Lekcja\_4\_Obliczenia w Pythonie**

**Aktualizacja zmiennej liczbowej** → gdy chcę zaktualizować zmienną liczbową mogę to zrobić albo tworząc ją raz jeszcze dodając do niej daną wartość albo skorzystać z instrukcji +=. Odejmowanie działa analogicznie tylko z -=. Mnożenie \*=. Dzielenie /=. Potęgowanie \*\*=. Dzielenie modulo %=.

Np.

# %%

saldo = 40

saldo = saldo + 30

ALBO

saldo += 30

**Czynnik akumulacyjny** → wartość, która po przemnożeniu przez daną liczbę (np. lokatę) da wartość przyszłą.

**Zmienna float** → zmienna przechowująca liczby zmiennoprzecinkowe.

**Przykład obliczenia czynnika akumulacyjnego** (1 = oprocentowanie lokaty, 0.04 = stopa procentowa).

Np.

# %%

lokata = 1000

czynnik\_akumulacyjny = 1 + 0.04

lokata\_po\_roku = lokata \* czynnik\_akumulacyjny

print ('Lokata po roku' + ' = ' + str(lokata\_po\_roku))

print ('Lokata po roku', '=', lokata\_po\_roku)

**# Powyższe dają taki sam wynik**

**Dodawanie tekstu, tworzenie nowej zmiennej** → przykład. Pamiętaj żeby w takim przypadku pierwsza zmienna miała po sobie spację, inaczej wynik (nowa zmienna) będzie sklejony.

Np.

# %%

name = "Python "

version = "3.7"

full\_name = name + version

print(full\_name)

**Aktualizacja zmiennej tekstowej o kolejny człon** → przykład. Pamiętaj żeby w takim przypadku pierwsza zmienna miała po sobie spację inaczej wynik (nowa zmienna) będzie sklejona.

Np.

# %%

name = "Python "

version = "3.7"

name += version

# Printuje się “Python 3.7”

print(name)

**Lekcja\_5\_Wprowadzanie wartości przez użytkownika - funkcja input{}**

**Input** → funkcja, która pozwala na pobranie od użytkownika programu informacji. W programie Spyder w konsoli bocznej wpisz dane. czerwony kwadracik w tej konsoli oznacza, że program czeka aż wypełnisz dane i zakończysz jego działanie. Wzór: zmienna = input(). W argumencie może, ale nie musi być treść zachęcająca do wpisania określonej rzeczy.

Np.

# -\*- coding: utf-8 -\*-

name = input('Podaj swoje imię: ')

# Poniższe robią to samo

print ('Hi', name + '!')

print ('Hi {}!'.format(name))

**Input za inputem** → pamiętaj, że nawet jeśli oddzielisz fragmenty kodu to możesz tworzyć zestawienia inputów i uruchamiać je za pomocą F5, który uruchamia cały kod

Np.

# -\*- coding: utf-8 -\*-

# %%

name = input('Podaj swoje imię: ')

print ('Hi {}!'.format(name))

# %%

jezyk = input('Jakiego języka chciałbys sie nauczyc: ')

print ('Chcesz nauczyc sie jezyka {}.'.format(jezyk))

**Input i string** → input przekazuje wszystkie podane, nawet te liczbowe jako string, czyli tekst. Aby było inaczej należy skorzystać z funkcji int() - wtedy zapisze podaną liczbę jako cyfra.

Np.

# %%

name = input ('Podaj swoje imie: ')

age = int(input ('Podaj swoj wiek: '))

job = input ('Podaj swoj zawod: ')

print('Hi {2}! I see You are {1} year\'s old. Also Your occupation is: {0}.'.format(job, age, name))

**Lekcja\_6\_Konwencja nazewnictwa zmiennych**

**Zmienne wewnętrzne dotyczące Pythona** → zaczynają się one od podkreślników jednego lub dwóch, tak się je zapisuje. Są to zmienne, których nie powinno się ruszać przez co nie ma ich wyprintowanych w zmiennych środowiskowych. Jeśli jednak wpiszesz tą zmienną w konsoli to jej wartość będzie wyrzucona, podana.

Np.

\_imie - ‘Leszek’

**Nie można zaczynać nazwy zmiennej od cyfr →** można cyfrę dać na koniec.

**Pro tip →** pisząc kod i deklarując zmienne dla własnego ułatwienia i przede wszystkim innych deklaruj je raczej tak, żeby było wiadomo czego dotyczą, np. “stawka\_godzinowa”, a nie “a”. Ponadto deklarując zmienne które wymagają spacji rób to za pomocą podkreślnika.

**Style przypisywania zmiennych** → jest ich parę, odnoszą się do zapisywania wielosłownych zmiennych. Każdy generalnie przyjmuje jeden i się go trzyma:

1. camelCase → pierwsza litera mała druga duża
2. PascalCase → wszystkie wyrazy z dużej litery
3. snake\_case → łączenie podkreślnikiem i wszystkie wyrazy małą literą
4. kebab-case → łączenie myślnikiem
5. UPPER → wszystkie literki z dużej litery

Generalnie używa się snake\_case do nazw, metod i zmiennych, a UPPER do zmiennych stałych czyli wartości, które się nie zmieniają w trakcie uruchamiania programu.

**Pro tip** → nie należy deklarować zmiennych, które są zarezerwowane dla Pytona. Np. “print”. Aby uzyskać całą listę takich nazw skorzystaj z atrybutu keyword.kwlist (import keyword[ENTER] keyword.kwlist).

Np.

# %%

import keyword

keyword.kwlist

# Po uruchomieniu w konsoli bocznej pojawią się te nazwy.

**Lekcja\_7\_Operator wycinania**

**Wycinanie danych ze string** → odbywa się za pomocą indeksu, który wstawiasz na końcu print, za nazwą zmiennej. Jeśli chcesz wycinać patrząc od początku wartości to zaczynasz od 0 (pierwsza literka), a jeśli od końca to od -1 (ostatnia), potem -2 (przedostatnia), itd.

Np.

# -\*- coding: utf-8 -\*-

name = 'Leszek'

# %%

print (name[0])

print (name[-5])

print (name[2])

print (name[-3])

print (name[4])

print (name[-1])

# Wynikiem będzie “Leszek”

**Wycinanie zakresu** → podaję zakres, który działa tak, że pierwsza podana wartość (początek zakresu) wpada do zakresu, a ostatnia nie wpada (koniec zakresu). Jest tak dlatego, że wycinanie działa na zasadzie lewostronnego zamknięcia. Inaczej: w zakresie [2:5] wycięta zostanie trzecia (bo od 0 się zaczyna) litera, czwarta i piąta. Szósta już nie. Pozostawiając pola zakresu puste sugerujesz, że albo wycinać wszystko od początku do… albo od… do samego końca. Pamiętaj, że wyciąganie wartości od drugiej strony działa tak, że wybierasz znak który interesuje cię od końca, np. trzeci, to wpiszesz [-3:] i program ci odda znaki od 3 ostatniego (przed,przedostatniego) w górę do ostatniego.

Np.

# -\*- coding: utf-8 -\*-

name = 'Leszek'

# %%

print (name[0:5])

# Wyprintuje “Lesze”

print (name[1:6])

# Wyprintuje “eszek”

print (name[:5])

# Wyprintuje “Lesze”

print(name[1:])

# Wyprintuje “eszek”

print (name[:])

# Wyprintuje wszystko

**Wyciąganie co którąś wartość** → używasz trzeciego argumentu (opcjonalnego) indeksacji. Wygląda to chociażby tak: [::3] - w tym przypadku wyciągasz co 3 wartość (tu nie działa, że od 0). Pamiętaj, że wycinana nie będzie wartość od 3, a potem 6, 9, itd. tylko zacznie od pierwszej, czyli będzie pierwsza litera,  czwarta, siódma…

Np.

# -\*- coding: utf-8 -\*-

name = 'Leszek Dekiert'

# %%

print (name[::2])

# Wyprintuje Lse eir, bo co druga literka

**Wyciąganie odwrócone** → za pomocą ujemnego trzeciego argumentu. Jeśli powiedzmy mam ciąg liczb 1,2,3,4,5,6,7,8,9,10, i chcę go wyprintować od końca do początku, to używam minus jeden. Uwaga: zwróć uwagę, że 10 nie wyprintuje się w takim wypadku jako 10, ale jako 01 bo idzie od końca (chyba, że te liczby będą w liście). Jeśli chcesz od końca co drugi, trzeci, to tak wpisz.

Np.

# %%

numbers = ('1,2,3,4,5,6,7,8,9,10')

print (numbers[::-1])

# Wyprintuje 01,9,8,7,6,5,4,3,2,1

**Operator in** → używany chociażby do sprawdzenia czy w mojej zmiennej jest dana wartość. Pamiętaj, że uwzględnia duże litery. Python odpowie na zasadzie True lub False.

Np.

# %%

program\_name = ('Python')

'P' in program\_name

# Zwraca True

**Lekcja\_8\_Typy danych**

**String** → typ danych tekstowych.

**Dir** → funkcja wyrzucająca do konsoli wszystkie atrybuty i metody dotyczące obiektu.

Np.

# %%

string = ‘Python’

print (dir(string))

**Pro tip** → wszystko w Python jest obiektem. Są obiekty różnych klas. Na obiektach tych samych klas możemy stosować te same metody. Teraz wiedz, że te metody/atrybuty z dwoma podkreślnikami to bardziej skomplikowane rzeczy, teraz cię nie interesują.

**Integer** → zmienna liczbowa.

**Type** → funkcja zwracająca typ zmiennej, w konsoli pomocniczej. Wprowadza dane do eksploratora zmiennych.

Np.

# %%

a = 10

print(type (a))

**Float** → typ zmiennych zmiennoprzecinkowych. Pozwala przechowywać wszystkie liczby rzeczywiste.

Np.

# %%

b = 4.5

type(b)

**Complex** → liczby zespolone, rzadko używany typ danych.

Np.

# %%

d = 3 + 3j

type(d)

**Typ danych logicznych (bool)** → TRUE/FALSE. W Python pisane z dużej litery.

Np.

# %%

type(True)

# Zwróci treść “bool” co oznacza, że jest to zmienna logiczna.

**Sprawdzanie szybko czy zmienna to zmienna logiczna (bool)** → w konsoli pomocniczej. W przykładzie zmienna nazywa się “value”.

# %%

type(True)

value = True

print (value)

# w konsoli pomocniczej wpisz type (value) - zwróci bool

**Pro tip** → W Spyder treść żółta oznacza specjalny ciąg znaków dla Pythona, która nie powinna być nazwą żadnej zmiennej.

**Type i format** → jeśli chcę wyprintować w jedno pod drugim typ określonych zmiennych. Funkcja Type znajdzie się w funkcji Format.

Np.

# -\*- coding: utf-8 -\*-

x = '1323435'

y = 12334

z = 'O'

print ('x: {0}\ny: {1}\nz: {2}'.format(type(x), type(y), type(z)))

\*Wynik to jedno pod drugim:

x: <class 'str'>

y: <class 'int'>

z: <class 'str'>

# Nie możesz przy pomocy type wstawić wszystkich sprawdzanych zmiennych - musisz to zrobić pojedynczo, tak jak w przykładzie powyżej

**Lekcja\_9\_Dane tekstowe i metody**

**Metody** → funkcje, które pozwalają robić przekształcenia obiektów. Pisanie metod: zmienna.metoda(), np. text.title().

**Łączenie metod** → jeśli chcesz zastosować wiele metod naraz możesz to zrobić po prostu dodając kolejne metody w ciągu, po kropkach.

Np.

string = 'Programowanie w języku Python - od A do Z'

string = string.lower().replace('ę', 'e').replace(' ', '').replace('-', '')

**Pamiętaj o funkcji dir** → wyrzuca ona wszystkie dostępne metody. W kursie omawiane są najczęściej spotykane.

**Funkcja help** → pozwala na dostanie się do dokumentacji Pythona, do tego jak działa konkretna metoda.

Np.

text = 'Witaj na kursie Pythona.\nPython jest wspaniały.'

dir (text)

help (str.count)

# powyżej najpierw określiłem jaka to zmienna, potem sprawdziłem dostępne metody i wybrałem jedną z nich - tutaj była to metoda count

**Metoda count** → zwraca liczbę powtórzeń w tekście. Tutaj musisz podać podciąg, czyli jakiego ciągu znaków szukasz.

Np.

text = 'Witaj na kursie Pythona.\nPython jest wspaniały.'

text.count('Python')

# zwraca liczbę 2 bo tyle razy ten ciąg znaków występuje.

**Metoda capitalize** → zwraca tylko pierwszą literę dużą w całym tekście, ignorując pozostałe duże litery. Zwróć uwagę, że metody podobnie jak funkcje wymagają nawiasu w celu wywołania tekstu. Chodzi o to, że ta metoda sprawi, że w tekście, w którym jest +2 wielkie litery, teraz będzie tylko jedna, pierwsza. Patrz przykład.

Np.

text = 'Witaj na kursie Pythona.\nPython jest wspaniały.'

text.capitalize()

# zwróci 'Witaj na kursie pythona.\npython jest wspaniały.'

**Metoda upper** → zwraca każdy znak z dużą literą.

Np.

text = 'Witaj na kursie Pythona.'

text.upper()

**Metoda title** → zwraca każdy wyraz w tekście jako tekst z dużą literą.

Np.

text = 'Witaj na kursie Pythona.\nPython jest wspaniały.'

text.title()

**Metoda lower** → zwraca każdy wyraz z małymi literami.

Np.

# %%

A = 'Leszek'

A.lower()

**Metoda startswith** → metoda logiczna, czyli zwraca True / False. Sprawdza czy dany ciąg znaków zaczyna się od podanego ciągu znaków.

Np.

# %%

text = 'Witaj na kursie Pythona.\nPython jest wspaniały.'

text.startswith('We')

# Tutaj zwróci False

text.startswith('Wi')

# Tutaj zwróci False

Powyższa metoda działa też na tekście bez przypisania zmiennej.

Np.

‘python’.startswith(‘py’)

# Zwraca True

**Metoda endswith** → metoda analogiczna do powyższej. Sprawdza czy ciąg znaków kończy się na coś. Metoda ta jest przydatna przy sprawdzaniu czy nazwy plików kończą się z określonym rozszerzeniem.

Np.

# %%

‘sample.py’.endswith(‘.py’)

# Zwraca True

**Metoda find** → metoda określa położenie szukanego ciągu znaków w postaci numeru indeksu. Pamiętaj, że indeksy liczysz od zera!

Np.

text = 'Witaj na kursie Pythona.\nPython jest wspaniały.'

text.find('Python')

wróci 16 bo pierwsze słowo Python zaczyna się od 16 znaku

Powyższą metodę można wykorzystać do wycinania tekstu.

Np.

text[text.find('Python'):]

# Zwróci 'Pythona.\nPython jest wspaniały.'

Albo wycinanie w drugą stronę

# %%

hashtags = "sport#gym"

inx = hashtags.find('#')

hashtags[:inx]

# Zwraca “Sport”

**Metoda isalnum** → sprawdza czy w znakach znajdują się tylko liczby lub litery. Jeśli pojawi się np. spacja lub wykrzyknik to zwróci FALSE. Chodzi o wykrywanie znaków innych niż cyfry czy litery - te mogą być. Pamiętaj: ta metoda sprawdza obecność liczb lub liter. Lub.

Np.

imie.isalnum()

# Zwróci TRUE

imie\_!.isalnum()

# Zwróci FALSE

**Metoda isdigit** → sprawdza czy wszystkie znaki to cyfry. Mogą być w cudzysłowiu lub luzem. Pamiętaj: metoda ta sprawdza występowanie cyfr w tekście.

Np.

# %%

digit = '2345'

digit.isdigit()

**Metoda isalpha** → sprawdza czy wszystkie znaki to litery. Pamiętaj: metoda ta sprawdza występowanie liter w tekście.

Np.

# %%

alpha = '2345'

alpha.isaplha()

**Metoda islower** → sprawdza czy wszystkie znaki to wielkie litery

**Metoda isupper** → sprawdza jeśli wszystkie litery są z dużej litery

**Metody szukania chociaż jednej cyfry, małej litery, dużej, etc.** → należy wykorzystać funkcję logiczną any i pętlę. Przykład poniżej.

Np.

password = 'my\_password\_123'

if len(password) <= 8:

print('The Password is too short.')

elif any(char.isdigit() for char in password) != True:

print('The password must contain at least one digit.')

elif any(char.isupper() for char in password) != True:

print('The password must contain at least one uppercase letter.')

elif any(char.islower() for char in password) != True:

print('The password must contain at least one lowercase letter.')

else:

print('The password is complex enough.')

**Metoda join** → metoda łącząca teksty. Łączy się poszczególne elementy za pomocą listy. Na początku metody wpisujesz czym chcesz żeby były łączone (np. spacją). Ale może to być cokolwiek. Pamiętaj, że w nawiasie metody poszczególne elementy muszą znajdować się w nawiasie kwadratowym.

Np.

# %%

‘ ‘.join([‘python’, ‘3.7’])

# W tym przykładzie na początku jest ‘ ‘, a zatem poszczególne elementy będą łączone spacją.

**Join, a lista** → biorąc pod uwagę, że poszczególne elementy w argumentach metody join muszą znajdować się w nawiasie kwadratowym, to jeśli obiekty już znajdują się na liście, przywoływanie zmiennej będącej listą nie wymaga umieszczania jej w nawiasie kwadratowym.

Np.

chronic\_conditions = ['hypertension', 'diabetes']

f"Chronic conditions: {','.join(chronic\_conditions)}\n"

**Metoda replace** → zastępuje znaki innymi. Pierwszy argument to znak, który podmieniasz, a drugi to znak na jaki podmieniasz. Pamiętaj jednak, że ta operacja dzieje się tylko na moment przerabiania przez Python linijki kodu z Replace. Jeśli chcesz żeby zmiana była stała musisz dodać, że nowa zmienna = stara zmienna.replace(‘cos’,  ‘cos nowego’)

Np.

# %%

'tytul kolumny pierwszej'.replace(' ', '\_')

# Wynik: tytul\_kolumny\_pierwszej

**Metoda strip** → wycina białe znaki (niepotrzebne spacje) z zakresu. Weź jednak pod uwagę, że usuwanie dotyczy skrajnych części kodu. Np. z '              55           dsadsa         eteteretr           ' po strip zostanie ‘55           dsadsa         eteteretr’

Np.

'     python     '.strip()

**Metoda rstrip** → wycina białe znaki (niepotrzebne spacje) z prawej zakresu.

Np.

'     python     '.rstrip()

**Metoda lstrip** → wycina białe znaki (niepotrzebne spacje) z lewej zakresu.

Np.

'     python     '.lstrip()

**Metoda split** → metoda wyciąga wszystkie obiekty z ciągu znaków po podaniu w argumencie co dzieli interesujące cię obiekty, które znajdą się w liście. Jeśli nie podasz w argumencie niczego to Python sam domyślnie podzieli obiekty (przykład ze spacjami). Każdy element między wybranym separatorem będzie pojedynczym obiektem i znajdzie się w liście []. Pamiętaj, że sama lista to tylko lista w konsoli, a jeśli chcesz żeby została zapisana musisz przypisać wynik wykorzystania metody split do nowej lub starej zmiennej.

Np.

'1,2,3,4,5'.split(',')

# przygotuje listę ['1', '2', '3', '4', '5'] - pamiętaj, że będą to stringi

'python java php sql sas'.split()

# Przykład domyślnego dzielenia

'#gym#fit#sport'.split('#')

**Metoda zfill** → wypełnia od lewej zera po określeniu ile ciąg ma mieć w sumie znaków. Przydatne do uzupełniania/poprawiania danych do naszych potrzeb. Przykładowo jakieś ID ma mieć 5 znaków liczbowych, ale użytkownicy wpisywali 2, 13, 156 zamiast 00002, 00013, 00156, etc. Pamiętaj, że zmodyfikowany obiekt nie będzie liczbą, dlatego przykłady są w nawiasie.

Np.

# %%

'1'.zfill(5)

# Daje wynik 00005

'12'.zfill(10)

**SEKCJA 3. STRUKTURY DANYCH**

**Lekcja\_10\_Zbiory**

**Zbiory** → proste struktury danych. Zbiór przechowuje nieuporządkowany ciąg elementów i jednocześnie nie ma w sobie zduplikowanych wartości. Zbiory znajdują się w nawiasach sześciennych.

**Różnice między zbiorami, tuplami i listami:**

Zbiór (set) = {}. Dane nieuporządkowane. Można je modyfikować. Brak duplikatów. Różne typy zmiennych = tak.

Tupla (tuple = (). Dane uporządkowane. Nie można ich modyfikować. Duplikaty dozwolone.Różne typy zmiennych = tak.

Lista (list) = []. Dane uporządkowane. Można je modyfikować. Duplikaty dozwolone. Różne typy zmiennych = tak.

**Tworzenie zbioru pustego** → na przykładzie. Nie doda się nic do eksploratora zmiennych. Jeśli skorzystasz z funkcji type to Python pokaże, że jest to obiekt typu set.

Np.

# %%

empty\_set = set()

print(type(empty\_set))

**Konwertowanie listy na zbiór** → jeśli chcesz żeby uporządkowana lista stała się nieuporządkowanym zbiorem. Na zasadzie przykładu:

Np.

# %%

fakt = 'python jest łatwy i przyjemny'

lista = list(fakt)

zbiór = set (lista)

# Najpierw stworzyłem listę, potem zbiór

**Prosty zbiór wartości tekstowych** → wypisuje się go w nawiasie sześciennym. Kolejność nie jest istotna. Jeśli nawet podasz pewną kolejność to może wyprintować się w innej kolejności, to nie ma znaczenia. Chodzi tylko o to czy dana zawartość tam jest.

Np.

techs = {‘python’, ‘C++’, ‘sql’, ‘java’}

print (techs)

# wyprintuje {'C++', 'python', 'sql', 'java'}, czyli w innej kolejności

**Funkcja len** → funkcja zwracająca wartość ile jest elementów w zbiorze. Len pochodzi od length.

Np.

techs = {‘python’, ‘C++’, ‘sql’, ‘java’}

print(len(techs))

**Funkcja set** → ma m.in. możliwość rozdzielania stringa na pojedyncze znaki i umieszczenie ich w zbiorze. Zwróci tylko unikalne znaki, więc jeśli użyjesz tego na ‘Leszek’ to ‘e’ pojawi się tylko raz.

Np.

# %%

set ('Leszek')

# Zwróci {'L', 'e', 'k', 's', 'z'}, bo zwraca tylko unikalne znaki

**Operator in** → dzięki niemu można sprawdzić czy ciąg znaków jest w zbiorze, tupli lub liście.

Np.

# %%

techs = {'python', 'C++', 'sql', 'java'}

'python' in techs

# True

'r' in techs

# False

**Metoda add** → pozwala na dodawanie obiektów do zbiorów. Możesz dodać jeden obiekt.

Np.

# %%

names = {'Leszek', 'Aga'}

print (names)

names.add('Franciszka')

**Metoda remove** → metoda pozwala na usuwanie elementów ze zbioru. Możesz usunąć jeden obiekt.

Np.

# %%

names = {'Leszek', 'Aga', ‘Franciszka’}

names.remove('Leszek')

print (names)

**Metoda pop** → metoda wyrzuca pojedynczo, randomowo obiekty ze zbioru. Uruchamiaj kod wystarczająco dużo razy, a w zbiorze nic nie zostanie. Żeby działać poprawnie, metoda pop będzie uruchamiana w innej, oddzielonej części kodu od definiowania zmiennej i dodawania do niej elementów. Inaczej usuwać się będzie ciągle jeden element i przywracać.

Np.

# %%

cutlery\_and\_crockery = {'glass', 'cup', 'knife', 'spoon'}

# %%

cutlery\_and\_crockery.pop()

**Metoda clear** → metoda czyści wartości ze zbioru.

Np.

# %%

cutlery\_and\_crockery = {'glass', 'cup', 'knife', 'spoon'}

cutlery\_and\_crockery.clear()

print(cutlery\_and\_crockery)

**Metoda issubset** → sprawdza czy dany zbiór liczb zawiera się w innym zbiorze. Dwa sposoby posługiwania się metodą. Podane nazwy zbioru lub wpisanie zbioru dosłownie. Sprawdza czy jest podzbiorem. Wzór: zmienna szukana w innym.issubset(zmienna w której się szuka). Tylko do liczb - dla string nie zadziała. Działa przy posługiwaniu się nawiasami sześciennymi, również przy podawaniu liczb “luzem” bez przypisanej zmiennej.

Np.

# %%

A = {1, 2, 3, 4, 5, 6, 7}

B = {5, 6, 7, 8, 9}

C = {5, 6}

C.issubset(A)

# True

C.issubset({5, 7})

# False

**Metoda issuperset** → metoda odwrotna do issubset. Sprawdza czy dany zbiór nadrzędny zawiera podzbiór. Teraz zamiast najpierw mniejszego zbioru podajesz większy.

Np.

# %%

A = {1, 2, 3, 4, 5, 6, 7}

B = {5, 6, 7, 8, 9}

C = {5, 6}

A.issuperset(C)

# True

**Sprawdzanie czy ciąg zawiera literę →** wykorzystaj operator in. Jak w przykładzie poniżej:

Np.

text = 'sfdvjklncdnskjccbnksjdnckjsdsnckjnsdkjnckjsnkjlcnqdlknwsx'

if 'q' in text:

print('The text contains the letter "q".')

else:

print('The text does not contain the letter "q".')

**Metoda union** → metoda sumowania zbiorów. Powtarzające się liczby nie powielają się, bo to zbiory. Kolejność podawania zmiennych jest tu więc nieistotna. Możesz też łączyć za pomocą “luźnych” ciągów liczb w nawiasach kwadratowych.

Np.

# %%

A = {1, 2, 3, 4, 5, 6, 7}

B = {5, 6, 7, 8, 9}

A.union(B)

# {1, 2, 3, 4, 5, 6, 7, 8, 9}

**Metoda intersection** → pozwala na pokazanie przecięcia się zbiorów. Czyli wartości wspólnych dla zbiorów. Kolejność podawania zmiennych we wzorze nie ma znaczenia. Możesz też korzystać z “luźnych” ciągów liczb, ale w tym przypadku to bez sensu.

Np.

# %%

A = {1, 2, 3, 4, 5, 6, 7}

B = {5, 6, 7, 8, 9}

C = {5, 6}

A.intersection(B)

# {5, 6, 7}

**Metoda symmetric\_difference** → metoda wyciąga różnicę symetryczną dla zbiorów. Oznacza to, że jeśli skorzystam w metodzie ze zbioru A i B to te liczby, które są w zbiorze A i nie są w zbiorze B to zostaną wyciągnięte i dodatkowo te elementy, które są w zbiorze B, ale nie A też zostaną wyciągnięte. Czyli po prostu wyrzuca różnice między zbiorami nawzajem. Można korzystać z “luźnych” ciągów liczb.

Np.

# %%

A = {1, 2, 3, 4, 5, 6, 7}

B = {5, 6, 7, 8, 9}

C = {5, 6}

A.symmetric\_difference(B)

# {1, 2, 3, 4, 8, 9}

**Metoda copy** → kopiuje zbiór. Korzystaj raczej razem z definiowanej innej zmiennej, takie jest m.in. zastosowanie.

Np.

# %%

A = {1, 2, 3, 4, 5, 6, 7}

B = {5, 6, 7, 8, 9}

C = {5, 6}

D = A.copy()

print (D)

# D = {1, 2, 3, 4, 5, 6, 7}

**Lekcja\_11\_Tuple**

**Tupla** → uporządkowana struktura, której nie można zmieniać. Raz włożone elementy do tupli nie mogą być zmienione. Można stworzyć pustą tuplę. Tuple tworzy się za pomocą nawiasów zwykłych. Tupla będzie miała taką wielkość ile ma obiektów w sobie.

Np.

# -\*- coding: utf-8 -\*-

empty\_tuple = tuple()

print (empty\_tuple)

**Definiowanie tupli za pomocą nawiasów okrągłych** → po prostu po przecinku. W tupli mogą być różne wartości: string, integer… Jest to uporządkowany ciąg elementów, jakby wiersze w excel. Na przykładzie Firma, Kraj, Branża, Numer. Kliknij dwukrotnie na wartość w eksploratorze zmiennych żeby podejrzeć z czego tupla się składa. Tupli nie można zmieniać. Można je wycinać.

Np.

# %%

amazon = ('Amazon', 'USA', 'Technology', 1)

google = ('Google', 'USA', 'Technology', 2)

**Definiowanie tupli bez nawiasów okrągłych** → j/w, ale bez nawiasu. Też zadziała.

Np.

# %%

stocks = 'Coca Cola', 'IBM', 'Apple'

**Wycinanie z tupli i indeksy** → indeksy działają w tuplach jak w Python. Zaczynają się od 0. Oznacza to, że jeśli wartość pierwszą w kolejności (Google) chciałbym przypisać innej, nowej zmiennej mogę to zrobić wywołując pierwszy numer indeksu. Ale jeśli chciałbym nadpisać tą wartość w tupli inną, Python mi na to nie pozwoli.

Np.

# %%

google = ('Google', 'USA', 'Technology', 2)

name\_google = google[0]

# tworzy nową zmienną o treści “Google”

google[0] = 'Google Company'

# błąd: 'tuple' object does not support item assignment

**Zagnieżdżanie tupli** → za pomocą dwóch lub więcej tupli mogę tworzyć większe tuple. Są to tuple zagnieżdżone. Tupla z dwóch tupli będzie długości 2, ale każda tupla wewnątrz będzie długości 4. Mogę te elementy mocno zagnieżdżać, dużo ich dodawać, ale nie powinno być ich zbyt dużo.

Np.

# %%

amazon = ('Amazon', 'USA', 'Technology', 1)

google = ('Google', 'USA', 'Technology', 2)

data = (amazon, google)

**Przypisywanie zmiennych, a tuple** → za pomocą tupli możesz dodawać szybko zmienne, które normalnie zdefiniowałbyś jedna pod drugą. Mogą to być zmienne różnego typu. Wzór = nazwa zmiennej 1, 2, 3, 4… = (wartość przypisana zmiennej 1, 2, 3, 4…)

Np.

# %%

imie, nazwisko, id\_user = ('Leszek', 'Dekiert', 1)

# Każda zmienna doda się jako osobna (str, str, int) zamiast jako jedna tupla

**Rozpakowywanie tupli** → można rozbijać istniejące tuple (czyli ciągi obiektów) na pojedyncze zmienne. Jeśli tupla ma długość 4 (tyle zmiennych) to podajesz po przecinku 4 nazwy nowych zmiennych, które będą im przypisane. Wzór: nowa zmienna 1, 2, 3, 4… = zmienna stara.

Np.

# %%

amazon = ('Amazon', 'USA', 'Technology', 1)

amazon\_name, country, sector, rank = amazon

**Tuple zagłębione** → można umieszczać tuple w tuplach, będą to tuple zagłębione.

Np.

# %%

nested = 'Europa', 'Polska', ('Warszawa', 'Krakow', 'Wroclaw')

# Tupla “nested” ma długość 3 bo ma 3 obiekty, z czego 1 jest po prostu zmienną typu tuple, podczas gdy dwie wcześniejsze to zmienne typu string.

**Tuple zagnieżdżone, a zagłębione** → to pierwsze oznacza łączenie tupli w jedną tuplę. Drugie oznacza definiowanie tupli za pomocą np. luźnych wartości i jednego obiektu w nawiasie, który automatycznie jest tuplą.

# %%

example\_1 = (1, 2, 3)

example\_2 = (4, 5, 6)

example\_3 = (7, 8, 9)

examples = (example\_1, example\_2, example\_3)

another\_example = (10, 11, example\_1)

yet\_another = (12, 13, (14, 15, 16))

# wszystkie powyższe mają wartość 3

**Tuple i zamiana wartości zmiennych** → zdarza się, że trzeba zaktualizować dane, ale tak, żeby zamienić wartości zmiennych między sobą, na przykład a i b. Są co najmniej dwa sposoby na to: jeden nieelegancki i wymagający stworzenia nowej zmiennej i jeden prosty, z tuplami.

Np.

# %%

a = 10

b = 20

c = b

b = a

a = c

print (a, b)

# ALBO

# %%

x, y = 30, 50

x, y = y, x

print (x, y)

# Drugi sposób prostszy

**Lekcja\_12\_Listy**

**Listy** → są to zbiory danych (zmiennych), w których mogą być duplikaty oraz, które można modyfikować. Inaczej niż w tuplach. Poniżej tworzenie pustej listy i wypełnionej listy. Zapisuj listy w nawiasach kwadratowych - jeśli skorzystasz z nawiasów to program stworzy tuplę, a to nie lista.

Np.

# -\*- coding: utf-8 -\*-

empty\_list = list()

techs = ['python', 'java', 'c++', 'go', 'sql']

print (techs[0])

# wyprintuje “python”

**Modyfikacja list** → w prosty sposób jak modyfikacja normalnych zmiennych, tylko podaj numer indeksu.

Np.

# -\*- coding: utf-8 -\*-

empty\_list = list()

techs = ['python', 'java', 'c++', 'go', 'sql']

techs[0] = 'Python 3.7'

print (techs[0])

**Różne typy zmiennych w listach** → w listach, podobnie jak w tuplach można umieszczać różne typy zmiennych.

Np.

# %%

mixed = ['python', 3.7, 4, True]

**Szybkie tworzenie pustej listy** → wystarczy nawias kwadratowy.

Np.

# %%

empty = []

**Tworzenie listy złożonej ze wszystkich elementów obiektu** → kiedy chcesz żeby np. string był podzielony na poszczególne litery. Wzór: zmienna\_w\_całości zmienna\_w\_częściach = list(zmienna\_w\_całości).

Np.

# %%

fakt = 'python jest łatwy i przyjemny'

lista = list(fakt)

**Zagnieżdżanie list** → listy można zagnieżdżać w sobie, ale należy pamiętać i kontrolować poziomy zagnieżdżania, czyli nawiasy kwadratowe. Zwróć uwagę, że w eksploratorze zmiennych wchodzisz głębiej i głębiej w specyfikacje list, aż dojdziesz do końca.

Np.

# %%

nested = [[1, 2, [3, 'sql']], ['druga zagnieżdżona lista']]

# Lista w liście, w liście

**Łączenie list** → po prostu za pomocą nawiasu kwadratowego. To jednak stworzy listę, w której zagnieżdżone będą dwie inne listy.

# %%

first = ['chleb',  'bulki', 'maslo']

second = ['woda', 'jajka']

bucket = [first, second]

**Funkcja len c.d.** → funkcja len pozwala na sprawdzanie długości listy i tupli.

Np.

# %%

first = ['chleb',  'bulki', 'maslo']

second = ['woda', 'jajka']

bucket = [first, second]

# %%

len(bucket)

**Dodawanie tymczasowe i stałe do listy** → możesz dodać element do listy jako część zadania na chwilę albo zrobić to na stałe. Dwa sposoby poniżej. Żeby dodać element na stałe musisz posłużyć się nową zmienną lub skorzystać z +=.

Np.

# %%

techs + ['R']

# “Działanie” dodawania tylko na czas działania

techs\_including\_R = techs + ['R']

# “Działanie doda do wszystkiego w nowej zmiennej R

# %%

techs += ['R']

# Dodaje do listy zmiennych

**Lekcja\_13\_Listy - wycinanie**

**Wycinanie** → istnieje kilka sposobów na wycinanie różnych danych z list. Pamiętaj, że wycinanie w Python odbywa się za pomocą indeksów (od zera) - patrz przykład poniżej. Poza tym pamiętaj, że wycinanie jest lewostronnie domknięte, co oznacza, że przy wycinaniu typu start:stop, wycięta zostanie wartość start, ale zatrzyma się przed stop - stop nie zostanie wycięte.

Np.

# -\*- coding: utf-8 -\*-

index\_normal =   [ 0,    1,   2,   3,   4,  5]

index\_negative = [-6,  -5,  -4,  -3, -2,  -1]

example =           [34, 23, 56, 24, 23, 76]

**Wycinanie pojedynczego elementu** → przykład. Wyciągnaie pierwszego obiektu to po prostu 0, bo indeksowanie. Ale wycinanie ostatniego obiektu musi już odbyć się za pomocą -1, bo właśnie od -1 zaczynają się końcowe obiekty.

Np.

# %%

example[0]

# wyprintuje 34, czyli pierwszy obiekt

example[-1]

# wyprintuje 76, czyli ostatni obiekt

**Wycinanie wszystkich elementów** → na zasadzie [start:].

Np.

# %%

example[0:]

**Wycinanie do określonego elementu** → na zasadzie [:stop], pamiętając, że nie wytnie stop, tylko przed stop.

Np.

# %%

example[:2]

**Wycinanie co któryś element** → na zasadzie [::co który element]. Można też to robić oczywiście od końca. Pamiętaj jednak, że ten trzeci argument nie podlega już indeksowaniu Pythonowemu. Oznacza to, że zero nie jest przeskakiwaniem co 1, tylko po prostu zwróci błąd. W tym trzecim argumencie przeskakiwanie co 1 wpisujesz jako jeden, co 2 jako dwa, itd. Nadal możesz też korzystać z pozostałych dwóch pierwszych argumentów jeśli chcesz np. zacząć od któregoś wycinanie, ale na którymś skończyć.

Np.

# %%

example[::2]

# Wyciąga wszystko co drugi zaczynając od domyślnego czyli pierwszego

example[1::2]

# Wyciąga wszystko co drugi zaczynając od drugiego

**Wycinanie od końca** → j/w tylko z minusem. Odwrotna jest też sytuacja z rozpoczęciem od końca od określonego obiektu. Tutaj, żeby to działało wpisujesz ostatniego interesującego obiektu od którego wstecz będzie szło wycinanie co którąś wartość, w pierwszym argumencie.

Np.

# %%

example[::-2]

# Wyciąga wszystko co drugi zaczynając od domyślnego ostatniego.

example[-2::-2]

# Wyciąga wszystko co drugi zaczynając od przedostatniego.

**Odwrócenie listy** → jeśli chcesz odwrócić kolejność wszystkiego to użyj [::-1]

Np.

# %%

example[::-1]

**Metoda .index** → metoda pokazująca, pod którym numerem indeksu znajduje się na liście określona wartość. Jest to tak jakby odwrotność printowania wartości za pomocą numeru indeksu.

Np.

# -\*- coding: utf-8 -\*-

techs = ['java', 'python', 'spark', 'hardoop', 'r']

print (techs.index('python'))

# Wyprintuje 1

**Podmiana wartości** → można to bardzo szybko zrobić jak na przykładzie poniżej.

Np.

# -\*- coding: utf-8 -\*-

techs = ['sql', 'python', 'r', 'php', 'C++']

techs[1] = "angular"

# Zmiana python na angular

**Różnica między metodami, a funkcjami** → Metoda Pythona jest wywoływana na obiekcie, w przeciwieństwie do funkcji. Ponieważ wywołujemy metodę na obiekcie, może ona uzyskać dostęp do zawartych w niej danych. Metoda może zmienić stan obiektu, ale funkcja Pythona zwykle działa tylko na nim, a następnie coś drukuje lub zwraca wartość. Funkcje zwracają określone wartości, metody mogą je modyfikować. Funkcje działają tak, że jest nazwa\_funkcji (wartość i ewentualne argumenty), a metody, że jest wartość.metoda(ewentualne argumenty).

**Wycinanie obiektów zagnieżdżonych**  → jeśli chcę wyciągnąć obiekt, który ukryty jest w zagnieżdżeniu, muszę skorzystać z nawiasów kwadratowych. Chodzi o to, że jeśli chcę wyciągnąć coś z listy zagnieżdżonej w liście (czyli w sumie 2 poziomy skomplikowania) to wypisuję obok siebie dwa nawiasy kwadratowe, które mnie do obiektu nawigują. Jak w przykładzie poniżej.

Np.

fruits = [['apple', 'banana'], ['cherry', 'orange'], ['kiwi', 'melon']]

print('Nested list:', fruits)

print('First item of second nested list: {}'.format(fruits[1][0]))

**Lekcja\_14\_Listy - metody**

**Metoda append** → pozwala na dodawanie wartości lub zagnieżdżanie nowych list w liście. Dodawanie zawsze na koniec listy. Powtarzanie kodu pozwoli na kolejne dodawanie tego samego, bo to lista, a nie tupla, ale musi być wykonywany kod oddzielony od zmiennej, czyli po # %%. Za pomocą append możesz dodawać tylko pojedynczo, więc jeśli chciałbyś dodawać więcej niż jedno musisz robić jedno append pod drugim. Append może mieć tylko jeden argument. Możesz teoretycznie dodawać więcej wartości, ale one muszą wtedy być w obiekcie, czyli w liście, które się zagnieździ.

Np.

# -\*- coding: utf-8 -\*-

techs = ['sql', 'python', 'r', 'php', 'C++']

techs.append('java')

# %%

techs = ['sql', 'python', 'r', 'php', 'C++']

techs.append(['go', 'sql server'])

# Zagnieździ listę

**Metoda extend** → pozwala na dodawanie wielu wartości, ale tak, że będą ze wszystkimi, a nie w osobnej, zagnieżdżonej liście, jak w przypadku append. Również dodawanie na końcu. Tu możesz dodawać więcej niż jedną wartość. Po prostu ukryj wiele wartości na liście (nawias kwadratowy).

Np.

# %%

techs = ['sql', 'python', 'r', 'php', 'C++']

techs.extend(['java', 'go'])

# Doda jako pojedyncze

**Metoda extend, a łączenie list →** za pomocą metody extend można również łączyć ze sobą dwie listy.

Np.

# -\*- coding: utf-8 -\*-

list\_1 = [4, 5, 3, 3]

list\_2 = [9, 7]

list\_1.extend(list\_2)

print(list\_1)

**Metoda insert** → pozwala na wstawienie wartości w określonym miejscu na liście. Ma dwa argumenty. Pierwszy to miejsce gdzie dodać (od zera bo indeks), drugi to wartość. Wstawiana wartość wejdzie w miejsce poprzedniej, która była w określonym miejscu i przesunie ją “w lewo” - niezależnie czy cyfra wstawienia jest na plusie czy na minusie (od końca liczone).

Np.

# %%

techs = ['sql', 'python', 'r', 'php', 'C++']

techs.insert(0, 'go')

techs.insert(5, 'java')

# Dodaje w określonych miejscach

**Metoda pop** → zwraca ostatnią wartość listy (a pamiętaj, że te nie są losowo skonstruowane, tylko uporządkowane) do konsoli pomocniczej po czym wyrzuca z listy tą wartość. Dwa argumenty, ale pierwszy opcjonalny. Pierwszy to numer indeksu, który chcę wyrzucić. Można oczywiście korzystać wiele razy pod rząd, ale jak w poprzednich przypadkach musi być wykonywany kod oddzielony od zmiennej, czyli po # %%.

Np.

# %%

techs = ['sql', 'python', 'r', 'php', 'C++']

techs.pop()

# %%

techs = ['sql', 'python', 'r', 'php', 'C++']

techs.pop(4)

# Usuwa w określonym miejscu

**Metoda index** → zwraca gdzie na liście jest określona wartość.

Np.

# -\*- coding: utf-8 -\*-

techs = ['sql', 'python', 'r', 'php', 'C++']

print(techs.index('r'))

**Metoda count** → zwraca ile określonych wartości jest na liście.

Np.

# -\*- coding: utf-8 -\*-

techs = ['sql', 'python', 'r', 'php', 'C++', ‘r’, ‘r’]

# %%

techs.count('r')

**Metoda sort** → sortuje alfabetycznie w eksploratorze zmiennych listę. Nie posiada argumentów docelowo. Można dodać jednak (reverse=True) i dzięki temu odwrócić kolejność sortowania.

Np.

# -\*- coding: utf-8 -\*-

techs = ['sql', 'python', 'r', 'php', 'C++']

techs.sort()

# posortuje alfabetycznie

techs.sort(reverse=True)

# posortuje alfabetycznie, ale od z do a

**Metoda reverse** → działa analogicznie do wycinania za pomocą [::-1]. Nie ma argumentów. Metoda odwraca listę i robi to w eksploratorze zmiennych.

Np.

# -\*- coding: utf-8 -\*-

techs = ['sql', 'python', 'r', 'php', 'C++']

techs.reverse()

**Metoda za metodą** → możesz wykorzystywać metody od razu po sobie dla określonego obiektu. Dzieje się to po kropce: obiekt.metoda().metoda().

Np.

# -\*- coding: utf-8 -\*-

with open('products.txt', 'r') as file:

lines = file.readlines()

lines = lines[1:]

products = []

for line in lines:

part = line.strip().split(',')

products.append(part)

for product in products:

print(product)

**Lekcja\_15\_Słowniki**

**Słowniki** → są najpotężniejszą strukturą danych. Tworzysz pusty słownik za pomocą empty\_dict = dict() albo za pomocą a = {}. Słowniki przechowywane są w nawiasach sześciennych. Słownik jest strukturą nieuporządkowaną. Każdy słownik zawiera klucz i wartość i może być ich wiele, to znaczy wiele par. Każdy wartość ma klucz. Klucze nie powtarzają się i są unikalne. Kluczem nie musi być liczba może być wyraz. Dane wydobywa się nie za pomocą indeksu tylko za pomocą klucza.

Np.

# -\*- coding: utf-8 -\*-

empty\_dict = dict()

print (empty\_dict)

a = {}

print (a)

# %%

pol\_to\_eng = {'jeden':'one', 'dwa':'two', 'trzy':'three'}

print (pol\_to\_eng)

**Różnica między słownikiem, a zbiorem** → zbiór istotnie też jest w nawiasie sześciennym, ale słownik ma formę klucz:wartość i tak program rozpozna, że to słownik.

**Funkcja dict()** → tworzy pusty słownik.

**Funkcja len() zwraca** → zwraca dla słowników ilość obiektów czyli par klucz-wartość.

**Wstawianie kolejnych elementów do słownika** → za pomocą wzoru zmienna [nowy\_klucz] = nowa wartość. Czyli zmienna klucz w nawiasie kwadratowym równa się wartość. Pamiętaj, że przy wstawianiu nowych elementów nie używasz nawiasu sześciennego, ale kwadratowego.

Np.

# %%

pol\_to\_eng ['cztery'] = 'four'

**Obliczenia na wartości istniejącego klucza** → przy dodawaniu możesz to zrobić za pomocą += jak w przykładzie poniżej. Pozostałe obliczenia analogicznie.

Np.

color = {'blue': 5,

      'red': 6,

      'black': 7}

color['blue'] += 5

print(color['blue'])

# Zwróci 10 bo do blue dodało się 5.

**Metoda clear** → czyści słownik.

Np.

# %%

pol\_to\_eng.clear()

**Metoda copy** → pozwala na kopiowanie słowników. Użyj nowej zmiennej równa się stara zmienna.copy()

Np.

# %%

pol\_to\_eng\_copied = pol\_to\_eng.copy()

**Metoda keys** → pozwala na wyciągnięcie wszystkich kluczy ze słownika. Ona powstaje jako lista, ale dopiero przekonwertowanie jej za pomocą metody list pozwoli na stworzenie takiej zmiennej, która przechowuje tylko klucze. Czyli jeśli nie chcesz tylko wyprintować kluczy, a mieć je jako osobna zmienna zamknij wszystko w nawiasie z list. Pamiętaj, że to co powstanie będzie listą, a nie słownikiem.

Np.

# %%

lista\_kluczy = list(pol\_to\_eng.keys())

**Metoda values** → pozwala na wyciąganie wartości. Ona powstaje jako lista, ale dopiero przekonwertowanie jej za pomocą metody list pozwoli na stworzenie takiej zmiennej, która przechowuje tylko wartości. Czyli jak wyżej.

Np.

# %%

lista\_wartosci = list(pol\_to\_eng.values())

**Metoda items** → pozwala na wyciągnięcie wszystkiego i kluczy i wartości. Ona powstaje jako lista, ale dopiero przekonwertowanie jej za pomocą metody list pozwoli na stworzenie takiej zmiennej, która przechowuje jedno i drugie. Pamiętaj, że jest to lista, ale w środku są tuple. Para klucz-wartość jest tuplą. Oznacza to, że obiektami na liście możesz manipulować, np. usuwać je, ale samych obiektów nie zmienisz bo są tuplami.

Np.

car = {

  "brand": "Ford",

  "model": "Mustang",

  "year": 1964

}

x = car.items()

print(x)

**Wyciąganie ze słownika konkretnych wartości** → można za pomocą metody .get albo sposobem: zmienna[‘klucz’]. Jeśli klucz nie istnieje, pojawi się błąd. Zwróć uwagę, że 1 to nie w tym kontekście iteracja po określonym obiekcie, tylko dosłownie 1. Inny przykład pokazuje, że wyciągając bierzesz klucz.

Np.

# %%

new\_dict = {1:'one', 2:'two', 3:'three'}

# %%

print (new\_dict.get(1))

# %%

print (new\_dict[1])

# Dadzą taki sam efekt: wyprintują “one”

# %%

# Inny przykład:

medications = {

'hypertension\_medications': ['enalapril', 'hydrochlorothiazide'],

'diabetes\_medications': ['metformin'],

}

print(medications['hypertension\_medications'])

**Metoda get** → pozwala na wyciąganie wartości i jest lepsza niż powyższy sposób. Jest tak bo poza kluczem ma drugi, opcjonalny argument w postaci tego jaka wartość ma być oddana jeśli klucz nie istnieje.

Np.

# %%

pol\_to\_eng['dwa']

print(pol\_to\_eng.get('jeden'))

print(pol\_to\_eng.get('piec', 'NaN'))

**Metoda pop** → wyrywa element i usuwa go z listy. Muszą być podane argumenty w tej metodzie. Wybiera jeden klucz.

Np.

# %%

pol\_to\_eng.pop('jeden')

**Metoda popitem** → zwraca i usuwa dowolny element losowo. Nie ma argumentu.

Np.

# %%

pol\_to\_eng.popitem()

**Metoda update** → zmienia wartość przypisaną do danego klucza. Wzór: zmienna.update({klucz:nowa wartość}).

Np.

# %%

pol\_to\_eng.update({'jeden':1})

**Metoda update inna** → możesz też zmienić wartość klucza w inny sposób. Jako argumenty update podaj klucz/klucze bez zmian i za znakiem równości ich nowe wartości. Pamiętaj, że jeśli klucze są stringami to nie wpisujesz ich w cudzysłowie.

Np.

slownik = {'Imie': 'Leszek', 'Nazwisko': 'Dekiert'}

slownik.update(Imie = 'Aga', Nazwisko = 'Tomaszewska')

print(slownik)

**Del** → słowo kluczowe del może być użyte do usunięcia w miejscu klucza, który jest obecny w słowniku w Pythonie. Jedną wadą jest to, że zgłasza wyjątek, jeśli klucz nie zostanie znaleziony, a zatem należy obsłużyć nieistnienie klucza.

Np.

fruits = {'apple': 2, 'banana': 3, 'cherry': 5, 'orange': 1}

del fruits['orange']

print('After deleting:', fruits)

**Usuwanie elementu ze słownika i przypisanie go do zmiennej** → za pomocą metody pop. Rejestrujesz nową zmienną, która równa jest usuniętemu (pop) elementowi.

Np.

people = {'Alice': 25, 'Bob': 30, 'Charlie': 35, 'David': 40}

age\_of\_bob = people.pop('Bob')

print(age\_of\_bob)

print(people)

**Setdefault** → metoda setdefault() zwraca wartość elementu o podanym kluczu. Jeśli klucz nie istnieje, włoży klucz o określonej wartości.

Np.

people = {'Alice': 25, 'Bob': 30, 'Charlie': 35, 'David': 40}

age\_of\_emma = people.setdefault('Emma', 20)

print(age\_of\_emma)

print(people)

# %%

#ALBO

people = {'Alice': 25, 'Bob': 30, 'Charlie': 35, 'David': 40}

age\_of\_emma = people.setdefault('Alice', 20)

print(age\_of\_emma)

print(people)

**Słowniki umieszczone na liście i wyciąganie konkretnej wartości** → wystarczy wykorzystać nazwę klucza w nawiasie kwadratowym przy iteracji. Przykład poniżej.

Np.

cars = [

{'model': 'Tesla', 'mileage': 15000, 'battery\_level': 100},

{'model': 'Nissan', 'mileage': 30000, 'battery\_level': 75},

{'model': 'BMW', 'mileage': 5000, 'battery\_level': 100},

{'model': 'Ford', 'mileage': 20000, 'battery\_level': 50}

]

for car in cars:

if car['battery\_level'] == 100:

     print('The first car with a full charge is:', car['model'])

     break

**Słowniki umieszczone na liście i wyciąganie wielu konkretnych wartości** → Skorzystaj z iteracji. Przykład poniżej.

Np.

cars = [

{'model': 'Tesla', 'mileage': 15000, 'battery\_level': 100},

{'model': 'Nissan', 'mileage': 30000, 'battery\_level': 75},

{'model': 'BMW', 'mileage': 5000, 'battery\_level': 100},

{'model': 'Ford', 'mileage': 20000, 'battery\_level': 50}

]

print([d['battery\_level'] for d in cars])

**Iterowanie po kluczach i wartościach jednocześnie** → jeśli potrzebujesz iterować po obu elementach słownika, musisz zaznaczyć to w instrukcji for poprzez nadanie po przecinku dwóch obiektów-iteratorów. Pierwszym będzie klucz, drugim wartość. Ponadto w iteracji musisz skorzystać z metody items, która wyciągnie i klucze i wartości.

Np.

proportions = {

'flour': 500,

'salt': 4,

'sugar': 200,

'butter': 150

}

recepie = 3000

ingredients = {}

counter = 0

while counter < recepie:

for ingredient, amount in proportions.items():

     if ingredient not in ingredients:

         ingredients[ingredient] = amount

     else:

         ingredients[ingredient] += amount

     counter += amount

print(f'To prepare {recepie} g of dough, you need:')

for i, a in ingredients.items():

print(f'{i} - {a} g'.capitalize())

**SEKCJA 4. KONTROLA PRZEPŁYWU PROGRAMU**

**Lekcja\_16\_Wartości logiczne**

**Bool** → Typ logiczny, typ boolowski – uporządkowany zbiór wartości logicznych, składający się z dokładnie dwóch elementów: prawda i fałsz. Bool = boolean.

**Koniunkcja zdarzeń** → część wspólna, jest prawdziwa jeśli oba zdarzenia są prawdziwe. Symbol koniunkcji zdarzeń to “and”. Koniunkcja to dwa zdania połączone spójnikiem logicznym i.

Np.

True and True - samo, bez poniższych linijek kodu odda wartość True

True and False - wystarczy do oddania False dla całej koniunkcji (wszystkich czterech jej elementów)

False and True - j/w

False and False - j/w

**Alternatywa** → symbol to “or”. Jest spełniona gdy chociaż jeden warunek jest prawdziwy. Alternatywa to dwa zdania połączone spójnikiem logicznym lub. W alternatywie wystarczy, że jeden element jest prawdziwy, reszta może być fałszywa

Np.

True or True

True or False

False or True

False or False - wszystko zwróci True

**Negacja** → zwraca wartość przeciwną. Negacja to zaprzeczenie zdania, czyli: nieprawda, że (zdanie). Symbol to “not”.

Np.

True - zwróci True

not True - zwróci True

False - zwróci False

not False - zwróci True

**Funkcja bool** → zwraca wartości logiczne obiektów. Każdy tekst zwróci True, ale pusty string, sam cudzysłów zwraca False. W przypadku liczb wartość 0 zwróci False, a każda inna liczba daje True. Pusty słownik zwróci False - zbiór, lista, tupla tak samo. Ale jeśli lista będzie składała się z pustego ciągu znaków (sam cudzysłów) to będzie True.

Np.

# %%

bool('') - False

bool(1) - True

bool([]) - False

bool(['']) - True

bool(0) - False

bool(0.1) - True

bool(0.0) - False

**Lekcja\_17\_Instrukcje warunkowe\_I**

**Operator porównania** → (==), to jest porównanie i zwróci wartość logiczną. To nie jest operator przypisania czyli jedno równa się (=). Pamiętaj, to częsty błąd.

Np.

# %%

b = 1200

b == 1200

# Zwraca True, a wszystko inne False

**Operator przeciwny do porównania** → czy wartość x jest różna od wartości y. Operator to != (wykrzyknik i równa się). Wykrzyknik jest tu negacją.

Np.

# %%

c = 1000

c != 1000

# Zwróci False, bo to negacja. Wszystko inne zwróci True.

**Instrukcja if** → konstrukcja poniżej. If oznacza tu jeżeli.

Konstrukcja:

if [warunek]:

4 spacje (które Spyder stworzy domyślnie) [instrukcje]

Np.

# %%

if 8 < 10:

    print ('Tak')

# Zwróci “Tak”

**Samodzielne if** → zwróć uwagę, że instrukcji if nie jest potrzebne else/elif żeby działać. Niektóre sytuacje nie wymagają stosowania else/elif.

Np.

products = [

('T-shirt', 'Clothing', 50.00),

('Pants', 'Clothing', 100.00),

('Shoes', 'Footwear', 150.00)

]

filtr = 'Clothing'

for p in products:

if filtr == p[1]:

     print(p[0], p[2])

**Zachodzenie warunku** → jeżeli zachodzi warunek to wykonuj instrukcję. W powyższym przypadku jeżeli nie zachodzi warunek to po prostu nic się nie wyprintuje.

**Wcięcie if** → te cztery spacje są konieczne, żeby instrukcja zadziałała. W innych językach nie ma takich wcięć, korzysta się np. z nawiasów kwadratowych.

**Else** → warunek drugi, dodatkowy. Uruchamia się jeśli spełniony jest właśnie on, a nie to co bezpośrednio za if. Też 4 spacje od nowej linii za dwukropkiem.

Np.

# %%

a = 5

if a > 10:

print ('a > 10')

else:

print ('a <= 10')

**Elif** → Kilka testów w jednej instrukcji. Warunek obsługujący nie wszystkie pozostałe możliwości jak Else, tylko jakiś warunek konkretny. Bezpośrednio za nim pisze się ten konkretny warunek, jak za if. Za nim w nowej linii jak zawsze 4 spacje.

Np.

# %%

age = 17

if age == 18:

    print ('Masz 18 lat i dostęp')

elif age < 18:

    print ('Nie masz dostępu')

elif age > 18:

    print ('Dostęp przyznany')

# W tym miejscu możesz dać też else na koniec i efekt będzie taki sam, a to dlatego, że została z logicznego punktu widzenia tylko jedna opcja zdarzenia.

**Funkcja warunkowa i input** → input zwraca wartość jako string. Dlatego musisz konwertować funkcją int.

# %%

wiek = int(input('Podaj swój wiek: '))

if wiek == 18:

    print ('Masz 18 lat i dostęp!')

elif wiek < 18:

    print ('Nie masz dostępu!')

elif wiek > 18:

    print ('Masz dostęp!')

**Lekcja\_18\_Instrukcje warunkowe\_II**

**Przygotowanie prostego programu z instrukcją i cyframi** → za pomocą string.

Np.

# %%

print ('System sie uruchamia...')

print ("""Witaj uzytkowniku. Podaj haslo.

Haslo sklada sie z dwoch znakow.

Uzyj znakow miedzy 0, a 2""")

pin = input('Podaj PIN: ')

if pin == '21':

    print ('Dostep do sysytemu przyznany')

elif pin == '20':

    print ('Byles blisko!')

else:

    print ('Nieprawidlowe haslo')

**Wartości bool (prawda/fałsz), a instrukcje warunkowe** → Wartość logiczna pustego ciągu jest fałszywa. Instrukcje warunkowe mogą działać na zasadzie True/False. Jeśli z instrukcji wychodzi True to po zwykłym if sprawdzającym daną zmienną pojawi się określona instrukcja, a jeśli False to instrukcja pojawi się za else. W teście prawda/fałsz nie trzeba niczego porównywać, wystarczy if wartość. If docelowo będzie rozumiało True jako wartość docelowa, chyba, że korzystasz z if not (o tym dalej poniżej).

Np.

# %%

string = ''

print (bool(string))

# False

if string:

    print ('Niepusty ciag znakow')

else:

    print ('Pusty ciag znakow')

# %%

number = 0.1

print (bool(number))

# True

if number:

    print ('Liczba niezerowa')

else:

    print ('Zero')

**Definiowanie flagi, która coś w programie robi**→ poniższy przykład pokazuje wykorzystanie wartości logicznych.

# %%

default\_flag = True

if default\_flag:

    print ('Doszlo do defaultu.')

else:

    print ('Nie doszlo.')

# Odda True

**If not** → instrukcja, która daje odwrotność szukanej. W przypadku wartości logicznych, ponieważ żeby If szukało True nie trzeba podawać operatora porównania, to przy zastosowaniu If not będzie szukać False.

Np.

# %%

default\_flag\_II = False

if not default\_flag\_II:

    print ('Nie doszlo do defaultu.')

else:

    print ('Doszlo do defaultu.')

# Odda False

**Lekcja\_19\_Instrukcje warunkowe\_III**

**Wiele warunków w instrukcji** → zwykle będzie tak, że dwa warunki = dwie zmienne, ale nie koniecznie, bo warunek to mogą być jakieś wewnętrzne obliczenia czy porównania, etc. Wzór: if warunek\_1 and warunek\_2 and warunek\_3…. Koniunkcja będzie prawdziwa jeśli wartości będą spełnione i prawdziwe. Poniżej przykład z całej lekcji.

**Flaga** → Nie ma definicji technicznej. Flaga to po prostu zmienna, której wartość (zazwyczaj Boolean) wskazuje na konkretną sytuację. Flaga w Pythonie działa jako sygnał dla programu, aby określić, czy program jako całość, czy konkretna część programu powinna zostać uruchomiona. Innymi słowy, możesz ustawić flagę na True, a program będzie działał nieprzerwanie, dopóki jakikolwiek typ zdarzenia nie spowoduje, że będzie False. Wtedy program, pętla lub cokolwiek, dla którego używasz flagi, zatrzyma się.

**And** → operator logiczny “i”. Zwraca True, jeśli oba/trzy… stwierdzenia są prawdziwe.

**Funkcja abs** → zwraca wartość bezwzględną (absolute). Wykorzystywana do zmiany wartości ujemnej na dodatnią.

Np.

# %%

print (abs(-10))

# Da wynik 10

# %%

saldo = 100000

klient\_zweryfikowany = int(input('Podaj PIN: '))

correct\_PIN = 1234

amount = abs(int(input('Podaj kwotę którą chcesz wypłacić: ')))

if saldo > 0 and klient\_zweryfikowany == correct\_PIN and saldo >= amount:

    print ('Pieniądze wypłacane. '

           'Na koncie zostało {} złotych.'.format(saldo - amount))

elif saldo > 0 and klient\_zweryfikowany == correct\_PIN and saldo < amount:

    print ('Brak wystarczających srodków do wypłaty.')

elif klient\_zweryfikowany > 1234:

    print ('PIN nieprawidłowy.')

elif klient\_zweryfikowany < 1234:

    print ('PIN nieprawidłowy.')

elif saldo < 0 and klient\_zweryfikowany == correct\_PIN and amount > 0:

    print ('Brak srodków na koncie.')

else:

    print ('Błąd systemu. Proszę skonsultować się z kimstam.')

**Lekcja\_19\_Instrukcje warunkowe\_IV**

**Operator in w instrukcjach** → skorzystaj z operatora in żeby sprawdzić czy w obiekcie znajdują się określone wartości.

Np.

# %%

tech = 'python'

if 'a' in tech:

print ('True')

else:

print ('False')

**Przypisanie nowej zmiennej za pomocą instrukcji** → w wyniku if, elif, else wprowadź nową zmienną i przypisane do niej wartości.

Np.

# %%

tech = input('Type You\'r tech and I\'ll tell You what else to try: ')

if tech == 'python':

other\_tech = 'sas'

elif tech == 'sas':

other\_tech = 'python'

else:

print ('tech not recognised')

**Pisanie instrukcji przypisującej wartość w jednej linii** → da się to zrobić w jednej linii. Wzór: prawda if warunek else inne

Np.

# %%

tech = input('Choose You\'r technology: ')

opinion = 'Good choice' if tech == 'python' else 'Try something else'

**Lekcja\_20\_Pętle for**

**Konstrukcja pętli** → for element\_iterator (kolejna wartość dla której będziemy iterować, czyli zmienna, której zostanie przypisana nowa wartość w wyniku iteracji) in obiekt\_iterowalny (obiekt, którego elementy będą iterowane przy każdej pętli): (na końcu instrukcji for jest dwukropek) np. print (wynik) LUB kolejne elementy pętli.

**Logika pętli →** pętla działa tak, że są iteracje. Dla prostej instrukcji for iteracje trwają od początku do końca obiektu iterowanego. Wzór mówi, że dla (for) wartości (value) przypisana jest wartość w (in) obiekcie iterowanym.

Np.

# -\*- coding: utf-8 -\*-

name = 'python'

for character in name:

print (character)

# Będzie iterować aż do końca i ostatni fragment obiektu, tutaj “n” zostanie przypisany wartości elementy iterującego, tutaj “character”

**Teksty jako obiekty iterowalne** → teksty w python są obiektami iterowalnymi i zwracają elementy pojedyncze znaki ciągu wyrazów. Słowo ‘python’ w pierwszej iteracji zwróci “p”, w drugiej “y”…

**Iterowanie po indeksach i bardziej skomplikowana pętla** → pętla jest tak skonstruowana, że index zaczyna się od 0, a iterator naturalnie będzie wzrastać z każdą iteracją o literkę. Żeby również index wzrastał o 1 to na końcu każdej iteracji znajdującej się w instrukcji for na końcu dodaję polecenie dodające 1 - to polecenie kończące pętlę.

Np.

Pierwsza metoda:

# -\*- coding: utf-8 -\*-

name = 'python'

index = 0

for character in name:

print (index, character)

index = index + 1

**Funkcja range** → zwraca obiekt iterowalny. Zwraca sekwencję liczb, domyślnie zaczynając od 0 i zwiększa się o 1 (domyślnie) i zatrzymuje się przed określoną liczbą. Wzór: range (start, stop, krok), z czego start i krok są opcjonalne. Start określi pozycję startową - jeśli nie są podane 3 warunki to opcjonalnie jest to zero. Krok określa inkrementację, co ile przeskakują liczby. Przy braku warunku jest to domyślnie 1. Żeby zobaczyć w konsoli wynik takiego zakresu umieść go w liście.

Np.

# %%

list (range (1, 10, 2))

# Zwróci 1, 3, 5, 7, 9

# %%

list (range (10))

# Zwróci 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 gdyż pamiętaj, że indeksowanie zaczyna się od zera

# %%

for index in range (10):

print (index)

# Wartości index zostanie przypisana cyfra 9

**Iterowanie po długości tekstu** → określ jakie iteratory cię interesują: obiekt - len - range list - znasz iteratory. Zagnieżdżanie funkcji: range(len(wartość)). Pamiętaj, że len zwraca ile obiektów jest w zbiorze. To działa ponieważ range tworzy listę, a lista jest iterowalnym obiektem i z każdą iteracją kolejna liczba będzie wybierana. Np. range(len(zmienna\_10), gdzie zmienna\_10 = aaaaaaaaaa, zamieni najpierw na 10, a range umieści 10 w liście.

Np.

# %%

name = 'Python'

for index in range(len(name)):

print (index, name[index])

# powyższy kod wyciąga w każdej iteracji kolejne cyfry zaczynając od 0 i kolejne litery zaczynając od “P”. Zwróć uwagę, że to index podlega jako pierwszy iteracji, a iteracja name działa dlatego, że skorzystałeś z funkcjonalności wyciągania określonej części obiektu z listy, co dzieje się za pomocą nawiasu kwadratowego.

**Customizacja pętli** → w print dodaj treść, która Cię interesuje w cudzysłów.

Np.

# %%

name = 'Python'

for index in range(len(name)):

print ('Numer:', index, 'Name:', name[index])

**Enumerate, rekomendowany sposób iteracji element, indeks** → funkcja enumerate zwraca numer indeksu i pierwszy element jako tuplę. To szybki sposób na iterację. Korzystając z możliwości rozpakowania tupli można też z automatu te dwie wartości dzielić do osobnych nowych zmiennych. Funkcja enumerate jest rekomendowana jeśli chodzi o pętle for i przechodzenie przez obiekty iterowalne. Enumerate działa tak, że podasz dwie zmienne przed for to pierwszym elementem będzie zautomatyzowany numer indeksu, a drugim kolejny element iterowanego obiektu. Zatem w przykładzie for a, b in enumerate to “a” będzie numerem indeksu. Jeśli natomiast podasz jedną zmienną dla przykładu for a in enumerate, to powstanie tupla składająca się z numeru indeksu i elementu obiektu.

Np.

# %%

imie = 'Leszek'

for i in enumerate (imie):

print (i)

# 5 i k znajdą się w zmiennej w formie tupli, bo były ostatnie

# %%

imie = 'Leszek'

for index, character in enumerate (imie):

print ('Numer indeksu: ', index, 'Litera: ', character)

# 5 i k znajdą się w osobnych zmiennych int i str, bo były ostatnie

**Iterowanie listy za pomocą enumerate** → lista jest obiektem iterowalnym.

Np.

# %%

lista = [4, 5, 6, 7, 8, 6, 8]

for i, v in enumerate (lista):

print (i, v)

**Range, zapis, korzystanie** → w funkcji range występuje zapis lewostronnie domknięty, czyli w range (10, 20) 10 wpada, 20 nie, tylko 19. Odwracanie kolejności (10, -1, -1) - żeby liczył od 10 do 0, bo -1 nie wpada. Pamiętaj, że jeśli podajesz w range tylko start, to będzie wyciągać od 0 jak idą indeksy. Jeśli jednak podasz start i stop to będzie wyciągać dosłownie od tej do tej cyfry jakiej chcesz.

Np.

# %%

for a in range (10,-1,-1):

print (a)

# Od 10 do 0

**Wycinanie do określonego indeksu** → jeśli chcesz żeby pętla zatrzymywała się w określonym miejscu lub szła w określonym kierunku możesz korzystać z kombinacji wycinania.

Np.

for char in string[:1]:

# %%

techs = 'java'

for a in techs[:1]:

print(a)

# Wytnie “j” i zmiennej przypisze “j”

**Krok iteracji** → jak na przykładzie poniżej.

Np.

# %%

techs = 'java'

for a in techs[::2]:

print(a)

# Wytnie “j” i “v” i zmiennej przypisze “v”

**Kroki od końca** → jak na przykładzie poniżej.

Np.

# %%

techs = 'java'

for a in techs[::-1]:

print(a)

# Wytnie “java” od końca

**For i instrukcje warunkowe** → Łączenie for i if. Za if tworzysz instrukcję, która podaje co iterować. Podaję np. warunek, że jeżeli iterowany znak nie jest równy czemuś, np. “#” to go nie printuję. Tak jak w poniższym przykładzie. Pamiętaj jednak, że printowanie to co innego niż samo działanie kodu. Program nadal przejedzie i uwzględni “#”, po prostu go nie wyprintuje.

Np.

# %%

hashtags = '#sport#gym#fit'

for a in hashtags:

if a not in '#':

     print (a)

# Wyprintuje “sportgymfit”

# %%

hashtags = '#sport#gym#fit#'

for a in hashtags:

if a not in '#':

     print (a)

# Dodałem na końcu zmiennej #, żeby pokazać, że istotnie program nadal nie wyprintuje “#”, ale jako zmienna “a” przypisany zostanie “#”.

**Zip →** funkcja zip() zwraca obiekt zip, który jest iteratorem tupli, w których pierwszy element w każdym przekazanym iteratorze jest sparowany razem, a następnie drugi element w każdym przekazanym iteratorze są sparowane razem itd. Jeśli przekazane iteratory mają różne długości, iterator z najmniejszą liczbą elementów decyduje o długości nowego iteratora. Zmienna powstała na skutek wykorzystania funkcji zip może zostać rozpakowana za pomocą funkcji list.

Np.

# %%

a = ("John", "Charles", "Mike")

b = ("Jenny", "Christy", "Monica", "Vicky")

x = zip(a, b)

print (list(x))

# Wynik: [('John', 'Jenny'), ('Charles', 'Christy'), ('Mike', 'Monica')]. “Vicky” tu nie ma bo nie było z czym jej sparować.

**Iterowanie po elementach zip** → funkcja rozpakowuje iterowane obiekty w tuple lub pojednycze obiekty, zależnie od potrzeby. Jeśli wartości mają różną długość, to funkcja zip obcina wszystkie zmienne do długości najkrótszej zmiennej. Wzór: for nowa zmienna in zip (‘wartość’, ‘wartość’).

Np.

# %%

techs = 'Python'

char = 'abcde'

for a in zip (techs, char):

print (a)

# zwraca tuple zaczynając od ('P', 'a')

# %%

techs = 'Python'

char = 'abcde'

for a, b in zip (techs, char):

print (a, b)

# Zwraca dwa stringi i przypisze a i b ich ostatnie elementy

# %%

techs = 'Python'

char = 'abcde'

number = 1, 2, 3, 4, 5

for a, b, c in zip (techs, char, number):

print (a, b, c)

# Działa też z liczbami

**Wyciąganie poszczególnych słów za pomocą for if** → kod wyciągający poszczególne słowa na podstawie separatora, wraz z opisem. Przecinek na końcu. Wytłumaczenie przykładu: potrzebna ci pusta zmienna gdyż będzie ona sukcesywnie wypełniana z każdą iteracją, a jak natrafi na separator (w tym przypadku separator słów), to wróci do pustej postaci by znów wypełnić się, ale już następną zmienną. Jeśli pętla nie natrafi na separator to do pustej zmiennej doda pierwszy element, z kolejną iteracją drugi, itd. aż nie natrafi na separator. Ostatecznie natrafienie na separator oznacza wyprintowanie zebranego ciągu obiektów.

Np.

# %%

imiona = 'Leszek,Aga,Frania,'

wynik = ''

for litera in imiona:

if litera not in ',':

     wynik = wynik + litera

else:

     print (wynik)

     wynik = ''

**Lekcja\_21\_Instrukcja break**

**Instrukcja break** → instrukcja break zatrzymuje działanie pętli w określonym momencie. Pamiętaj, że to co znajduje się pod break i w jakim miejscu ma znaczenie. Jeśli chcesz, żeby kolejna linia kodu pod break zadziałała kiedy uruchomi się break (gdyż spełniony zostanie warunek break), to ustaw linię kodu po enterze i od skrajnej, prawej linii. Wzór: for zmienna in wartość: if warunek: break.

**Int** →  formuła zamieniająca str na int. Działanie: zmienna = int(zmienna). Poniżej przykład dla Int i Break.

# %%

for n in '0123456789':

    n = int(n)

    print (n)

    if n == 5:

        break

print ('Koniec')

# Zwraca:

0

1

2

3

4

5

Koniec

**Komentarz do przykładu** → Pamiętaj, że w przykładzie powyżej ostatnie print nie może być na równi z linią break. Powinno być równe z for, inaczej kod nie zadziała prawidłowo. Rzeczy takie jak “Koniec” warto dodawać żeby kontrolować działanie kodu.

**Instrukcje poza pętlami** → oddziel je enterem i pisz od skrajnej prawej strony.

Np.

# %%

name = 'Leszek Dekiert'

for char in name:

    if char == ' ':

        break

    print (char)

print ('Koniec!')

**For, break i else** → umieść blok instrukcji wykonywanej jeśli nie dojdzie do spełnienia warunku break. Break na tej samej pozycji co print w tym przypadku. Else pozwala na określenie dodatkowego warunku dla innego wyniku pętli jeśli nie dojdzie do break.

# %%

email = 'jankowalski@gmail.com'

for at in email:

if at == '@':

   print ('Adres email jest poprawny.')

   break

else:

print ('Adres email nie jest poprawny.')

print ('Koniec pętli.')

**Bardziej skomplikowany przykład z dodatkowym warunkiem** → Przykład.

Np.

# %%

password = 'jnhvsoics!vd'

for a in password:

    if a == '!' and len(password) > 10:

        print ('Hasło poprawne')

        break

else:

    print ('Hasło niepoprawne')

**Różnica między pokazywaniem procesu poszukiwań, a brakiem pokazania** → chodzi o to, że możesz printując pokazać, że program szuka czegoś zanim uruchomi się break. Masz dwie opcje (na przykładzie):

Np.

# %%

ciag = 'qwesda2323r4wefaw3s4q34eqwa'

for password in ciag:

if password == 'f':

     print ('Found letter f!')

     break

else:

     print ('Couldn\'t find letter f...')

print ('End.')

# Powyższe pokaże ciąg “Couldn\'t find letter f…” przed break. Jeśli chcesz tylko zobaczyć break i wynik, to zakomentuj “print ('Couldn\'t find letter f...')” i else.

**Lekcja\_22\_Instrukcja continue**

**Instrukcja continue** → pozwala pominąć jedną wybraną iterację i przechodzi do następnego elementu iterowalnego w pętli. Ona pozwala pomijać interesujący cię iterator, który chcesz ominąć. Pamiętaj: continue służy do pomijania.

Np.

# %%

numbers = range(10)

for a in numbers:

if a == 6:

     continue

print (a)

# pokaże w konsoli wszystkie możliwe cyfry, nie zatrzyma się na 6, gdyż zadziałało continue, które nakazało zignorować to, co jest pod continue. Czyli pokaże 0,1,2,3,4,5,7,8,9.

**Logika if i continue** → pamiętaj, że w przypadku połączenia if i continue, tylko jeśli (if) zostanie spełniony określony warunek to continue zadziała - a zatem ten obiekt zostanie pominięty. Jeśli warunek nie zostanie spełniony, to continue nie zadziała, a zatem uruchomione zostanie to co jest pod continue. Poniższy przykład ma pokazywać tylko liczby nieparzyste.

Np.

# %%

numbers = range(20)

for a in numbers:

if a % 2 == 0:

     continue

print (a)

print ('End.')

**Continue i litery** → continue można wykorzystywać oczywiście też do pomijania liter.

Np.

# %%

tech = 'Python Course'

for char in tech:

if char == ' ':

     continue

print (char)

print ('End.')

**Continue, if i co jeśli nie-continue** → w zależności od położenia kodu, if zrobi wszystko do continue, a to co pod continue stanie się jako takie else. A to co pod tym jako takim else to stanie się po zakończeniu pętli. Chodzi o to, że w przykładzie poniżej to co za znakiem hashtag zaczyna się od nowa i dlatego instrukcja if  przy # usuwa wszystko co zebrało się do tej pory i to printuje, a potem kasuje dla następnego ciągu liter.

Np.

# %%

hashtags = '#summer#beach#free#sand'

result = ''

for char in hashtags:

    if char == '#':

        print (result)

        result = ''

        continue

    result = result + char

print (result)

print ('End.')

# Jako ostatnie wyprintowany będzie “Sand”, bo trzeba to było zrobić ręcznie. Kod printuje sam to co za #. No i na końcu wyprontowany zostanie “End.”

**Lekcja\_23\_Pętle for - jeszcze więcej przykładów**

**Oczyszczanie danych, oddzielanie kodu ze zmiennymi od reszty** → na zasadzie przykładu wymiany znaków w ciągu liczb.

Np.

raw\_data = '345!23!3234!43434'

modified\_data = ''

for number in raw\_data:

    if number != '!':

        modified\_data += number

    else:

        modified\_data += ','

print (modified\_data)

print ('End.')

**Iterowanie po obiektach, które są podzielone znakiem** → najpierw dodawaj elementy obiektów do pustej zmiennej, po czym zaczynaj operację od nowa gdy pojawi się rozgranicznik.

Np.

???

**Wartości rozdzielane przecinkami** → często spotykany sposób zapisu danych w CSV, czyli Comma Seperated Values. CSV: format przechowywania danych w plikach tekstowych i odpowiadający mu typ MIME text/csv.

**Sumowanie elementów, jakie dotychczas się pojawiły** → Przykład: suma pierwszych 9 liczb.

Np.

# %%

ciag = range(10)

suma = 0

for a in ciag:

    suma += a

print (suma)

**Bardziej rozbudowany przykład z bankiem. Łączenie for i format.**

Np.

# %%

saldo = 450

wyplata = range (10,60,10)

print ('Saldo początkowe = {}.'.format(saldo))

for a in wyplata:

    print ('Wyplacona kwota = {}.'.format(a))

    saldo -= a

    print ('Saldo = {}.'.format(saldo))

print ('Stan konta po pol roku = {}.'.format(saldo))

**Próbowanie logowania się za pomocą kodu pin cyframi** → Poniżej mój pomysł.

Np.

# %%

print ('Witaj w systemie logowania!')

print ('\*' \* 30)

poprawny\_pin = '1234'

pin = input('Podaj PIN: ')

pusty\_pin = ''

for num in pin:

    if num.isdigit() == True:

        pusty\_pin += num

    else:

        continue

if pusty\_pin == poprawny\_pin:

    print('PIN poprawny!')

else:

    print('PIN niepoprawny!')

**Przesuwanie całego wybranego kodu o 4 spacje** → wybierz wszystko i tab.

Np.

# %%

print ('Witaj w systemie logowania!')

print ('\*' \* 30)

nick = input ('Podaj swój nick: ')

pin = input ('Podaj kod PIN {}: '.format(nick))

empty\_pin = ''

if len(pin) == 4:

    for char in pin:

        if char not in '0123456789':

            print ('Kod PIN niepoprawny! Korzystaj tylko z liczb!')

            break

        else:

            empty\_pin += char

    if empty\_pin == '1234':

        print ('PIN poprawny!')

    else:

        print ('PIN niepoprawny!')

else:

    print ('PIN niepoprawny - podałes za mało lub za duzo znaków!')

**Lekcja\_24\_Pętle while**

**Pętle while** → w pętlach while to użytkownik podaje kryterium zatrzymania. Podaję warunek, kiedy pętla się skończy. Wzór: while warunek: dalsze zapętlone działania.

**Program, który nigdy się nie kończy** → zdarza się, że błędnie tworzy się programy, które nie mają końca, np. ciągle powtarzają wynik.

Np.

a = 0

while a < 10:

    print (a)

**Zatrzymanie nie kończącego się programu** → Stop the current command - czerwony kwadracik.

**Zatrzymanie kodu** → Dodana inkrementacja w celu zatrzymania kodu.

Np.

a = 0

while a < 10:

    print (a)

    a += 1

# %%

**While i True** --> kod sprawdza prawdziwość warunku zerojedynkowego True/False.

Np.

a = 0

while True:

    print (a)

    if a > 10:

        break

    else:

        a += 1

**Pętla while i pobieranie danych od użytkownika** → na przykładzie.

Np.

# %%

while True:

    name = input ('Insert Your\'e name here: ')

    if len(name) > 3 and name.isalpha():

        break

print ('Hi {}'.format(name))

# Pętla będzie powtarzać się dopóki nie podasz poprawnego imienia

**While True →** While True używasz zwykle gdy nie chcesz żeby pętla się w ogóle zatrzymała.

Np.

def fibonacci():

     a, b = 0, 1

     while True:

         yield a

         a, b = b, a + b

# %%

fib = fibonacci()

# %%

next(fib)

**Przykład z wyciąganiem liczb parzystych/nieparzystych** → przykład poniżej.

Np.

# %%

n = 0

while n < 20:

    n = n + 1

    if n % 2 != 0:

        continue

    print (n)

**Poszukiwanie obiektu na liście** → przykład poniżej.

Np.

# %%

# Poszukiwanie czy obiekt występuje

lista\_do\_przeszukania = [12, 53, 34, 56, 333]

flaga = False

wartosc = int(input('Podaj szukana liczbe: '))

idx = 0

while idx < len(lista\_do\_przeszukania):

    if lista\_do\_przeszukania[idx] == wartosc:

        flaga = True

        break

    idx += 1

if flaga:

    print ('Znaleziono poszukiwana wartosc {}.'.format(wartosc))

else:

    print ('Nie znaleziono wartosci {}.'.format(wartosc))

**Program sprawdzający występowanie i dodający nowy element** → na przykładzie.

Np.

# %%

lista\_do\_przeszukania = [12, 53, 34, 56, 333]

flaga = False

wartosc = int(input('Podaj liczbe: '))

idx = 0

while idx < len(lista\_do\_przeszukania):

    if lista\_do\_przeszukania[idx] == wartosc:

        flaga = True

        break

    idx += 1

if flaga:

    print ('Wartosc {} juz znajduje sie na liscie.'.format(wartosc))

else:

    lista\_do\_przeszukania.append(wartosc)

    print ('Wartosc {} zostala dodana do listy.'.format(wartosc))

**Lekcja\_25\_Pętle while - jeszcze więcej przykładów**

**Przykład** → zmienne stałe pisane dużymi literami. Kod prosi o kolejne próby.

# -\*- coding: utf-8 -\*-

#

KOD\_PIN = 0000

pin = input('Podaj kod PIN: ')

while int(pin) != KOD\_PIN:

    pin = input('PIN nieporawny. Podaj kod ponownie: ')

print ('Kod PIN poprawny!')

# %%

**Przykład** → kod dający 3 próby na wpisanie PIN

KOD\_PIN = '0000'

pin = ''

counter = 0

while pin != KOD\_PIN and counter < 3:

    pin = input('Podaj kod PIN: ')

    if pin == KOD\_PIN:

        print ('Zalogowany!')

        break

    counter += 1

else:

    print ('Zbyt duzo prob logowania!')

**Or i and w instrukcjach warunkowych while** → zwróć uwagę, że określone warunki, np. iteracje będą zapętlone tak długo jak warunek logiczny zwraca True. Jeśli postawisz dwa lub więcej warunków, to przy or pętla zatrzyma się jeśli wszystkie zwrócą False, a and gdy tylko jeden zacznie zwracać False. Or to albo czyli wystarczy, że co najmniej jeden będzie zwracać True, a and to i czyli wszystkie muszą zwracać True.

#Np.

hour = 8

solar\_power = 50

battery\_capacity = 500

battery\_level = 0

while hour < 15 or battery\_level < battery\_capacity:

battery\_level += solar\_power

hour += 1

print(f'The solar battery charge level is: {battery\_level} Watt-hours')

**SEKCJA 5. INPUT/OUTPUT - czytanie i zapisywanie plików w Pythonie**

**Lekcja\_26\_Wczytywnie plików**

**Stworzenie pliku** → Plik tekstowy - stwórz w Spyder. Files, dodaj, zakończ nazywanie z .txt. Zapisz i zamknij. Następnie skorzystaj z funkcji open.

**Open** → funkcja przyjmuje nazwę pliku jako 1 parametr, 2 parametr - atrybut określający czy chcesz plik odczytać, zapisać, itd. “r” oznacza przeczytać. Jeśli chcesz żeby zawartość została wyprintowana musisz dokonać iteracji po liniach tekstu. Funkcja będzie działać sama, tak jak poniżej jeśli plik, który odczytuję znajduje się w folderze wyjściowym w Spyder.

Np.

file = open ('simple.txt', 'r')

for line in file:

    print (line)

**Read** → metoda read() zwraca określoną liczbę bajtów z pliku. Wartość domyślna to -1, co oznacza cały plik. Używana z open i print. Przewaga nad zwykłą iteracją to nie trzeba zwracać uwagi na usuwanie pustych linii.

Np.

file\_2 = open('C:/Users/Leszek/data.txt', 'r')

print(file\_2.read())

**Wczytywanie plików z określonego źródła** → jako pierwszy argument podaj całą ścieżkę dostępu (na końcu jest nazwa pliku). Użyj funkcji open i podaj, że chodzi o “r” czyli read. Uwaga: tylko dlatego, że w ścieżce pliku w Windows korzysta się z backslash nie oznacza, że w Spyder piszesz też backslash - tu piszesz normalny apostrof.

Np.

f = open('C:/Users/Leszek/Documents/Szkolenia/Pliki tekstowe/patients.txt', 'r')

print(f.read())

**Niwelowanie przerw w odczycie** → wykorzystywane przy iterowaniu. Program będzie printować przerwy między liniami gdyż w pliku są entery - tak jak na przykładzie w pliku tekstowym jest: pierwsza linia[ENTER]druga linia… Niwelowanie przerw: za pomocą parametru end w print i podać go jako pusty.

# %%

file = open ('simple.txt', 'r')

for line in file:

    print (line, end='')

**Parametr end w print** → parametr określający w którym miejscu ma skończyć się odczyt. Jeśli w poniższym przykładzie podałem nic to znaczy, że kończy się przed enterem. Domyślnie funkcja print() w Pythonie kończy się znakiem nowej linii.

# %%

file = open ('simple.txt', 'r')

for line in file:

    print (line, end='')

**Metoda close i zasada zamykania plików** → metoda zamykająca plik. Poniższy przykład otwiera plik, odczytuje dane i zamyka plik. Zawsze powinieneś zamykać swoje pliki. W niektórych przypadkach, ze względu na buforowanie, zmiany wprowadzone w pliku mogą nie być widoczne, dopóki nie zamkniesz pliku.

Np.

file = open ('simple.txt', 'r')

for line in file:

    print (line, end='')

file.close()

**Zamykanie pliku, inna wersja** → za pomocą komendy with.

Np.

with open ('simple.txt', 'r') as file:

    for line in file:

        print (line, end='')

**With i write** → to wytłumaczenie wykorzystywania instrukcji With i Write. Instrukcja with jest używana do obsługi wyjątków, aby kod był czystszy i bardziej czytelny. Upraszcza zarządzanie typowymi zasobami, takimi jak strumienie plików. Logika: with open (plik) as zmienna\_plik for zmienna\_przypisanie wartości in zmienna\_plik print (zmienna\_przypisanie end =’’). Poniższe przykłady pokazują ten sam wynik, ale z i bez rozwiązania with. With ma to do siebie, że to instrukcja, która już zawiera w sobie zamknięcie pliku po wykonaniu.

Np.

# %%

file = open('file\_path', 'w')

file.write('hello world !')

file.close()

# Rozwiązanie bez with

# %%

with open('file\_path', 'w') as file:

    file.write('hello world !')

# Rozwiązanie z with

**With i append** → wykorzystaj with, żeby dodawać (append) kolejne linie do pliku tekstowego, a nie za każdym je nadpisywać, czyli coś co write by zrobiło. W drugim argumencie open wpisz ‘a’, ale nadal skorzystaj z metody write.

Np.

file = open('data.txt', 'r')

for line in file:

print(line, end='')

# %%

with open('data.txt', 'a') as file:

file.write('Plus, 65)

**With i open** → wykorzystaj with żeby czytać zawartość pliku tekstowego. W drugim argumencie funkcji open wpisz ‘r’, ale pamiętaj, żeby w funkcji print wykorzystać metodę read.

Np.

# %%

with open('data.txt', 'r') as file:

print(file.read())

**Funkcja open, kontynuacja** → (file, mode) pozwala otworzyć plik i zwraca go jako obiekt. Najczęściej stosowane tryby otwierania plików:

r -  read - otwiera plik do odczytu, zwraca błąd jeśli plik nie istnieje. Tylko przeczyta.

a - append - otwiera plik do dopisania, tworzy plik jeśli nie istnieje. Tylko dopisze na końcu.

w - write - otwiera plik do zapisu, tworzy plik jeśli nie istnieje. Napisze wszystko co tam było.

**Lekcja\_27\_Wczytywnie plików - przykłady**

**Metoda readline** → pozwala na przeczytanie pierwszej linii pliku tekstowego. Wykorzystaj z metodą with i słowem klucz as i funkcją open. Wzór: with open (‘plik.txt’, ‘r’) as zmienna: inna zmienna = zmienna.readline() print (nowa zmienna).

Np.

# -\*- coding: utf-8 -\*-

with open('simple.txt', 'r') as file:

    line = file.readline()

    print (line)

**Readlines** → odczytuje wszystkie linie w pliku w postaci listy. Dodawaj end='' do print, żeby unikać przerw pomiędzy akapitami.

#Np.

# %%

with open ('simple.txt', 'r') as file:

    lines = file.readlines()

    for line in lines:

        print (line, end='')

**Tworzenie listy ze wszystkimi liniami, ale wyprintowanie jednej, określonej linii** → na przykładzie. Jak widać wybierana jest pierwsza linia, bo iteracja zaczyna się od 0, ale można iterować w dowolny sposób.

Np.

# %%

with open ('simple.txt', 'r') as file:

    lines = file.readlines()

print (lines[0])

**Tworzenie list z linii tekstu od określonej linii** → jak na przykładzie poniżej. Najpierw definiujesz nową zmienną, którą są wszystkie linie tekstu poza tymi, których nie chcesz, potem definiujesz zmienną, która przechowuje listy elementów w jednej liście. W tym konkretnym przykładzie usuwasz też białe znaki i dzielisz elementy znajdujące się po przecinku.

Np.

# -\*- coding: utf-8 -\*-

with open('products.txt', 'r') as file:

lines = file.readlines()

lines = lines[1:]

products = []

for line in lines:

part = line.strip().split(',')

products.append(part)

for product in products:

print(product)

**Podejście inne do readline** → while tutaj zwraca TRUE, bo jeśli jest jakikolwiek string to zwróci TRUE. Należy jednak zatrzymać działanie kodu albo będzie tak zwracać pierwszą linię w nieskończoność. W tym celu należy dodać na koniec przeczytanie kolejnej linii.

Np.

# %%

with open ('simple.txt', 'r') as file:

    line = file.readline()

    while line:

        print (line, end='')

        line = file.readline()

**As** → Słowo kluczowe as służy do tworzenia aliasu.

# %%

#Np.

with open ('simple.txt', 'r') as file:

    line = file.readline()

    while line:

        print (line, end='')

        line = file.readline()

**Proste odczytywanie całego pliku i przypisanie treści do zmiennej** → na przykładzie.

Np.

# %%

with open ('simple.txt', 'r') as file:

    lines = file.read()

    print (lines)

**Readlines i \n** → przy użyciu metody readlines jeśli po prostu wynik zostanie wyprintowany

to Python sam doda znaki \n, ktore tworza akapity w miejscu gdzie są one w pliku tekstowym.

Np.

# %%

with open('data.txt', 'r') as file:

    lines = file.readlines()

    print (lines)

**Tworzenie słowników w liście na podstawie danych z pliku tekstowego** → definiujesz pustą listę gdzie znajdą się słowniki. Otwierasz plik linia po linii. Iterujesz po każdej linii najpierw wprowadzając zmienne, które będą kluczami i pozbywasz się dla linii białych znaków oraz przecinków - kluczowa jest tu metoda split, która tworzy obiekty w liście, a obiektów powinno być tyle ile podałeś zmiennych wewnątrz iteracji. Append dla pustej listy i w append w formie słownika dodajesz wszystkie wartości, ewentualnie konwertując je.

Np.

# -\*- coding: utf-8 -\*-

products = []

with open('products.txt', 'r') as file:

for line in file:

     name, price, quantity = line.strip().split(',')

     products.append({

         'name': name,

         'price': float(price),

         'quantity': int(quantity)

     })

for product in products:

print(product)

**Formatowanie liczb w string** → jeśli chcesz formatować miejsca po przecinku w string bo np. w string pojawia się zmienna float to możesz modyfikować miejsca po przecinku tej zmiennej za pomocą następującego wzoru: {zmienna:.2f} - tu oznacza to pokazanie dwóch miejsc po przecinku.

Np.

# -\*- coding: utf-8 -\*-

with open('data.csv', 'r') as file:

    rows = file.readlines()

all\_energy = 0

iterations = 0

for row in rows[1:]:

    row = row.strip().split(',')

    all\_energy += float(row[2])

    iterations += 1

average\_energy = all\_energy/iterations

print(f'Average energy generated: {average\_energy:.2f} kWh')

**Lekcja\_28\_Zapisywanie do pliku**

**Funkcja print, kontynuacja** → Funkcja print() wyświetla określoną wiadomość na ekranie lub innym standardowym urządzeniu wyjściowym. Wiadomość może być stringiem lub dowolnym innym obiektem, obiekt zostanie przekonwertowany na string przed zapisaniem na ekranie.

Wartości parametrów:

- obiekt(y): Dowolny obiekt i tyle, ile chcesz. Zostanie przekonwertowany na string przed wydrukowaniem

- sep='separator': Opcjonalne. Określ sposób rozdzielenia obiektów, jeśli jest więcej niż jeden. Wartość domyślna to „ ”

- end='end': Opcjonalne. Określ, co wydrukować na końcu. Wartość domyślna to „\n” (przesunięcie wiersza)

- file: opcjonalny. Obiekt z metodą zapisu. Wartość domyślna to sys.stdout

- flush: opcjonalny. Wartość logiczna określająca, czy dane wyjściowe są opróżniane (prawda), czy buforowane (fałsz). Wartość domyślna to Fałsz

**Zapisywanie danych** → na przykładzie listy w komendzie open piszesz nazwę nowego pliku i literę “w” w drugim argumencie. open('techs.txt', 'w'). W = write. Funkcja print ma tutaj za zadanie przypisanie danych do pliku. Ustaw parametr file na plik. Pamiętaj, że iteracja na liście zwraca każdy obiekt, czyli słowo osobno. Gdyby to był zwykły string iteracja zapisze w stylu: linia = znak, linia = znak, idt. Ten sposób zapisu nadpisuje poprzednie dane. Jeśli plik nie istnieje wykonanie zapisu go stworzy.

Np.

# %%

techs = ['python', 'java', 'sql', 'r', 'scala']

with open('techs.txt', 'w') as file\_II:

    for tech in techs:

        print (tech, file=file\_II)

**Write** → metoda write wykorzystywana jest zapisu treści w pliku. Zapisu dokonaj za pomocą open w trybie ‘w’, oraz iterowaniu, gdzie z każdą iteracją używasz metody write na nazwie pliku w instrukcji with. Pamiętaj, że jeśli na końcu zapisywanej linii tekstu wstawisz \n to w pliku tekstowym na końcu linii pojawi się enter.

Np.

# -\*- coding: utf-8 -\*-

products = [

{'name': 'T-shirt', 'price': 29.99},

{'name': 'Shoes', 'price': 99.99},

{'name': 'Pants', 'price': 49.99},

]

with open('products.txt', 'w') as file:

for product in products:

     file.write(f"{product['name']},{product['price']}\n")

**Drugi sposób zapisu. Metoda file** → na przykładzie liczby parzyste mniejsze niż 100. Najpierw przygotuj odpowiednio zmienną. Jeśli zostawisz kod jako iteracje i oddawanie cyfr nie liter to wystąpi błąd. Dlatego należy przekonwertować zwracane obiekty na stringi. Należy też dodać znak przejścia do następnej linii, inaczej w pliku będzie ciąg cyfr ('\n'). Ten sposób nadpisuje dane.

# Np.

# %%

even\_numbers = list(range (100))[::2]

with open ('numbers.txt', 'w') as numbers\_file:

    for number in even\_numbers:

        numbers\_file.write(str(number) + '\n')

**Rozwiązanie dodające dane, a nie nadpisujące. Tryb append** → czyli literka 'a' w drugim argumencie funkcji open. Dane zostaną dodane w kolejnej linii kodu.

# Np.

# %%

techs = ['python', 'java', 'sql', 'r', 'scala']

with open('techs.txt', 'a') as file\_II:

    for tech in techs:

        print (tech, file=file\_II)

**Zapisywanie do pustej listy danych z pliku tekstowego, linia po linii** → Python z automatu iteruje linia po linii. W append aby nie zapisywał się znak akapitu \n to musisz od każdej linii odjąć znak.

Np.

# %%

technologies = []

with open ('techs.txt', 'r') as file:

    for line in file:

        technologies.append(line[:-1])

print (technologies)

**Rozwiązanie write i print. Print jest szybsze** → na dwóch przykładach.

Np.

# %%

lista = ['python', 'java', 'sql', 'sas']

with open ('techs.txt', 'a') as file:

    for line in lista:

        file.write(line + '\n')

Albo:

# %%

techs = ['python', 'java', 'sql', 'sas']

with open('techs.txt', 'w') as file:

    for tech in techs:

        print(tech, file=file)

**Lekcja\_29\_Zapisywanie do pliku - przykład**

**Format\_kontynuacja** → Metoda format() formatuje określone wartości i wstawia je do symbolu zastępczego string. Symbol zastępczy jest zdefiniowany za pomocą nawiasów klamrowych: {}. Metoda format() zwraca sformatowany ciąg znaków. Wewnątrz symboli zastępczych możesz dodać typ formatowania, aby sformatować wynik (poniżej niektóre możliwości):

:< Do lewej wyrównuje wynik (w dostępnym miejscu)

:> Do prawej wyrównuje wynik (w dostępnym miejscu)

Jak na poniższym przykładzie, dostępna przestrzeń oznacza, że jeśli sformatujesz trzy-literowy tekst z potrójnym wyrównaniem to nic się nie stanie. Dopiero poczwórne wyrównanie zrobi jedno wcięcie.

Np.

# %%

print('{:>4}'.format('123'))

na poniższym przykładzie “Leszek” zostanie wyrównanie do “Dekiert” po prawej stronie. Tzn, przed “Leszek” pojawi się spacja. Jeśli znaki “mniejsze niż” zostałyby zmienione na “większe niż” to wyrównaie tekstu byłoby do lewej strony.

name\_1 = 'Leszek'

name\_2 = 'Dekiert'

print('{:>7}|\n{:>7}|'.format(name\_1, name\_2))

Przykład z ćwiczenia:

# -\*- coding: utf-8 -\*-

print('|{:<10}|{:>10}|{:>10}|\n|{:<10}|{:>10}|{:>10}|\n|{:<10}|{:>10}|{:>10}|\n|'

'{:<10}|{:>10}|{:>10}|'.format('Product','Price','Quantity','Apple','0.50','10',

'Banana','0.30','20','Orange','0.60','5'))

**Budowanie choinki za pomocą pętli i zapisywania do pliku** → Pamiętaj, że range zwraca informację ile razy nastąpi iteracja. Liczba iteracji zatrzyma się przed wskazaną liczbą. Najpierw drugie print czyli prawa strona choinki: zwróć uwagę, że format będzie oddawać tyle gwiazdek ile będzie iteracji, zaczynając od 0. Print przypisze wynik do pliku. Pierwsze print: zacznie wpisywać gwiazdki od 10 punktu w wierszu, bo formatowanie wyrównuje do 9 - oznacza to, że wiersz tekstu będzie miał 9 znaków w sumie, zawsze. Musi też zakończyć na pustej przestrzeni zanim przejdzie do drugiego print i prawej części choinki. Łączenie printow: oba printy maja przypisanie za pomocą file i w iteracji jeden następuje po drugim, wiersz po wierszu.

Np.

# %%

with open ('tree.txt', 'w') as file:

    for j in range(2):

        for i in range(10):

            print('{:>9}'.format('\*' \* i), end='', file=file)

            print('{}'. format('\*' \* i), file=file)

**Przykład z choinka i append** → To samo tylko z 'a'.

#Np.

# %%

with open ('tree.txt', 'a') as file:

    for j in range(2):

        for i in range(10):

            print('{:>9}'.format('\*' \* i), end='', file=file)

            print('{}'. format('\*' \* i), file=file)

**SEKCJA 6. FUNKCJE**

**Lekcja\_30\_Funkcje wbudowane**

**Funkcje, kolor, pisanie** → tekst na żółto w spyder oznacza funkcje. Funkcje piszesz tak jak w Excel, czyli funkcja(obiekt).

**Abs** → funkcja zwraca wartość bezwzględną elementu który przekazujesz. Wartość bezwzględna czyli dodatnia. Wartość bezwzględna dla -10 to 10.

Np.

print (abs(-10))

# wynik 10

**Bool** → sprawdza się do testowania np. przy pętli while gdy chcę sprawdzić jaką logiczną wartość ma dany element.Ta funkcja sprawdza wartość logiczną obiektu.

Np.

bool(True)

bool(0)

# Zwróci false

**Dir** → przydaje się gdy chce przypomnieć sobie lub poznać z jakich metod mogę korzystać.

Np.

value = (1, 2, 3)

dir(value)

**Enumerate** → pozwala np. na szybką iterację po obiektach. Pierwszą wartością jest numer iteracji, a drugą obiekt, po czym następne przypisanie iteracji. W zależności od tego czy chcę żeby połączenie iteracja+obiekt znalazły się w liście, tupli czy zbiorze, muszę je sam do nich przypisać.

en = (tuple((enumerate((1, 2, 3)))))

print (en)

# Odda ((0, 1), (1, 2), (2, 3))

**Eval** → funkcja, która wykrywa w stringu liczby i pozwala na prowadzenie działań matematycznych, mimo, że to były stringi. Uwaga w tekście tylko cyfry lub znaki działań np. dodawanie. Rozpozna też nazwę zmiennej.

Np.

number = 10

word = eval('5+1')

eq = number + word

print (eq)

**Filter** → funkcja która zwraca tylko wartości true, ale poprzedzone inną operacją, która stanowi pierwszy argument. Drugim argumentem jest filtrowany ciąg znaków. Czyli po prostu filtrujesz operację. Filter nie jest tu najważniejszym elementem, raczej dodatkiem. Pamiętaj umieścić wynik na liście, tupli czy zbiorze.

print (list(filter(bool, (1, 2, 0))))

**Float** → konwertuje liczbę lub wartość tekstową podaną jako liczbę na liczbę zmiennoprzecinkową.

Np.

float(1)

float(1.5)

float('1.6')

float('77')

**Type** → funkcja zwraca typ danego elementu.

Np.

type(1)

**Help** → funkcja pomoc. Poznaj dokumentację funkcji, metod, obiektów.

help(float())

**Isinstance** → sprawdza czy obiekt należy do danej klasy obiektów (np. do int).

Np.

isinstance(1, int)

# True

isinstance(1, float)

# False

**Len** → pozwala zwracać długość danego elementu.

Np.

len ('python')

# Zwraca 6

len([[3,4], [4,5,6,7, [5,5]]])

# Zwraca 2 ponieważ nie schodzi na niższy poziom niż pierwszy w przypadku listy.

**List** → przyjmuje obiekt iterowalny i zwraca listę.

Np.

list('python')

# ['p', 'y', 't', 'h', 'o', 'n']

list(range(10))

# zwróci cały zakres od 0 do 10, wszystkie cyfry po drodze

list((10, 1, 11))

# [10, 1, 11]

**Map** → funkcja mapująca wyniki innej funkcji, która normalnie mogłaby pokazać tylko jeden wynik naraz. Np. poniższe abs może tylko jedną cyfrę naraz obrobić, ale jeśli wrzucić funkcję do listy i zmapować to można pokazać wiele cyfr poddanych abs jednocześnie. Chodzi o to, że niektóre funkcje przyjmują tylko jeden argument i można obrobić tylko jeden obiekt. Ta funkcja pozwala na obrobienie wielu naraz. Dodawaj z list i pamiętaj o przecinku między obrabianą funkcją, a obiektami.

Np.

print(list((map(abs,(-1.2, -5)))))

print(list(map(str.title,('leszek', 'dekiert'))))

value = ('leszek', 'dekiert', 'aga')

print(list(map(str.title, (value))))

**Funkcje statystyczne maximum i minimum**: maximum pozwala na wyodrębnienie najwyższej wartości ze zbioru iterowanego. Na danych tekstowych zwraca najwyższą literę alfabetu. Funkcja minimum zwraca najniższą wartość. Na obiekcie tekstowym zwraca najniższą literę alfabetu.

Np.

values = (1,2,3,0,-1,4)

words = ('Leszek', 'Dekiert')

letters = ('abcde')

max(values)

min(values)

max(words)

min(words)

max(letters)

min(letters)

**Pow** → baza i wykładnik w argumentach. Funkcja podnosi wartości do potęgi.

Np.

pow(4, 3)

pow(3, 4)

# ALBO

4 \*\* 3

**Reversed** → funkcja odwraca kolejność obiektu iterowanego. Na tekście zrobi to samo co z cyframi. Do wykorzystania z list.

Np.

list(reversed([1, 2, 3, 4]))

list(reversed('python'))

list(reversed(['python', 'sas', 'r']))

**Round** → funkcja zaokrągla wartości. Jako dodatkowy argument możesz podać ilość miejsc po przecinku. Domyślnie usuwa wszystko po przecinku. Wzór: round(wartość zaokrąglana, ilość miejsc po przecinku - opcjonalne).

Np.

height = 1.85

weight = 85

BMI = round(weight/(height\*\*2), 2)

print(f'The patient\'s BMI is: {BMI}')

**String** → konwertuje liczby na tekst.

Np.

str(1)

**Sum** → wylicza sumę elementów. Funkcja statystyczna.

Np.

sum([2, 3, 4])

**Zip** → funkcja łącząca elementy z dwóch zbiorów w tuple. Przydatne jak trzeba iterować po więcej niż jednej liście. Będzie działać do momentu najkrótszej listy jeśli któraś lista będzie dłuższa. Na obiektach tekstowych funkcja łączy w tuple kolejne litery. Trzeba używać z funkcją list. Można łączyć więcej niż dwa zbiory

Np.

lista\_1 = 1, 2, 3, 4

lista\_2 = 5, 6, 7, 8

list(zip(lista\_1, lista\_2))

list(zip('python', 'sql'))

techs = ('python', 'sas', 'SQL')

numbers = (11, 22, 33, 44)

other = ('a', 'b, c')

print(list(zip(techs, numbers, other)))

**Sorted** → Funkcja sorted() sortuje elementy danej iterowalności w określonej kolejności (rosnącej lub malejącej) i zwraca ją w postaci listy. Funkcja ma 3 parametry: zmienna, klucz - dodatkowa, reverse=True/False. Funkcja, która może zmienić sortowanie, np. len, abs. Pamiętaj, że żeby przypisanie nowej listy musi być w nowej wartości.

Np.

values = [-5, 2, 8, -1, 9, -3]

sorted\_values = sorted(values, key=abs, reverse=True)

print(sorted\_values)

**Enter w print** → jeśli chcę uzyskać enter w print wystarczy, że wpiszę print ('\n').

**End w print** → print docelowo kończy string przechodząc do nowej linii. Robi po prostu na końcu enter. Jeśli jednak chcę, żeby na końcu danego printu był inny znak, bo np. chcę żeby zaraz po nim pojawił się kolejny print (bo np. po nich iteruję i chcę jeden ciąg znaków, a nie po enterach), muszę wykorzystać jego argument end = jak w poniższym przykładzie.

Np.

for number\_1 in range(1,11):

    print (number\_1, end = ' ')

print ('\n')

for number\_2 in range(0,10,2):

    print (number\_2, end = ' ')

print ('\n')

for number\_3 in range(100,0,-10):

    print (number\_3, end = ' ')

# -\*- coding: utf-8 -\*-

**Any** → zwraca wartość logiczną True jeśli przynajmniej jeden z argumentów ma wartość logiczną True. Warto wykorzystać chociażby do sprawdzania czy obiekty zawierają określone reguły, np. co najmniej 1 małą literę, etc.

Np.

mylist = [False, True, False]

x = any(mylist)

# %%

password = 'my\_password\_123'

if len(password) <= 8:

print('The Password is too short.')

elif any(char.isdigit() for char in password) != True:

print('The password must contain at least one digit.')

elif any(char.isupper() for char in password) != True:

print('The password must contain at least one uppercase letter.')

elif any(char.islower() for char in password) != True:

print('The password must contain at least one lowercase letter.')

else:

print('The password is complex enough.')

**Lekcja\_31\_Definiowanie własnych funkcji - część I**

**Definiowanie funkcji w pythonie** → Służy do tego komenda def, nazwa funkcji w nawiasie można lub nie dodatkowe argumenty i dwukropek. Wywoływanie funkcji: użyj jej nazwy i wywołaj nawiasem okrągłym.

Np.

def funkcja\_1():

    print ('Pierwsza funkcja urchomiona!')

funkcja\_1()

**Definiowanie funkcji z argumentami i format** → definiowanie funkcji może zawierać format i nawiasy kwadratowe do wypełnienia wymaga to podania w komendzie def argumentów, które będą tożsame do procesu podstawiania z format. Przykład poniżej. Pamiętaj, że możesz manipulować kolejnością lub w ogóle zrezygnować z jakichś wartości.

Np.

def funkcja\_2(x, y):

    print ('Argumenty w funkcji to: {} i {}'.format(x, y))

funkcja\_2(1,2)

def funkcja\_2\_1(x, y):

    print ('Argumenty w funkcji to: {0} i {0}'.format(x, y))

funkcja\_2\_1(1,2)

**Funkcja bez parametru** → możesz już na etapie definiowania funkcji określić jaka wartość ma być przypisana do argumentu, za pomocą znaku równości. Ale jeżeli przy wywołaniu funkcji przypiszę inną wartosć dla argumentu to ta wartość ma pierwszeństwo. Mogę też wywoływać funkcje pisząc parametry jak w definicji czyli ze znakiem równości.

Np.

def funkcja\_3(x, y=10):

    print ('Argument x={0}, a argument y={1}'.format(x, y))

funkcja\_3(2)

funkcja\_3(2,9)

funkcja\_3(x=1,y=2)

**Funkcja pierwiastkowania i import funkcji** → bardzo dużo funkcji już gdzieś jest i można je pobrać z biblioteki. Funkcja SQRT (skrót od pierwiastka). Można ją pobrać z biblioteki "math". Nazywa się to importowanie modułu math. W argumencie wartość którą chciałbym spierwiastkować. Dużo funkcji nie trzeba samemu zaimplementować bo ktoś już je przygotował. Trzeba tylko wiedzieć w jakiej bibliotece szukać.

Np.

import math

math.sqrt(2)

**Moduł math i exponent** → moduł liczący wartość funkcji eksponencjalnej, czyli funkcja wykładnicza o podstawie równej e e (czyli podstawie logarytmu naturalnego).

Np.

import math

math.exp(1)

**Moduł funkcji trygonometrycznej** → sinus, czyli oznaczany w Polsce jako sin \sin – stosunek długości przyprostokątnej a a leżącej naprzeciw kąta α \alpha i długości przeciwprostokątnej c ; {\displaystyle c;}

Np.

import math

math.sin(2)

**Funkcje, a obiekty** → w Python wszystko jest obiektem. Pamiętaj, że jeśli tworzysz funkcję to ona jest obiektem co można potwierdzić funkcją type. Po stworzeniu ta funkcja też będzie zapisana.

**Funkcje zwracające wartości** → poprzednie funkcje tylko coś printowały poniższe natomiast zwrócą. Tutaj potrzebna jest instrukcja return i określenie co zwracasz (return), co ma być zrobione z argumentami które podasz. Możesz też przypisać zmienną do wyniku. Czyli generalnie wygląda to tak, że podajesz ile będzie wprowadzanych wartości i pod postacią jakich obiektów one będą, a potem podajesz co będzie z nimi robione (return).

Np.

def add(x, y):

    return x + y

add(1,2)

result = add(1,2)

**Wymuszanie klasy argumentu** → jeśli chcę żeby było z góry założone jaka klasa argumentu może być podawana w funkcji, mogę to zrobić za pomocą dwukropka. Jak w przykładzie.

Np.

def add\_2(x: int, y: int):

    return x + y

add\_2(1,0.3)

**Dokumentacja funkcji** → Warto zawsze robić dokumentację. Najlepiej robić to między 3 cudzysłowami. Pewną formą dokumentacji będzie też podanie jakiego typu klasy argumentów. Wymagam, co warto też robić (chodzi o dwukropek i wymuszanie klasy).

Np.

def subtract(x: int, y: int):

    """

    Odejmuje od siebie dwie liczby

    """

    return x - y

subtract(10, 6)

**Reproduktowanie funkcji** → jeżeli jakieś działania są powtarzalne to wtedy należy korzystać z funkcji. Możesz np. przygotować menu prostego programu i powtarzać je w wielu takich programach.

Np.

def print\_menu():

    print('Start programu...')

    print('\*' \* 30)

    print('''Wybierz jedną z opcji:

          1. Logowanie.

          2. Zamknięcie programu.''')

    print('\*' \* 30)

    print('Zamykanie programu...')

print\_menu()

**Lekcja\_32\_Definiowanie własnych funkcji - część II**

**NoneType** → typ obiektu, który pojawi się przy przypisaniu nazwy zmiennej dla wyniku zdefiniowanej funkcji, która zwraca tylko print. Jest tak, ponieważ po prostu zwracanie print nie jest żadną wartością.

Np.

def Nontype(x):

    print(x)

example = Nontype(10)

**Definiowanie funkcji i iteracje lub jeżeli** → na przykładzie printowania wartości parzystych do określonego maximum. Pamiętaj, że w iteracji nie zostanie uwzględniona ostatnia wartość, którą podajesz (w przykładzie 10) bo iteracja zaczyna się od 0 więc dodaj 1 do range.

Np.

def even\_numbers(maximum):

    for i in range(maximum + 1):

        if i%2 == 0:

            print(i)

even\_numbers(10)

num = even\_numbers(20)

**Przypisywanie wyniku iteracji do zmiennej** → należy to zrobić za pomocą listy. W tym celu potrzebujesz pustą listę zdefiniowaną przed iteracją i jako wynik iteracji za pomocą metody append  dodawać do niej to co przejdzie przez filtr if. Ponadto potrzebne jest żeby instrukcja return na samym końcu iteracji zwracała cały wynik iteracji, czyli wszystkiego przypisanego do wcześniejszej pustej zmiennej. No i na końcu to przypisujesz do zmiennej.

def even\_num (liczby):

    even = []

    for l in range(liczby + 1):

        if l%2 != 0:

            even.append(l)

    return even

przypisanie = even\_num(20)

**Funkcja tworząca plik tekstowy i wypełniająca jego treść** → Zacznij od określenia argumentów jako nazwa pliku i zawarty tekst. Chcesz żeby plik był typu, że go wypełniasz, czyli 'w'. Na końcu w print podajesz tekst.

Np.

def write\_file(file\_name, text):

    with open(file\_name, 'w') as file:

        print(text, file=file)

write\_file('przykład.txt', 'Jeden.\nDwa.\nTrzy.')

**Zmiana domyślnych parametrów w stworzonej funkcji** → jeśli tworzę funkcję z domyślnymi parametrami, to nadal mogę przy wywoływaniu tej funkcji zmienić te parametry. Robię to pisząc w nawiasie nazwy tych parametrów i czemu się równają.

Np.

def calculate\_profit(stan\_konta=1000, stopa\_procentowa=0.02, czas\_w\_latach=1):

    return stan\_konta \* (1 + stopa\_procentowa) \*\* czas\_w\_latach

calculate\_profit()

calculate\_profit(stan\_konta=2000)

**Funkcja z negacją w if** → na przykładzie.

Np.

def print\_odd\_numbers(odd):

    numbers = []

    for o in range(odd + 1):

        if o%2 != 0:

            numbers.append(o)

    return numbers

od = print\_odd\_numbers(20)

**Funkcja z określeniem True/False w if** → na przykładzie.

Np.

def is\_even(num):

    if num %2 == 0:

        num = True

        print('Liczba jest parzysta!')

    else:

        print('Liczba jest nieparzysta...')

is\_even(3)

**Funkcja z or w if** → na przykładzie.

Np.

def divisible\_by\_3\_or\_5(first, last):

    numbers = []

    for fl in range(first, last + 1):

        if fl %3 == 0 or fl %5 == 0:

            numbers.append(fl)

    return numbers

div = divisible\_by\_3\_or\_5(2, 14)

**Lekcja\_33\_Dokumentowanie funkcji**

**Dokumentowanie funkcji** → niezbędne do pracy z python. Zdarza się, że kod bez dokumentacji będzie bezużyteczny. Przykład pisania dokumentacji poniżej. Dokumentację wywołuje się za pomocą funkcji help.

Np.

def add(a, b):

    """Zwraca sumę dwóch liczb.

    Inputs:

        a: int

        b: int

    Outputs:

        sum: int"""

    return a + b

help(add)

**Lekcja\_34\_Wyrażenie lambda**

**Wywoływanie funkcji, a przypisanie jej do zmiennej** → mogę wywołać funkcję na przykład za pomocą print, ale należy pamiętać, że równie dobrze, mogę funkcję tą przypisać do określonej zmiennej i wywoływać w dokładnie taki sam sposób. Pamiętaj tylko, że przy przypisaniu nie stawiasz nawiasu dla argumentów.

Np.

def parabola(x):

    return x\*\*2 + 3

print(parabola(30))

funkcja\_1 = parabola

print(funkcja\_1(30))

**Wyrażenie lambda** → instrukcja, która pozwala tworzyć funkcje w jeden linii w Python. Warto korzystać z tego rozwiązania do szybkiego pisania funkcji w jednej linii. Wzór: nazwa\_nowej\_zmiennej = lambda argument/y: działanie na argumencie/tach. Wywoływanie: po prostu wywołujesz jak funkcję przypisaną do zmiennej.

f = lambda x: x\*\*2 + 3

f(30)

print(f(30))

f\_2 = lambda word: word.upper()

f\_2('leszek')

add = lambda a, b: a + b

add(1, 2)

f\_3 = lambda word\_1, word\_2: word\_1 + ' ' + word\_2

f\_3('Leszek', 'Dekiert')

**Pisanie funkcji w środku kodu** → plusem lambdy jest to, że można ją wygodnie pisac "w srodku" innego kodu. Poniżej na przykładzie wykorzystania funkcji map, która bierze każdy obiekt iterowalny i przepuszcza przez wybraną (lub własnie napisaną) funkcję. W takim przypadku nie musisz przy pisaniu lambdy przypisywać funkcji do zmiennej, a nawet nie możesz.

lista = ['python', 'sas', 'sql']

list(map(lambda word: word.upper(), lista))

**Przykład wykorzystania lambdy i len do stworzenia tupli w liście** → jak widzisz aby umieścić wyniki mapowania w tuplach len i samo word (czyli w tym przypadku np. 'python') są w nawiasie, co stworzy tuple. Jeśli dałbyś tam nawiasy kwadratowe to byłyby to listy w liście.

Np.

lista = ['python', 'sas', 'sql']

list(map(lambda word: (len(word), word), lista))

**Lambda i agregowanie danych** → możesz wykorzystać lambdę w celu zastosowania na obiekcie funkcji. Chodzi o to, że możesz napisać funkcję za pomocą klasycznego def, której jednym z argumentów będzie podawana wartość, a drugim funkcja, którą wobec tej wartości zastosujesz. Pamiętaj, że w drugim argumencie musisz napisać dosłownie wzór tego co stanie się z pierwszym.

def apply\_function(x, fn):

    return fn(x)

apply\_function('leszek', lambda x: x.upper())

**Lambda i listy** → pamiętaj, że lista, tupla, etc. są obiektami. Możesz zatem potraktować listę jako obiekt x.

apply\_function([12, 42], lambda x: sum(x))

**Lambda, a numbers i funkcja-klucz** → numbers jest przykładową funkcją w ramach której możemy zagnieździć inną funkcję (np. właśnie napisaną przez lambdę)w celu zmiany jej działania. W funkcji numbers jest możliwość dodania argumentu key=, w celu napisania za pomocą funkcji klucza na podstawie którego sortowanie ma się odbywać. Na poniższym przykładzie znajdują się liczby ujemne i dodatnie już posortowane w standardowy sposób i zwróć uwagę, że powstały klucz zmieniający liczby na całkowite (abs) nie zmieni ich wartości jak nowa lista będzie wyprintowana. Ona tylko wpłynie na kolejność.

Np.

lista = [-3, -2, -1, 0, 1, 2, 3]

print(sorted(lista, key=lambda x: abs(x)))

**Lambda, sorted i sortowanie po n-tym obiekcie** → powiedzmy, że jest tupla w liście. Chcę sortować nie po pierwszym obiekcie alfabetycznie, a po drugim. W tym celu ustawiamy klucz żeby sorted działało właśnie na drugi obiekt wskazując na niego w lambda.

lista\_2 = [('jeden', 'one'), ('dwa', 'two'), ('trzy', 'three')]

sorted(lista\_2, key=lambda x: x[1])

**Lambda, a iterowanie w map** → jak widać na poniższym przykładzie jako x był kolejny w kolejności iterowalny obiekt. A to dlatego, że to jaka lista jest iterowana, z jakiej listy wyciągałem iterowalne obiekty zostało podane w drugim argumencie map.

Np.

cities = ['Warsaw', 'London', 'Berlin', 'New York']

lista\_3 = list(map(lambda x: x[0:3], cities))

print(lista\_3)

**Łączenie def i lambda** → możliwe jest w return umieścić lambdę. Jak na przykładzie poniżej.

Np.

lista\_1 = ['apple', 'pear', 'banana', 'pineapple', 'orange']

lista\_2 = ['laptop', 'mouse', 'keyboard', 'screen']

def sort\_by\_length(x):

    return sorted(x, key=lambda y: len(y))

sort\_by\_length(lista\_2)

**Lekcja\_35\_Przekazanie dowolnej liczby argumentów: \*args**

**Dowolna ilość argumentów i rozpakowywanie** → jeśli nie wiem ile argumentów chcę podać w funkcji bo nie wiem ile będzie potrzebne, mogę spakować tą nieznaną ilość. Pierwszy argument w takiej funkcji piszę normalnie (np. x), ale nie muszę, może być bez normalnego argumentu, a drugim jest oznaczenie spakowania (najczęściej stosuje się '\*args'). Jest to iterowalny obiekt. Teraz możesz pod \*args podstawić dowolną liczbę argumentów. Pamiętaj, że jak na poniższym przykładzie z pętlą for w pętli nie korzystasz już z gwiazdki, którą pisałeś przy definiowaniu  funkcji.

Np.

def test\_args(x, \*args):

    print('Pierwszy argument:',x)

    for a in args:

        print('Kolejny argument to:',a)

test\_args(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

**Obowiązkowe argumenty i tuple** → obowiązkowych argumentów możesz podawać więcej niż jeden. Ponadto zwróć uwagę, że obiekty tworzone w print w definicji funkcji domyślnie wpadają do tupli.

Np.

def test\_1(x, y, \*args):

    print('x =',x)

    print('y =',y)

    print('\*args =',args)

test\_1(1,2,3,4,5)

**Wykorzystanie samego \*args** → jak wspomniałem wyżej do zdefiniowania funkcji mogę wykorzystać same dowolne argumenty.

Np.

def dowolna(\*args):

    print(sum(args))

dowolna(1,2,3,4,5)

**Dowolna liczba argumentów, if i for** → na przykładzie iloczynu wszystkich wartości.

def multiply(\*args):

    if len(args) == 0:

        return None

    result = 1

    for a in args:

        result \*= a

    return result

multiply(2, 4, 2)

**Rekurencja** → jest jednym ze sposobów rozwiązywania problemu. Sposób ten zakłada rozbijanie problemu na coraz to mniejsze i mniejsze części, aż dojdziemy do momentu, że problem będzie tak niewielki, że zostanie łatwo rozwiązany. W programowaniu rekurencja sprowadza się do funkcji, które same siebie wywołują.

**Lekcja\_36\_Przekazanie dowolnej liczby argumentów: \*kwargs**

**Kwargs** → to skrót od "Key word arguments". Kwargs pozwala na przekazywanie dowolnej liczby argumentów do funkcji, tak jak args, ale z tą różnicą, że te argumenty będą miały przypisane nazwy. Owe połączenia nazwa: argument, będą zapisywane w postaci słownika. Inaczej jak w przypadku args gdzie zapis odbywa się w tupli. Inną różnicą jest też zapis takiej funkcji. Args wymagało jednej gwiazdki, a kwargs dwóch.

Np.

def funkcja(\*\*kwargs):

    for kwarg in kwargs:

        print(kwarg)

funkcja(\*\*{'a': 1, 'b': 2})

**Przykład wykorzystania kwargs i args** → Jak poniżej. Zwróć uwagę, że nie muszę oddzielać args i kwargs. Zostanie to zrobione automatycznie poprzez rozpoznanie, które obiekty są pojedyncze, a które stanowią pary.

Np.

def fun(\*args, \*\*kwargs):

    print(args)

    print(kwargs)

fun(1, 2, 3, 4, a=1, b=2, c=3, d=4)

Praktyczne wykorzystanie kwargs → jak w przykładzie.

Np.

def fun\_2(\*args, \*\*kwargs):

    print(sum(args))

    print(kwargs.values())

fun\_2(1, 2, 3, 4, a=1, b=2, c=3, d=4)

**F-string** → Ideą f-string jest uproszczenie interpolacji string. Aby utworzyć f-string, poprzedź ją literą „f”. Sam ciąg znaków można  sformatować w taki sam sposób, jak w przypadku funkcji str.format(). F-stringi zapewniają zwięzły i wygodny sposób osadzania wyrażeń Pythona w literałach łańcuchowych w celu formatowania. Pamiętaj, że w f-string w nawiasach kwadratowych podajesz nazwy zmiennych, które się tam znajdą.

Np.

name = input('State Your name: ')

age = input('How old are You: ')

print(f'My name is {name} and I am {age} years old.')

Get, \*\*kwargs jako słownik i f-string --> f-string działa jak wytłumaczono powyżej. Natomiast zwróć uwagę, że dla kwargs wykorzystano metodę get żeby dla określonych zmiennych przypisać wartości, a pominąć klucze. Po czym za pomocą f-string przypisano je do tekstu. Dla metody get wykorzystano też drugi, opcjonalny argument określający wartość opcjonalną. W przykładzie zakomentowano linię kodu, która robi to samo co f-string.

def person\_info(\*\*kwargs):

    name = kwargs.get('name', 'John')

    age = kwargs.get('age', '<unknown>')

    city = kwargs.get('city', '<unknown>')

    return f'{name} is {age} years old and lives in {city}.'

#    return '{} is {} years old and lives in {}.'.format(name, age, city)

person\_info(name = 'Leszek', age = 30, city = 'Poznań')

**F-string i symbole ucieczki** → jeśli w tekście poprzedzonym f-string znajdują się słowa kluczowe dla Python, np. name, możesz umieścić je w zwykłym cudzysłowiu.

Np.

missions = [

{

     'name': 'Apollo 11',

     'date': '20.07.1969',

     'status': 'completed',

},

{

     'name': 'Mars Pathfinder',

     'date': '04.07.1997',

     'status': 'completed',

},

{

     'name': 'Chang\'e 4',

     'date': '03.01.2019',

     'status': 'in progress',

},

{

     'name': 'Cassini',

     'date': '15.10.1997',

     'status': 'completed',

},

]

for m in missions:

if m['status'] == 'in progress':

     continue

print(f'Mission {m["name"]} took place on {m["date"]}')

**Lekcja\_37\_Generator logów**

**Generator logów** → przygotowanie takiego generatora (przykłady poniżej) pozwala użytkownikowi na nadzorowanie na którym etapie pracy, programu jestesmy. Zwróć uwagę, w poniższym przykładzie, że jest właściwie funkcja w funkcji. Zdefiniowałeś jedną funkcję i potem zawarłeś ją w drugiej. Po drugie ustawiłeś w pierwszej funkcji jako domyślny argument czasu, żeby zawsze właściwa funkcja iterując podawała aktualny czas. Teraz w jako argument funkcji iterując podajesz tylko jeden argument.

Np.

import datetime

import time

def log(message, dt=datetime.datetime.utcnow()):

    print(dt, message)

def logi(\*args):

    for command in args:

        log(command)

        time.sleep(1)

logi('Uruchomienie systemu', 'Logowanie', 'Restart', 'Wylogowanie')

**Biblioteka datetime** → biblioteka, z której można wyciągnąć daty i formatowanie dat.

Np.

import datetime

print(datetime.datetime.utcnow())

**Utcnow** → metoda, połączona z typem danych datetime zwraca dzisiejszą datę.

#Np.

import datetime

print(datetime.datetime.utcnow())

**Metoda sleep** → połączona z biblioteką time, pozwala na uśpienie, zatrzymanie działania programu, np. iteracji na określony czas - jeśli w argumencie podasz 1 to zatrzyma na 1 sekundę, itd.

Np.

import datetime

import time

def log(message, dt=datetime.datetime.utcnow()):

    print(dt, message)

def logi(\*args):

    for command in args:

        log(command)

        time.sleep(1)

logi('Uruchomienie systemu', 'Logowanie', 'Restart', 'Wylogowanie')

**Lekcja\_38\_Zakres zmiennych - global, local, nonlocal**

**Nadpisywanie zmiennych w Python** → w Python można nadpisywać zmienne i pamiętaj, że dzieje się to w kolejności kodu.

Np.

i = 2

j = i

i = 3

**Zmienne globalne i lokalne** → przypisywanie wartości do zmiennych w tradycyjny sposób to przypisywanie zmiennych globalnie, a lokalnie to wewnątrz określonych operacji, np. w tworzeniu funkcji. Chodzi o to, że zmiana wartości zmiennej wewnątrz funkcji zadziała prawidłowo jak tego chcę, ale poza funkcją w eksploratorze zmiennych, zmienna nadal będzie miała charakter globalny, a nie ten przypisany w funkcji.

Np.

b = 10

def funkcja\_1():

    b = 8

    print(b)

funkcja\_1()

**Błąd używania zmiennej lokalnej globalnie** → nie ma możliwości wykorzystania, wywołania zmiennej utworzonej lokalnie (np. w funkcji) globalnie, np.printując jej wartość. Zwrócony zostanie błąd.

def funkcja\_2():

    x = 5

    print(x)

funkcja\_2()

print(x)

**Błąd zmiany wartości zmiennej globalnej w środowisku lokalnym** → jesli mam globalnie podaną wartość zmiennej to nie będę w stanie na stałe, globalnie zmienić jej wartości w środowisku lokalnym (np. w zdefiniowanej funkcji)

Np.

tech = 'Python'

print(tech)

def change\_tech(new\_tech):

    tech = new\_tech

    print(tech)

change\_tech('java')

print(tech)

**Global** → instrukcja, która podnosi wybraną zmienną do rangi globalnej, czyli dotyczącej całego kodu, a nie tylko fragmentu w zamkniętym środowisku, np. funkcji. Teraz zmiany, które wprowadzamy, działają też poza lokalnym środowiskiem.

Np.

qwe = 10

def fun\_1(x):

    global qwe

    qwe = x

    return qwe + 100

fun\_1(20)

**Instrukcja nonlocal** → to instrukcja, która wykorzystywana jest w zagnieżdżonych funkcjach. Jest funkcja nadrzędna i podrzędna albo funkcja rodzic i dziecko. Jeśli chcę, żeby zmiana zmiennej w funkcji dziecko zmieniło też wartość zmiennej u funkcji rodzin, skorzystam z nonlocal. Pamiętaj jednak, że globalnie wartość zmiennej pozostaje bez zmian.

Np.

poziom = 0

def F1():

    poziom = 1

    def F2():

        nonlocal poziom

        poziom = 2

        print("Poziom F2 = ", poziom)

    F2()

    print("Poziom F1 = ", poziom)

print("Poziom globalny = ", poziom)

F1()

**Zagnieżdżanie funkcji** → F2 zadziała tylko w środowisku F1. Spróbuj printować F2 - nic się nie stanie. Poza tym jeśli w def F2 najpierw wyprintujesz ("Poziom F1 = ", poziom) to też nic się nie zmieni bo najpierw wykonuje się istrukcja że poziom = 1

Np.

poziom = 0

def F1():

    poziom = 1

    def F2():

        nonlocal poziom

        poziom = 2

        print("Poziom F2 = ", poziom)

    F2()

    print("Poziom F1 = ", poziom)

print("Poziom globalny = ", poziom)

F1()

**Lekcja\_39\_Generatory**

**Next** → funkcja zwracająca kolejny (zaczynając od pierwszej) iterowany obiekt. Wykorzystywana wraz z generatorem. W argumencie nazwa zmiennej. Next będzie zwracać obiekty, dopóki nie skończą się w zmiennej, czyli właściwie w powstałym generatorze.

Np.

def generator():

    yield 1

    yield 2

    yield 3

gen = generator()

# %%

next(gen)

**Generatory** → funkcje, które zwracają obiekt generator, z którego można wyciągać wartości i traktować generator jako zwykły iterator. W definiowaniu generatora nie wykorzystuje się instrukcji return, a yield. Za pomocą yield podajesz tyle obiektów ile ma być iteracji. Wywołaj iteracje za  pomocą funkcji next przy założeniu, że wcześniej przypiszesz nowej zmiennej nowo powstałą funkcję generator. Zwróć uwagę, że funkcja next znajduje się w nowej części kodu. Jest tak, ponieważ jeśli spróbujesz uruchomić kod w całości to iteracja będzie się zaczynała i wracała do punktu wyjścia.

Np.

def generator():

    yield 1

    yield 2

    yield 3

gen = generator()

# %%

next(gen)

**Generator, a iteracja** → jesli chcę żeby generator iterował po literach w obiekcie, można zaimplementować iterację. Potem next będzie zwracać iterowane obiekt (aż się nie skończą)

Np.

def generator\_2(word):

    letters = list(word)

    for letter in letters:

        yield letter

gen\_2 = generator\_2('Leszek')

# %%

next(gen\_2)

**Generator, a pętle** → zamiast wyciągać pojedyncze obiekty za pomocą next możesz zastosować pętlę, która od razu wyciągnie wszystkie obiekty. Stwórz więc pętlę, która będzie zwracać pojedyncze obiekty, a potem za pomocą kolejnej pętli zwróć wszystkie obiekty. Nie stosujesz tu już next.

Np.

def generator\_2(word):

    letters = list(word)

    for letter in letters:

        yield letter

for item in generator\_2('predator'):

    print(item)

**Generator i if** → wykorzystuj klauzulę if do konkretyzowania iteracji i wyciągania danych za pomocą generatorów. Można potem również szybko po takim generatorze iterować.

Np.

files = ['python.py', 'leszek.txt', 'aga.py', 'lol.mp3']

def gen\_files(lista):

    for item in lista:

        if item.endswith('.py'):

            yield item

gen = gen\_files(files)

# %%

next(gen)

# %%

for i in gen:

    print(i)

**Generator i pętla while** → pętla while połączona z generatorem pozwala na  niekończącą się pętlę.

Np.

    def fibonacci():

        a, b = 0, 1

        while True:

            yield a

            a, b = b, a + b

# %%

fib = fibonacci()

# %%

next(fib)

**SEKCJA 7. PRZEKSZTAŁCANIE LISTY, SŁOWNIKA, ZBIORU**

**Lekcja\_31\_List comprehention**

**List comprehension** → jest to funkcjonalność Python, która pozwala szybciej i krócej prowadzić obliczenia. Na powyższym przykładzie widać iterowanie do 100 i podnoszenie każdego elementu do kwadratu, najpierw w standardowy sposób, a potem za pomocą List comprehension. List comprehension wymaga, żeby cały zapis znajdował się w liscie [], na początku pojawiło się obliczenie, potem for iterator in obiekt iterowany.

Np.

lista\_1 = []

for i in range(100):

    lista\_1.append(i\*\*2)

#Standardowy sposób

# %%

lista\_2 = [i\*\*2 for i in range(100)]

#List comprehension

**List comprehension i połączenie for i if** → do powyższego przykładu mogę dodać klauzulę if i w list comprehension znajdzie się ona na końcu. W przykładzie chcę żeby dodawane elementy do listy były tylko podzielne przez 5.

Np.

# %%

lista\_1 = []

for i in range(100):

    if i % 5 == 0:

        lista\_1.append(i\*\*2)

#Standardowy sposób

# %%

lista\_2 = [i\*\*2 for i in range(100) if i % 5 == 0]

**Łączenie elementów dwóch list ze sobą za pomocą pętli for** → Najpierw  iteracja po jednym elemencie i w tą iterację wbudowana iteracja po  analogicznym elemencie i na końcu łączenie elementów w pamięci.

Np.

letters = ['a', 'b', 'c']

numbers = [1, 2, 3]

results = []

for letter in letters:

    for number in numbers:

        results.append(letter + str(number))

print(results)

**Łączenie elementów przez list comprehension i składnia** → na poniższym przykładzie widać składnię list comprehension. Pierwszy w liscie jest wynik,  następnie obliczenia czyli pętle.

Np.

letters = ['a', 'b', 'c']

numbers = [1, 2, 3]

results\_2 = [letter + str(number) for letter in letters for number in numbers]

**Łączenie elementów w list comprehension z warunkiem if** → piszesz najpierw wynik iteracji, potem iteracje, a na końcu warunek.

Np.

letters\_1 = ['a', 'b', 'c']

letters\_2 = ['a', 'b', 'c']

results = [letter\_1 + letter\_2 for letter\_1 in letters\_1 for letter\_2 in \

           letters\_2 if letter\_1 != letter\_2]

**Łamanie kodu przez backslash** → jeżeli chcesz łamać pisanie kodu zrób to za pomocą backslash jak w przykładzie poniżej.

Np.

letters\_1 = ['a', 'b', 'c']

letters\_2 = ['a', 'b', 'c']

results = [letter\_1 + letter\_2 for letter\_1 in letters\_1 for letter\_2 in \

           letters\_2 if letter\_1 != letter\_2]

**Tworzenie list wewnątrz list** → składnia to iterowanie po powstałej liście określoną ilość razy. j for j jest potrzebne żeby przeiterować po range(10) bez tworzenia zmiennych. Następnie wynik iteruje się określoną w drugim range ilość razy.

Np.

[[j for j in range(10)] for i in range(5)]

**Lista list gdzie parami są tuple** → definiujesz tuplę po czym iterujesz jeden element po określonej ilości, następnie iterujesz drugi element.

Np.

[[(i, j) for j in range(10)] for i in range (3)]

**Tabliczka mnożenia** → za pomocą list comprehension można mnożyć w różnych

#konfiguracjach dwa lub więcej obiekty przez siebie w pętli.

Np.

[[i \* j for j in range(10)] for i in range (3)]

**List comprehension, a tekst** → zwróć uwagę, że kolejność co jest iterowane jako pierwsze nie ma znaczenia. Znaczenie ma działanie, tj. obliczenie.

Np.

[[l1 + l2 for l2 in 'abcde'] for l1 in '12345']

# %%

**Funkcja rekurencyjna** → na przykładzie wyliczania silni. Silnia to iloczyn wszystkich liczb naturalnych dodatnich nie większych niż n. Patrz wikipedia. Funkcja rekurencyjna to funkcja która wywołuje samą siebie. Po przygotowaniu wzoru wywołaj silnię za pomocą list comprehension.

Np.

def silnia(n):

    if n == 0 or n == 1:

        return 1

    else:

        return n \* silnia(n - 1)

[silnia(i) for i in range(15)]

**Print w list comprehension** → możesz wyprintować całe zadanie list comprehention w standardowy sposób.

Np.

print([t for t in range(30) if t % 4 == 0])

**Wyciąganie określonych elementów słownika z list comprehension** → jak na przykładzie poniżej, jako obiekt iterujący zaznacz, że chodzi o konkretny element już na początku.

Np.

products = [

    {'name': 'T-shirt', 'price': 20, 'quantity': 10},

    {'name': 'Jeans', 'price': 50, 'quantity': 0},

    {'name': 'Sneakers', 'price': 80, 'quantity': 5},

    {'name': 'Hat', 'price': 15, 'quantity': 3},

    {'name': 'Backpack', 'price': 30, 'quantity': 7}

]

in\_stock = [s['name'] for s in products if s['quantity'] > 0]

print(in\_stock)

**Lekcja\_33\_Dict comprehention**

**Dict comprehension** → analogia dla list comprehension, ale dla słowników. Wzór dla dict comprehension: zdefiniuj zmienną. W nawiasie kwadratowym zmapuj klucze z wartościami za pomocą dwukropka jak w słowniku for to skąd bierzesz dane czyli klucze, słowniki w źródło.items() bo bierzesz wszystkie obiekty ze słownika.

Np.

stocks = {'AMZN.US': 'Amazon.com Inc', 'GOOGL.US': 'Alphabet Inc',

          'AAPL.US': 'Apple Inc', 'UBER.US': 'Uber Technologies Inc',

          'MSFT.US': 'Microsoft Corp'}

#Zwykły sposób:

for key, value in stocks.items():

    print('{:8}: {}'.format(key, value))

#Sposób dict comprehension:

stocks\_dict = {key:value for (key,value) in stocks.items()}

print(stocks\_dict)

**Wyrównywanie tekstu do określonego miejsca** → jeśli chcę wyrównać tekst do określonego miejsca, np. długości najdłuższego obiektu, to w nawiasach kwadratowych, gdzie znajdą się wartości za pomocą metody format mogę umieścić np. {:8} jeśli chcę wyrównać do 8 miejsca.

Np.

stocks = {'AMZN.US': 'Amazon.com Inc', 'GOOGL.US': 'Alphabet Inc',

          'AAPL.US': 'Apple Inc', 'UBER.US': 'Uber Technologies Inc',

          'MSFT.US': 'Microsoft Corp'}

for key, value in stocks.items():

    print('{:8}: {}'.format(key, value))

# %%

stocks\_dict = {key:value for (key, value) in stocks.items()}

print(stocks\_dict)

**Zbiór dict comprehension** → na przykładzie definiowanie zbioru tupli, nie  mylić ze słownikiem.

Np.

stocks\_set = {(key, value) for (key, value) in stocks.items()}

**Odwracanie wartości w słowniku** → jeśli chcę odwrócić klucze z wartościami miejscem mogę użyć do tego dict comprehension. W przypisaniu nowej zmiennej i iteracji po prostu odwróć kolejność i przypisuj klucze wartościom, a  wartości kluczom. Pamiętaj jednak, że nie o ile oryginalnie klucze są różne to wartości nie, więc przy zamianie może się okazać, że "nowe" klucze mają takie same wartości.

Np.

stocks\_inverted = {value : key for (key, value) in stocks.items()}

**Metody dla dict comprehension** → na przykładzie metody lower widać, że można korzystać z metod w dict comprehension dla kluczy/wartości.

Np.

{key.lower():value.lower() for (key, value) in stocks.items()}

**Funkcje dla dict comprehension** → na przykładzie funkcji lower widać, że można korzystać z funkcji w dict comprehension dla kluczy/wartości.

Np.

key\_length\_name = {k: v + ':' + str(len(k)) for (k, v) in stocks.items()}

**Dict comprehension i własne funkcje** → na przykładzie pokazana implementacja

#własnej funkcji dla dict comprehension.

Np.

def replace\_corp\_inc(name):

    name = name.replace('Corp', '0')

    name = name.replace('Inc', '1')

    return name

stocks\_flag = {k:replace\_corp\_inc(v) for (k, v) in stocks.items()}

**Warunek if w dict comprehension** → na końcu zapisu podaj warunek if. Czyli brane będą poprzednie zdarzenia pod uwagę jeśli spełniony zostanie warunek.

Np.

#Jeden warunek

stocks\_if = {key:val for (key,val) in stocks.items() if 'Inc' in val}

#Dwa warunki

stocks\_if\_if = {key:val for (key,val) in stocks.items()\

                if val.startswith('A') if len(val) < 13}

**Przełamywanie treści kodu** → za pomocą backslash "\".

Np.

stocks\_if\_if = {key:val for (key,val) in stocks.items()\

                if val.startswith('A') if len(val) < 13}

**Instrukcje warunkowe w dict comprehension** → jest możliwe stosowanie instrukcji warunkowych w dict comprehension. Wzór: {instrukcja if warunek else inna instrukcja for (klucz, wartość) in zmienna.items()}

Np.

stocks = {'AMZN.US': 'Amazon.com Inc', 'GOOGL.US': 'Alphabet Inc',

          'AAPL.US': 'Apple Inc', 'UBER.US': 'Uber Technologies Inc',

          'MSFT.US': 'Microsoft Corp'}

{key:'Corporation' if 'Corp' in val else 'Incorporated' for (key,val)\

 in stocks.items()}

**Zagnieżdżone słowniki w dict comprehension** → poprzez odwoływanie się do określonych elementów słownika. Poniższy słownik to słowniki zagnieżdżone w słowniku.

Np.

nested\_dict = {'001': {'price': 100, 'items': 4},

               '002': {'price': 40, 'items': 2},

               '003': {'price': 60, 'items': 10}}

price\_times\_items = {k:v['price'] \* v['items'] for (k,v)\

                     in nested\_dict.items()}

**Dict comprehension, a lista** → z listy można w prosty sposób stworzyć słownik za pomocą dict comprehension.

Np.

languages = [

    ('Python', '.py'),

    ('JavaScript', '.js'),

    ('C++', '.cpp'),

    ('Java', '.java'),

]

languages\_II = {k:v for v, k in languages}

**Funkcje w dict comprehension** → możesz korzystać z funkcji w dict comprehension.

Np.

soldiers = [

    {'name': 'Alice', 'rank': 'Private', 'service\_years': 2},

    {'name': 'Bob', 'rank': 'Sergeant', 'service\_years': 4},

    {'name': 'Charlie', 'rank': 'Sergeant', 'service\_years': 6},

    {'name': 'David', 'rank': 'Lieutenant', 'service\_years': 3},

    {'name': 'Eve', 'rank': 'Private', 'service\_years': 1},

    {'name': 'Frank', 'rank': 'Lieutenant', 'service\_years': 7},

]

rank\_dict = {soldier['rank']:sum(1 for s in soldiers if s['rank'] == soldier['rank']) \

             for soldier in soldiers}

print(rank\_dict)

**Kolejne poziomy reguły for w dict comprehension** → omówienie poniższego  przykładu. Chcę tworzyć nowy słownik, który odwzorowuje każdy unikalny atom  we wszystkich związkach na całkowitą liczbę wystąpień we wszystkich związkach. Aby tego dokonać definiuję chwilowo atom:suma wszystkich wystąpień atomu jeśli ten znajduje się w elemencie będący iterowany w danym momencie. Dalej: ten atom i suma pojawią się dla elementu słownika, który pojawi się dla w elemencie elementu['atoms'].

compounds = [

    {

        'name': 'Water',

        'atoms': ['H', 'O', 'H'],

    },

    {

        'name': 'Methane',

        'atoms': ['C', 'H', 'H', 'H', 'H'],

    },

    {

        'name': 'Ethanol',

        'atoms': ['C', 'H', 'O', 'H', 'H', 'C', 'H', 'H', 'H'],

    },

]

atom\_dict = {

    atom: sum(1 for c in compounds if atom in c['atoms'])

    for compound in compounds

    for atom in compound['atoms']

}

print(atom\_dict)

**Dict comprehension, ogólne wyjaśnienie** → Poniższy przykład wymagał  stworzenia słownika, który odwzorowuje  każdą pozycję na całkowitą liczbę bramek zdobytych przez graczy na tej.pozycji. Patrz od początku: chodzi o zdefiniowanie słownika poprzez iterację. W słowniku może być tylko jeden taki sam klucz, a więc można zdefiniować go bez problemu nie patrząc na powtarzalność przy iteracji. Wartościami są sumy goli, ale to musi być inna nazwa tymczasowej zmiennej, nie "player" tylko  np. "p". Reguła jeżeli sprawdza czy nowy klucz i klucz iterowanego elementu są zbieżne. Wszystko to dzieje się dla (for) player in players.

players = [

    {'name': 'Alice', 'position': 'Forward', 'goals': 15},

    {'name': 'Bob', 'position': 'Midfielder', 'goals': 7},

    {'name': 'Charlie', 'position': 'Defender', 'goals': 2},

    {'name': 'David', 'position': 'Goalkeeper', 'goals': 0},

    {'name': 'Eve', 'position': 'Forward', 'goals': 12},

    {'name': 'Frank', 'position': 'Midfielder', 'goals': 5},

]

position\_dict = {

    player['position']: sum(

        p['goals']

        for p in players

        if p['position'] == player['position']

    )

    for player in players

}

print(position\_dict)

**Dict comprehension, a tylko jedna wartosć** → jak widać no poniższym  przykładzie iteracja odbywała się za pomocą jednej wartości, która została potraktowana w różny sposób.

Np.

slownik = {i: i \*\* 2 for i in range(5)}

print(slownik)

**Instrukcja warunkowa dict comprehension dla stworzenia nowego słownika, który zawiera tylko elementy z oryginalnego** → tworzenie słownika nowego na podstawie starego należy przeprowadzić za pomocą metody items(). Za pomocą tej metody wyciągasz wszystkie elementy ze słownika, które możesz poddać instrukcji warunkowej.

Np.

slownik = {'jeden':5, 'dwa': 6, 'trzy': 7, 'cztery': 8}

nowy\_slownik = {k:v for (k, v) in slownik.items() if v > 5}

**Dict comprehension, a tworzenie słownika na podstawie dwóch list, w którym elementy jednej listy są kluczami, a elementy drugiej listy są wartościami** → tutaj podaj jako klucze i wartości po prostu nazwy tych zmiennych (list) oraz w nawiasie kwadratowym iteratory, po czym iteruj po nich zaznaczając, że robisz to tyle razy ile jest elementów w liście z kluczami (range + len).

Np.

lista\_1 = [1,2,3,4,5,6]

lista\_2 = ['jeden', 'dwa', 'trzy', 'cztery', 'pięć', 'szesć']

slownik = {lista\_1[i]: lista\_2[i] for i in range(len(lista\_1))}

**Lekcja\_34\_Set comprehention**

**PyInstaller** → program służący do konwertowania kodu, skryptu na tekstowy program. Sposób instalacji:

Otwórz wiersz poleceń lub terminal (znak Windows + R, a potem cmd) i uruchom następujące polecenie, aby zainstalować PyInstaller:

pip install pyinstaller

Użyj polecenia `cd`, aby przejść do katalogu, w którym znajduje się skrypt Pythona.

​cd path\to\your\script

Uruchom PyInstaller za pomocą następującego polecenia:

pyinstaller --onefile your\_script.py

Po zakończeniu pracy PyInstaller w katalogu skryptu znajdziesz katalog „dist”. W katalogu `dist` zobaczysz samodzielny plik wykonywalny o tej samej nazwie co Twój skrypt, ale z rozszerzeniem `.exe`.

**Set comprehension** → pamiętaj, że to zbiór, więc będzie zwracać unikalne wartości. Jeśli chcę przypisywać unikalne obiekty za pomocą iteracji do zbioru, mogę korzystać z kompresji zbioru. Wzór iteracji: wynik for obiekt\_iterowany in zmienna. Zwróć uwagę, że zbiory znajdują się w nawiasach klamrowych {}.

Np.

text = 'Python jest wspaniały. Python jest elastyczny. Python rządzi.'

words = text.lower().replace('.', '').split(' ')

unique\_words = {word for word in words}

**Set comprehension pod warunkami** → mogę chcieć dodawać określone warunki do kompresji, co należy zrobić dodając warunek if na końcu zapisu.

Np.

text = 'Python jest wspaniały. Python jest elastyczny. Python rządzi.'

words = text.lower().replace('.', '').split(' ')

unique\_words\_grater\_than\_4 = {word for word in words if len(word) > 4}

**Set comprehension i modyfikacja zwracanych obiektów** → jeśli chcę żeby obiekt iterowany zwracany był w określony sposób mogę swoje żądania  podać na początku zapisu.

Np.

text = 'Python jest wspaniały. Python jest elastyczny. Python rządzi.'

words = text.lower().replace('.', '').split(' ')

capitalized\_python = {word.capitalize() if word.startswith('pyt') else word

                      for word in words}

**Set comprehension i wyciąganie elementu słownika zagnieżdżonego w liście słowników** → chodzi o zasadę, że jeśli wyciągasz coś co jest zagnieżdżone to zaczynasz od najbardziej ogólnego, największego obiektu (w przykładzie jest tym obiektem zmienna courses) po czym odwołujesz się do kolejnych, mniejszych. Jak na przykładzie: wyciągasz tag dla kursu w kursach, dla tagu znajdującego się elemencie tags słownika. Pierwsza część kodu zwróci w związku z tym tag określony, czyli dany kurs i potem dla tego tag następna część kodu wyciąga elementy zagnieżdżone niżej czyli course[‘tags’]

Np.

courses = [

    {

        'id': 1,

        'name': 'Python for Beginners',

        'tags': ['Python', 'Programming', 'Beginner']

    },

    {

        'id': 2,

        'name': 'Data Science with Python',

        'tags': ['Python', 'Data Science', 'Statistics']

    },

    {

        'id': 3,

        'name': 'Machine Learning',

        'tags': ['Python', 'Machine Learning']

    }

]

unique\_tags = {tag for course in courses for tag in course['tags']}

**SEKCJA 8. MODUŁY I PAKIETY**

**Lekcja\_31\_Moduły**

**Moduły w Python** → moduł to rozszerzenie w Pythonie .py, czyli standardowy plik pythonowski, w którym zawarto zestaw funkcji, które można z tego modułu zaimportować metodą import i korzystać w skrypcie. Pamiętaj, że importując moduł, importujesz wszystkie funkcje, które się w nim znajdują. Moduły w założeniu ułatwiają zarządzanie dużymi projektami poprzez podział kodu na mniejsze, niezależne części.

**Import** → komenda ładująca moduł. Dwie sprawy. Jeśli w kodzie są dwie komendy import, to Python bierze pod uwagę tylko ten pierwszy, a nie drugi. Druga sprawa: jak zaimportujesz moduł, to zanim zacznie od razu działać, musisz zrestartować jądro. Wzór: import nazwa\_pliku\_modułowego

Np.

import rocket\_science

**Ważne dwie sprawy o importowaniu/wykorzystywaniu modułów** → pierwsza sprawa: ścieżkę modułu importowanego/używanej funkcji z tego modułu podajesz tak, że patrzysz na główny plik z którego Python pobiera pliki i jeśli coś jest tam w folderach, to podajesz ich ścieżkę po kropce. Druga sprawa: kiedy przywołujesz w kodzie nazwy plików czy folderów, bierz pod uwagę WIELKIE LITERY.

Np.

# -\*- coding: utf-8 -\*-

from Moduly.string\_operations import count\_vowels

print(count\_vowels('Python'))

print(count\_vowels('Hello Pythonista!'))

**Tworzenie modułu** → najpierw stwórz najlepiej folder gdzie będziesz trzymać moduły. Stwórz potem prosty plik Python-owski. Stwórz parę przykładowych funkcji w typu pliku. Stwórz kolejny plik, w którym dokonasz sprawdzenia czy moduł działa. Tam za pomocą funkcji import zaimportuj moduł i za pomocą funkcji dir sprawdź czy faktycznie znalazły się wśród możliwych do  wykorzystania nowe funkcje.

Np.

import rocket\_science

# %%

dir(rocket\_science)

**Wykorzystywanie funkcji z modułów** → aby wywoływać funkcje, które pochodzą z modułów, należy to zrobić następującym wzorem: nazwa\_pliku\_modułowego.funkcja()

Np.

import rocket\_science

# %%

dir(rocket\_science)

# %%

rocket\_science.licz\_srednia([3,4])

**From** → Importowanie tylko poszczególnych funkcji z modułu. Jeśli nie chcesz importować wszystkiego, tylko określone funkcje, skorzystasz z instrukcji from. Wzór: from nazwa\_pliku\_modułowego import funkcja. Dzięki temu dodatkowo, przywołując w kodzie tą funkcję nie musisz podawać nazwy pliku modułowego, tylko od razu nazwę funkcji. Jest to opcja rekomendowana, zwłaszcza gdy wiesz, że chcesz skorzystać tylko z tej jednej funkcji. Poza tym jak ktoś spojrzy na mój kod to zobaczy co robiłem i co chciałem osiągnąć.

Np.

from rocket\_science import licz\_srednia

licz\_srednia([3,4])

**Półśrodek między import, a from** → możesz podać w from ... import \*.  Gwiazdka oznacza tu wszystko i daje to tyle, że nie masz wszystkie funkcje, ale jednocześnie nie musisz pisać nazwy pliku modułowego przywołując funkcje. Aczkolwiek nie jest to rekomendowany sposób.

Np.

from rocket\_science import \*

licz\_srednia([3,4])

Importowanie wielu funkcji --> posługując się rekomendowanym sposobem  przywoływania funkcji, ale chcąc improtować więcej niż jedną, mogę to zrobić po prostu po przecinku.

Np.

from rocket\_science import licz\_srednia, licz\_maximum

sred = licz\_srednia([3,4])

maxi = licz\_maximum([3,4])

**Folder \_\_pycache\_\_** → jest to folder, który powstaje automatycznie przy operacji importowania w miejscu z którego importuję dane. Zawiera on plik typu .cpython, który zawiera importowaną treść, co ma pozwalać Python-owi na szybszą pracę.

**Lekcja\_32\_Pakiety**

**Pakiety** → inaczej nazywane bibliotekami. To przestrzenie nazw, które zawierają wiele modułów. Teoretycznie pakiety mogą zawierać inne pakiety. Są to foldery z plikami Pythonowymi. Każdy pakiet jest folderem i musi on zawierać plik o nazwie \_\_inid\_\_.py - ten plik może być pusty i służy poinformowaniu, że ten folder zawiera pakiet Python i może być zaimportowany tak jak moduły.

**Tworzenie nowego pakietu** → eksplorator plików, prawym na folder gdzie  trzymasz moduły/pakiety, nowy, pakiet, nazwij go (np. "rocket"). Powstał plik init. Możesz zapisywać teraz w tym folderze moduły.

**Wywoływanie funkcji z pakietu** → importujesz funkcję wskazując na plik. Po kropce wskazujesz na adres pliku. Następnie tą samą ścieżką na koniec po kropce dodajesz nazwę interesującej cię funkcji. Inaczej: piszesz nazwę pakietu, nazwę modułu i funkcję. Pamiętaj, że musisz zaimportować najpierw daną bibliotekę. Jeśli nie działa to dodaj na początku nazwę folderu, w którym znajduje się pakiet.

Np.

import Moduły\_Python.rocket.data

# %%

dir(Moduły\_Python.rocket.data)

# %%

dane = Moduły\_Python.rocket.data.get\_data()

# %%

import Moduły\_Python.rocket.algorytmy

# %%

dir(Moduły\_Python.rocket.algorytmy)

Moduły\_Python.rocket.algorytmy.drzewa\_decyzyjne()

**Szybsze importowanie wybranych funkcji** → jeśli chcę wyciągnąć daną funkcję, a nie całą bibliotekę, mogę to zrobić za pomocą polecenia from, tak jak w przypadku modułów. from pakiet.moduł import funkcja. Jeśli nie działa to dodaj na początku nazwę folderu, w którym znajduje się pakiet.

Np.

from Moduły\_Python.rocket.algorytmy import drzewa\_decyzyjne

# %%

drzewa\_decyzyjne()

**Podpakiety i korzystanie z funkcji z różnych modułów** → W pakietach można tworzyć podpakiety, które wywyłowane są na tej samej zasadzie importu i  podawania ścieżki, ale ich funkcją jest oddzielanie niektórych danych i funkcji od reszty. Na poniższym przykładzie zatem podajesz ścieżkę: nazwa folderu.pakiet.moduł.podpakiet import funkcja.

Np.

import Moduły\_Python.rocket.data

# %%

dane = Moduły\_Python.rocket.data.get\_data()

# %%

from Moduły\_Python.rocket.funkcje.stats import mean

srednia\_dane = mean(dane)

**Wywoływanie pakietów** → pakiet można wywołać jeśli jest się w katalogu  roboczym w którym taki pakiet się znajduje. Jeśli chcesz żeby pakiet działał niezależnie od jego lokalizacji (za pomocą komendy pwd w konsoli wywołasz lokalizację plików) trzeba go zainstalować w środowisku python w którym działa. Autor kursu podaje to w innych lekcjach.

**Komentarz do narzędzia szyfrującego →** poniższy przykład pochodzi z ćwiczenia po lekcji o pakietach i był interesujący. Zwróć uwagę na parę rzeczy: po pierwsze lower jest metodą obniżającą literkę do małej, po drugie isupper sprawdza czy litera jest duża, po trzecie if nie potrzebuje else, po prostu nic się nie stanie jeśli warunek nie zostanie spełniony, a po czwarte kod nie zawiera osobnych instrukcji co jeśli będzie literka mała, a co jeśli duża - ktoś po prostu zrobił test dla literki zmniejszonej (metoda lower) nieważne jaka była, a potem wstawił dodatkowy warunek jeśli pierwotna literka była duża.

Np.

def encrypt(message, key):

    alphabet = "abcdefghijklmnopqrstuvwxyz"

    encrypted\_message = ""

    for char in message:

        if char.lower() in alphabet:

            char\_index = alphabet.index(char.lower())

            encrypted\_char = key[char\_index]

            if char.isupper():

                encrypted\_char = encrypted\_char.upper()

            encrypted\_message += encrypted\_char

        else:

            encrypted\_message += char

    return encrypted\_message

def decrypt(message, key):

    alphabet = "abcdefghijklmnopqrstuvwxyz"

    decrypted\_message = ""

    for char in message:

        if char.lower() in alphabet:

            char\_index = key.index(char.lower())

            decrypted\_char = alphabet[char\_index]

            if char.isupper():

                decrypted\_char = decrypted\_char.upper()

            decrypted\_message += decrypted\_char

        else:

            decrypted\_message += char

    return decrypted\_message

**Polecenie służące do instalacji pakietów zewnętrznych** → w języku Python jest to polecenie pip install nazwa\_pakietu

**Polecenie służące do usuwania pakietów zewnętrznych** → w języku Python jest to polecenie pip remove nazwa\_pakietu

**Polecenie służące do wyświetlania pakietów zewnętrznych** → w języku Python jest to polecenie pip list

**SEKCJA 9. WBUDOWANE MODUŁY I PAKIETY**

**Lekcja\_33\_Pakiet os**

**Pakiet os** → pakiet pozwalający na wykonywanie podstawowych czynności związanych z systemem operacyjnym.

Np.

import os

# %%

dir(os)

**Os.getcwd** → funkcja os, Get Working Directory. Ona zwraca ścieżkę do aktualnego katalogu roboczego.

Np.

# %%

os.getcwd()

**Os.chdir** → funkcja przenosząca katalog roboczy. Jedna kropka oznacza dotychczasowy katalog roboczy, dwie przenoszą go stopień wyżej (czyli do folderu gdzie on wcześniej się znajdował). Możesz też podać pełną nazwę ścieżki.

Np.

# %%

os.chdir('C:\\Users\\Leszek')

# %%

os.chdir('..')

# %%

os.getcwd()

**Os.system** → funkcja pozwalająca przekazać unixowe polecenia. Możesz np. wykorzystać tu komendę mkdir, która tworzy katalogi - następnie po spacji będą nazwy tych katalogów. Katalogi czyli foldery. Weź pod uwagę, że możesz szybko i prosto zautomatyzować tworzenie katalogów (folderów) dzięki Python w ten sposób.

Np.

os.system('mkdir dir1 dir2 dir3')

**Os.rmdir** → funkcja usuwająca katalog (folder). W argumencie podaj nazwę katalogu.

Np.

os.system('mkdir test')

# %%

os.rmdir('test')

**Os.listdir** → metoda podaje listę plików w katalogu. Docelowo poda listę plików z katalogu (pliku) roboczego. Możesz w argumencie podać ścieżkę do katalogu, którego pliki chcesz wylistować. Możesz też alternatywnie w argumencie podać kropkę ('.') co oznacza bieżący katalog.

Np.

os.listdir()

# %%

os.listdir('C:\\Users\\Leszek\\Dysk Google')

**Wykorzystanie pakietu os do różnych operacji, np. iterowania** → jak na przykładzie, możesz wplatać te funkcje z pakietu os do pracy w Python.

Np.

for file in os.listdir('C:\\Users\\Leszek\\Dysk Google'):

if file.endswith('0'):

     print(file)

**Os.walk** → funkcja pozwala na przechodzenie po katalogach. Metoda walk zwraca katalog podany w ścieżce, która zostaje podana. Na przykładzie pętli. Metoda Pythona walk() generuje nazwy plików w drzewie katalogów, przechodząc po drzewie z góry lub z dołu do góry. Wykorzystuj z pętlą - pętla ta jak widać będzie printować podany katalog (root), katalogi (dirs) i pliki (files). Interpretacja wyniku: pokazuje katalog ze ścieżki, potem w formie listy pierwszy folder i w formie listy zawartość folderu, po czym następuje kolejna iteracja. Jeśli lista jest pusta to dlatego, że nie było katalogu lub plików w katalogu. Zwróć uwagę, że jeśli musi wejść na kolejne poziomy, to iteracje zmieniają ścieżki katalogów, na te położone głębiej.

Np.

for root, dirs, files in os.walk('C:\\Users\\Leszek\\Moduly'):

print(root)

print(dirs)

print(files)

**Os.path.join** → funkcja, którą trzeba pobrać z modułu "path". Join łączy nazwy ścieżek. W argumencie przed tekstem łączonym możesz dodać "r" co oznacza "raw tekst" czyli surowy tekst - gdyby tego nie było to bylibyśmy narażeni na błędy takie jak białe znaki. W argumencie po przecinkach podajesz ścieżki, które chcesz połączyć i one mogą mieć jeden backslash, bo funkcja ta poprawi na dwa. Pamiętaj tylko nie podawać prawidłowo dwóch baskshashy bo on to zrozumie jakby nic nie było między nimi i zrobi z nich cztery.

Np.

os.path.join(r'home\qwqe\\sddasdasd', 'python', 'leszek')

**Lekcja\_34\_Pakiet sys: Uruchamianie skryptów z poziomu PowerShell’a i Bash’a\***

\*Końcówka lekcji dotyczyła Bash’a, z którego korzysta się korzystając z Unix-a, czyli np. Linux. Na ten moment to mnie nie interesuje.

**Pakiet sys** → Uruchamianie skryptów z poziomu PowerShell’a.

**Wykonywanie skryptów python z poziomu wiersza poleceń** → albo inaczej z poziomu powłoki albo terminala.

**Otwieranie okna PowerShell** → możesz to zrobić w dowolnym folderze, ale  dla przykładu otwórz z poziomu eksploratora plików Spyder, który folder poprzez prawy i "show in folder". Następnie przyciskając shift i prawym, włącz "Otwórz w terminalu".

**Przykładowe komendy PowerShell** → pwd (zwraca ścieżkę do pliku), ls (zwraca pliki w folderze + niektóre dane), cd [spacja] nazwa\_pliku (przy czym możesz zacząć pisać nazwę pliku i za pomocą tab resztę uzupełnić - za pomocą tej komendy wejdziesz do pliku).

**Wykonywanie plików Python w PowerShell** → na przykładzie prostego printowania "Hello world!", tworzysz plik, który wykonuje taką prostą operację i wywołujesz go w PowerShell, po czym korzystasz z komendy: python [nazwa pliku].py.

**Dostęp do interpretera pythona z poziomu PowerShell** → jeśli chcesz pisać bezpośrednio komendy w języku python z poziomu PowerShell to skorzystaj z komendy python. Teraz możesz pisać komendy tego języka w PowerShell. Jest to w tej chwili to samo co konsola w Spyder, możesz też używać go jako kalkulatora.

**Praktyczne wykorzystanie PowerShell** → powiedzmy, że mam dostęp do serwera postawionego na Linux i mogę mieć dostęp do niego i robić na nim operacje tylko przez terminal. Jedyne co to jeśli masz zainstalowaną więcej niż jedną wersję pythona to w komendach piszesz python1 albo python2 albo python3.

**Wychodzenie z interpretera python** → za pomocą komendy exit().

**Lekcja\_35\_Pakiet sys: Specjalna zmienna \_\_name\_\_**

**\_\_name\_\_** → przy uruchomieniu skryptu w Python inicjalizowana specjalne zmienne i jedną z tych zmiennych jest zmienna \_\_name\_\_, która przechowuje nazwę modułu. Przy bezpośrednim uruchomieniu skryptu moduł ten przyjmuje nazwę '\_\_main\_\_'. Możesz to sprawdzić wpisując do konsoli po prostu \_\_name\_\_. Zrób też inny test: stwórz nowy folder i w nim dwa moduły. Jeden niech printuje nazwę modułu, a drugi niech tylko importuje pierwszy. W takim  przypadku modułem dla second będzie first. Sens sprawdzania czy \_\_name\_\_ jest równe \_\_main\_\_ jest taka, że sprawdzasz czy plik jest uruchamiany bezpośrednio czy przez import. Sprawdzane jest to po to, że jak piszesz testy jednostkowe albo jak chcesz wykonać blok kodu przy pomocy skryptów Python-owych.

Np.

if \_\_name\_\_ == '\_\_main\_\_':

    print('Uruchomiony bezpośrednio')

else:

    print('Uruchomiony z importu')

**Lekcja\_36\_Pakiet sys: Uruchamianie skryptów z argumentami**

**Pisanie poleceń w konsoli Spyder jak w konsoli PowerShell** → możesz pisać tak samo, tylko na początku wyboru języka, czyli Python wpisz ! a po spacji pisz polecenia.

**Sys.argv** → polecenie, które zwraca w pierwszej kolejności nazwę pliku w którym znajduje się skrypt (czyli plik Python, .py) a w drugiej kolejności argumenty, które znajdują się w tym skrypcie. Wszystko jest podawane w formie listy. Załóżmy, że plik ze skryptem nazywa się "pakiet\_sys.py". Jak widzisz w poniższym kodzie, czyli skrypcie, który zostanie uruchomiony w konsoli, importuje on pakiet sys, zwraca nazwę pliku, gdyż pierwszym co zwraca sys.argv to właśnie nazwa pliku, po czym za pomocą pętli while w każdej kolejnej inkrementacji zwraca argumenty. Argumenty te natomiast podawane są pisząc np. "!python pakiet\_sys.py 15 2 1344343 13413423 'aadssd'". Piszesz zatem !python nazwa\_skryptu(czyli pliku) po spacji argumenty.

Np.

import sys

print(sys.argv[0])

args = sys.argv

i = 0

while args:

    print('Argument nr {}: {}'.format(i, args.pop(0)))

    i += 1

**Wykorzystanie sys.argv w PowerShell** → musisz mieć przygotowany skrypt, czyli plik python z kodem, który chcesz uruchomić. Pamiętaj, że w  argumenty podajesz po spacji od podania nazwy skryptu.

Np.

PS C:\Users\Leszek> python pakiet\_sys.py 90 99 hello world

**Przekazywanie do terminala danych, uruchamianie bezpośrednio i dane liczbowe**

→ jeśli tak jak na przykładzie chcesz podać w konsoli PowerShell jeden argument w ramach skryptu zapisanego w Python, skorzystaj z sys.argv i pamiętaj podać [1], bo [0] zwróci nazwę pliku. Ponadto zwróć uwagę, że dane w PowerShell zawsze będą z założenia tekstem, nawet liczby, a zatem musisz je przekonwertować najpierw na int. Ostatnia rzecz, chcesz żeby skrypt wykonywał się wtedy kiedy plik był uruchamiany bezposrednio.

Np.

import sys

def silnia(n):

    if n == 0 or n == 1:

        return 1

    else:

        return n \* silnia(n - 1)

if \_\_name\_\_ == '\_\_main\_\_':

    print(silnia(int(sys.argv[1])))

PS C:\Users\Leszek> python silnia.py 5

**Sys.version** → Skrypt przekazujący informacje o wersji języka Python. Ten kod jest bardzo przydatny do pracy z językiem Python. Pozwala nam dostać informacje na temat wersji języka Python. Po uruchomieniu tego skryptu zostanie wyświetlona używana wersja języka.

Np.

import sys

python\_version = sys.version

print('Python version:', python\_version)

**Sys.stdout** → W tym przykładzie najpierw importujemy bibliotekę sys. Następnie tworzymy i otwieramy (chyba, że już istnieje) plik o nazwie output.txt do zapisu przy użyciu funkcji open() i instrukcji with. Ten plik zostanie użyty do przekierowania standardowego wyjścia. Następnie używamy zmiennej sys.stdout, aby przekierować standardowe wyjście do pliku. Oznacza to, że wszelkie dane wyjściowe drukowane do konsoli w bloku kodu with... zostaną teraz przekierowane do pliku. Następnie drukujemy dane wyjściowe za pomocą funkcji print(), która zostanie teraz zapisana w pliku zamiast w konsoli. Na koniec poza blokiem kodu with... standardowe wyjście wraca do konsoli.Po uruchomieniu tego skryptu w aktualnym katalogu roboczym zostanie utworzony plik output.txt z podaną zawartością: "This output will be written to a file". Ten kod jest przydatny na przykład do generowania logów do pliku.

Np.

import sys

with open('output.txt', 'w') as file:

    sys.stdout = file

    print('This output will be written to a file')

print('This output will be printed to the console')

**SEKCJA 10. PROGRAMOWANIE OBIEKTOWE (OBJECT-ORIENTED PROGRAMMING)**

**Lekcja\_37\_Wprowadzenie do programowania obiektowego (klasa, obiekt, atrybut, metoda)**

**Programowanie obiektowe** → wszystko w Python jest obiektem. Inaczej object oriented programming. To paradygmat programowania, w którym programy definiujemy za pomocą obiektów. Obiekty tworzone są za pomocą specjalnie  zdefiniowanych klas. Przykład: tworzę program, w którym uzupełniane są  dane o pracownikach. Klasą jest tu pracownik, a obiektem będzie konkretny pracownik. Abstrakcją jest klasa, która przedstawia typowego pracownika. Atrybuty/pola oznaczają w tym przykładzie właściwości takie jak staż pracy, wynagrodzenie, itd.

**class** → za pomocą tej instrukcji definiuje się klasy. Konwencja jest taka, że nazwy klas pisane są z dużej litery, nawet jeśli klasa ma więcej niż jedno słowo, to nadal każde jest z dużej. Wzór: class Klasa:. enter, cztery spacje wcięcia i atrybuty, nazwa = 'nazwa', enter bez wcięcia pisz nazwę obiektu z małej litery = Nazwa Klasy()

**id, a obiekt** → za pomocą funkcji id można wyświetlić identyfikator danego obiektu.

**dir, a obiekt** → za pomocą dir zobacz jakie atrybuty ma obiekt.

**Metoda kropka** → za pomocą tej metody można dostać się do atrybutów. Wzór: nazwa\_obiektu.nazwa\_atrybutu. To zwraca wartości przypisane obiektom.

**Atrybut klasy, a atrybut instancji** → atrybuty klasy to atrybuty, które  zostały założone przy tworzeniu klasy. Atrybuty instancji tworzy się w locie.

**Dodawanie atrybutów instancji** → wzór: nazwa\_obiektu.nazwa\_nowego\_atrybutu.

**Dodawanie atrybutów dla klasy** → jeśli nie chcę dodawać atrybutu tylko do jednego obiektu, jak w przypadku instancji, mogę to zrobić dla wszystkich atrybutów naraz. Wzór: Klasa.nowy\_atrybut = 'wartosc\_atrybutu'

**Zmiana wartości atrybutów** → nadpisując, tzn. jakbyś chciał na nowo pisać wartość.

**Instancja** → instancja to obiekt danej klasy, czyli np. dla klasy Drzewo instancją jest np. drzewo\_1.

Przykład to całego wykładu:

Np.

class Drzewo:

    nazwa = 'Sosna'

    wiek = 40

    wysokosc = 25

drzewo\_1 = Drzewo()

drzewo\_2 = Drzewo()

# %%

print(id(drzewo\_1))

print(id(drzewo\_2))

# %%

dir(drzewo\_1)

# %%

print(drzewo\_1.nazwa)

print(drzewo\_1.wiek)

print(drzewo\_1.wysokosc)

# %%

print(drzewo\_2.nazwa)

print(drzewo\_2.wiek)

# %%

drzewo\_1.stan = 'dobry'

print(dir(drzewo\_1))

# %%

Drzewo.miejsce = 'las'

dir(drzewo\_2)

# %%

drzewo\_2.miejsce = 'park'

dir(drzewo\_2)

**Lekcja\_38\_Specjalny parametr self**

**Metody i self** → funkcje zawarte w klasach to metody. Przy definiowaniu takiej funkcji zawartej w klasie, należy wpisać parametr self w nawiasie w odróżnieniu od definicji normalnej funkcji. Wykonywanie metody na obiekcie: odwoływanie się do metody obiektu przez nazwę i po kropce podać metodę. Są dwa sposoby zapisu, oba równoważne, ale korzystaj z tego krótszego: pierwszy zapis to obiekt.metoda(), a drugi, dłuższy to  Klasa.metoda(obiekt). Do obiektu wewnątrz Klasy należy odwoływać się przez słówko self, które jest reprezentantem ogółu.

Np.

class Drzewo:

    def wyswietl\_info\_o\_drzewie(self):

        self.nazwa = 'Sosna'

        self.wiek = 30

        print(f'Drzewo {self.nazwa} ma {self.wiek} lat.')

drzewo = Drzewo()

# %%

drzewo.wyswietl\_info\_o\_drzewie()

# %%

Drzewo.wyswietl\_info\_o\_drzewie(drzewo)

**pass** → instrukcja, która wymusza zignorowanie danego bloku kodu. Jeżeli tworzę szkielet swojej klasy, mogę skorzystać z pass i on nic nie  zrobi.

Np.

class Drzewo:

    def wyswietl\_info\_o\_drzewie(self):

        pass

drzewo = Drzewo()

drzewo.wyswietl\_info\_o\_drzewie()

**f-string, a self** → w nawiasach kwadratowych podajesz self.atrybut.

Np.

class Drzewo:

    def wyswietl\_info\_o\_drzewie(self):

        self.nazwa = 'Sosna'

        self.wiek = 30

        print(f'Drzewo {self.nazwa} ma {self.wiek} lat.')

drzewo = Drzewo()

**Lekcja\_39\_Konstruktor i metody**

**\_\_init\_\_** → metoda, która zostaje uruchomiona przy okazji utworzenia obiektu. Konstruktor uruchamia się przy tworzeniu obiektu. Ma on specjalną  nazwę i jak każda metoda przyjmuje parametr self. Konstruktor pozwala przekazać atrybuty w momencie tworzenia obiektu, czyli każdy obiekt może mieć inną wartość atrybutu. \_\_init\_\_ to skrót od initialiser i tą nazwę zawsze trzeba tak podać. Wzór: def \_\_init\_\_(self, atrybut\_1, atrybut\_2...): enter self.atrybut\_1 = atrybut\_1. Sens korzystania z konstruktora polega na  tym, że definiujesz ogólny obiekt, self i nadajesz mu atrybuty, po czym tworzenie szczegółowych obiektów jest szybsze. A więc działa to tak, że przy wywoływaniu klasy zmuszony jestem do dodaniu potrzebnych parametrów i przypisania wartości zmiennym.

Np.

class Drzewo:

    def \_\_init\_\_(self, nazwa, wiek, wysokosc):

        self.nazwa = nazwa

        self.wiek = wiek

        self.wysokosc = wysokosc

# %%

drzewo\_1 = Drzewo('Sosna', 35, 25)

drzewo\_2 = Drzewo('Brzoza', 15, 18)

# %%

dir(drzewo\_1)

# %%

print(drzewo\_1.nazwa)

print(drzewo\_2.nazwa)

**Konwencja tworzenia klasy** → według standardów Python tworząc klasy  zostaw dwie linie puste pomiędzy blokami kodu i to samo jak kończysz klasę. Wewnątrz klasy zostaw jedną linię pustą i zacznij definiować metodę.

**Myślenie o self** → o self należy myśleć jak o uogólnieniu obiektu, który tworzysz, takim szablonie. Jeśli nie chcesz podawać parametrów do metody, to nadal musisz podawać słowo self.

**Definiowanie metody z warunkiem w ramach konstruktora** → możesz zdefiniować metodę, która będzie sprawdzać określone warunki i zwracać cos. Pamiętaj, że odwołujesz się do self pisząc zmienne i w f-string.

Np.

class Drzewo:

    def \_\_init\_\_(self, nazwa, wiek, wysokosc):

        self.nazwa = nazwa

        self.wiek = wiek

        self.wysokosc = wysokosc

    def czy\_chronione(self):

        if self.wiek >= 20 and self.wysokosc >= 20:

            print(f'Drzewo o nazwie {self.nazwa} jest pod ochroną')

        else:

            print(f'Drzewo o nazwie {self.nazwa} nie jest pod ochroną')

# %%

drzewo\_1 = Drzewo('Sosna', 35, 25)

drzewo\_2 = Drzewo('Brzoza', 15, 18)

# %%

drzewo\_1.czy\_chronione()

drzewo\_2.czy\_chronione()

**Zmiana wartości atrybutów w ramach konstruktora** → utwórz odpowiednią metodę która będzie odwoływać się do obiektów.

#Np.

class Drzewo:

    def \_\_init\_\_(self, nazwa, wiek, wysokosc):

        self.nazwa = nazwa

        self.wiek = wiek

        self.wysokosc = wysokosc

    def czy\_chronione(self):

        if self.wiek >= 20 and self.wysokosc >= 20:

            print(f'Drzewo o nazwie {self.nazwa} jest pod ochroną')

        else:

            print(f'Drzewo o nazwie {self.nazwa} nie jest pod ochroną')

    def postarz\_o\_rok(self):

        self.wiek += 1

# %%

drzewo\_1 = Drzewo('Sosna', 35, 25)

drzewo\_2 = Drzewo('Brzoza', 15, 18)

# %%

print(drzewo\_1.wiek)

drzewo\_1.postarz\_o\_rok()

print(drzewo\_1.wiek)

**Prosta metoda aktualizacji atrybutów** → po prostu metoda definiująca nową zmienną, która staje się starą zmienną.

    class Product:

        def \_\_init\_\_(self, name, price, quantity):

            self.name = name

            self.price = price

            self.quantity = quantity

        def update\_price(self, new\_price):

            self.price = new\_price

        def update\_quantity(self, new\_quantity):

            self.quantity = new\_quantity

**Konstruktor i return** → w metodach jeśli tylko chcesz zwrócić jakieś  wyniki skorzystaj z return.

    class Product:

        def \_\_init\_\_(self, name, price, quantity):

            self.name = name

            self.price = price

            self.quantity = quantity

        def get\_total\_price(self):

            return self.price \* self.quantity

        def is\_in\_stock(self):

            return self.quantity > 0:

**Lekcja\_40\_Dekorator @staticmethod**

**Konstruktor, a dodawanie automatyczne obiektu do listy** → mogę chcieć aby w momencie tworzenia obiektu przez konstruktora, żeby ten od razu znalazł się na danej liście. Robię to w definicji klasy dodając obiekt do już istniejącej listy za pomocą append.

Np.

class Student:

    lista\_studentow = []

    def \_\_init\_\_(self, imie, nazwisko, wiek):

        self.imie = imie

        self.nazwisko = nazwisko

        self.wiek = wiek

        self.lista\_studentow.append(self)

**Dekorator @ i metoda statyczna** → metoda ta pozwala zdefiniować funkcje i metody w klasie bez parametru self. Czyli działa jak zwykła, standardowa funkcja. Jedyne co trzeba zrobić to przed zdefiniowanem takiej metody trzeba wstawić dekorator, czyli @ i staticmethod. Oznacza to, że to metoda statyczna, która nie odnosi się do żadnego obiektu. Definiując metodę statyczną  zwróć uwagę, że nie korzystasz z self, które odnosi się do danego obiektu, ale do danej klasy, ponieważ metoda jest definiowana w klasie. Zwróć uwagę, że zapis @staticmethod nie zawiera na końcu nawiasu, mimo, że Spyder podpowiada, żeby ten się tam znalazł. Wywoływanie metod statycznych: metody stworzone w ten sposób wywołujesz za pomocą wzoru: Klasa.metoda.

Np.

# %%

class Student:

    lista\_studentow = []

    def \_\_init\_\_(self, imie, nazwisko, wiek):

        self.imie = imie

        self.nazwisko = nazwisko

        self.wiek = wiek

        self.lista\_studentow.append(self)

    @staticmethod

    def liczba\_studentow():

        print('Liczba studentów:', len(Student.lista\_studentow))

# %%

student\_1 = Student('Jan', 'Kowalski', 18)

student\_2 = Student('Tomasz', 'Nowak', 23)

student\_3 = Student('Jan', 'Nowak', 20)

# %%

Student.liczba\_studentow()

**Divmod** → funkcja divmod() zwraca tuplę zawierającą iloraz i resztę, gdy argument1 (dzielna) jest dzielona przez argument2 (dzielnik).

Np.

print(divmod(50,25))

print(divmod(50,20))

# %%

#ALBO

class Soldier:

    military\_count = 0

    def \_\_init\_\_(self, name, rank, years\_of\_service):

        self.name = name

        self.rank = rank

        self.years\_of\_service = years\_of\_service

        Soldier.military\_count += 1

    @staticmethod

    def calculate\_years\_of\_service(service\_length):

        years, days = divmod(service\_length, 365)

        return f'{years} year(s), {days} days'

# %%

Soldier.calculate\_years\_of\_service(1000)

**Lekcja\_41\_Dekorator @classmethod**

**Dekorator @classmethod** → odnosi się do całej klasy i tylko na klasie może być wykonywana, to jedyna różnica między tym, a @staticmethod. Wzór to @classmethod i w definicji, w ardumencie należy wpisać cls. Czyli wykonujesz obliczenia, które zwrócą wartosci odnoszące się do całej klasy, sumujące, etc.

Np.

class Substance:

    all\_substances = []

    def \_\_init\_\_(self, name, initial\_temp):

        self.name = name

        self.temperature = [initial\_temp]

        Substance.all\_substances.append(self)

    def add\_temperature(self, temp):

        self.temperature.append(temp)

    @classmethod

    def average\_temperature(cls):

        total\_temp = 0

        for substance in cls.all\_substances:

            total\_temp += sum(substance.temperature)

        return (

            total\_temp

            / len(cls.all\_substances)

            / len(substance.temperature)

        )

# %%

water = Substance('Water', 20)

water.add\_temperature(21)

water.add\_temperature(22)

# %%

oil = Substance('Oil', 25)

oil.add\_temperature(26)

oil.add\_temperature(27)

# %%

print(Substance.average\_temperature())

**Wywoływanie wyniku funkcji @classmethod** → za pomocą print. Sam wynik w konsoli się nie znajdzie.

class AirQualityMeasurement:

    all\_measurements = []

    def \_\_init\_\_(self, location, aqi):

        self.location = location

        self.aqi = [aqi]

        AirQualityMeasurement.all\_measurements.append(self)

    def add\_aqi\_reading(self, aqi):

        self.aqi.append(aqi)

    @classmethod

    def average\_aqi(cls):

        total\_aqi = 0

        num\_measurements = 0

        for measurement in cls.all\_measurements:

            total\_aqi += sum(measurement.aqi)

            num\_measurements += len(measurement.aqi)

        return round(total\_aqi / num\_measurements, 2)

# %%

measurement1 = AirQualityMeasurement('Downtown', 50)

measurement1.add\_aqi\_reading(45)

measurement1.add\_aqi\_reading(60)

# %%

measurement2 = AirQualityMeasurement('Suburb', 40)

measurement2.add\_aqi\_reading(30)

measurement2.add\_aqi\_reading(50)

# %%

print(AirQualityMeasurement.average\_aqi())

**Lekcja\_42-43\_Projekt: Magazyn cz.1-2**

**Projekt: Magazyn Cz.1** → projekt zawierać będzie klasę, która zawierać  będzie obiekty, metody, funkcje. Będzie można pracować z jej instancjami, a  ponadto będzie posiadać interfejs. Zastanów się co jest potrzebne. Na przykładzie magazynu. Informacje przechowane przy pomocy listy, która będzie tworzona przy okazji tworzenia obiektu. Podaj utworzenie listy w konstruktorze - na starcie tworzenia magazynu będzie można jakieś produkty dodać. Czyli definiując klasę podajesz nazwę listy produktów, a pozostałe poniższe metody dodają same produkty, etc. Dodaj metodę wyświetlającą dostępne produkty linia, po linii. Dodaj metodę dodającą produkt. Dodaj metodę usuwającą produkt.

Np.

import sys

class Magazyn:

    def \_\_init\_\_ (self, lista\_produktow):

        self.lista\_produktow = lista\_produktow

    def wyswietl\_produkty (self):

        print('Dostępne produkty:')

        for produkt in self.lista\_produktow:

            print(produkt)

    def dodaj\_produkt (self):

        self.nowy\_produkt = input ('Podaj nazwę produktu, który chcesz dodać: ')

        if self.nowy\_produkt not in self.lista\_produktow:

            self.lista\_produktow.append(self.nowy\_produkt)

        print(f'Produkt {self.nowy\_produkt} został dodany')

    def usun\_produkt (self):

        self.nazwa\_produktu = input('Podaj nazwę produktu, który chcesz'

                                      ' usunąć: ')

        if self.nazwa\_produktu in self.lista\_produktow:

            self.lista\_produktow.remove(self.nazwa\_produktu)

            print('Produkt został usunięty.')

        else:

            print('Produktu nie ma na magazynie.')

**Projekt: Magazyn Cz.2, przygotowanie programu, lista, sys.exit() i znak zachęty ">>>"** → poniżej na przykładzie widać przygotowanie programu, który uruchamia stworzone metody. Jako zakończenie działania programu, wykorzystana jest metoda "sys.exit", która musi zostać wyciągnięta z biblioteki sys - biblioteka ta powstała oczywiście wcześniej, przed zdefiniowaniem klasy. Zwróć jeszcze uwagę na znak ">>>", to oznacza zachętę użytkownika do wprowadzenia swojej odpowiedzi. Biorąc pod uwagę, że w definicji obiektu magazynu jest, że jest to lista, należy zgodnie z tym definiując nowy obiekt wkładać w niego listę.

Np.

# %%

magazyn = Magazyn(['jajka', 'mleko', 'masło'])

print('Witaj Użytkowniku.\nŻeby wyswietlić produkty nacisnij 1\n'

      'Żeby dodać produkt nacisnij 2\nŻeby usunąć produkt nacisnij 3\n'

      'Żeby wyjsć z programu nacisnij 4')

while True:

    wybrana\_opcja = int(input('>>>'))

    if wybrana\_opcja is 1:

        magazyn.wyswietl\_produkty()

    elif wybrana\_opcja is 2:

        magazyn.dodaj\_produkt()

    elif wybrana\_opcja is 3:

        magazyn.usun\_produkt()

    elif wybrana\_opcja is 4:

        print('Zamykanie programu...')

        sys.exit()

    else:

        print('Wybrano niepoprawną opcję!')

**Uruchamianie powyższego programu z PowerShell** → show in external file, Shift+prawy, Open PowerShell, ls, przejdź do intersującego cię katalogu, ls, wybierz plik, uruchom w python (python .\nazwa\_pliku). Potem clear żeby wyczyscić.

**Lekcja\_44\_Dziedziczenie klas**

**Dziedziczenie klas** → dziedziczenie pozwala tworzyć podstawowe klasy, a następnie je rozszerzać. Klasą bazową nazywa się klasę z której się dziedziczy. Proces dziedziczenia wygląda tak, że masz klasę bazową, z której dziedziczysz i jak powstaje inna klasa, która będzie z niej dziedziczyć, to ta dziedzicząca będzie miała w parametrze nazwę tej klasy dziedziczonej. Dziedziczenie oznacza, że klasa dziedzicząca będzie miała wszystkie funkcjonalności klasy dziedziczonej.

**Super i nadpisywanie** → możesz nadpisać w klasie dziedziczącej coś z klasy dziedziczonej, np. parametry \_\_init\_\_. Zrobisz to za pomocą funkcji super(). Funkcja ta nie przyjmie tu żadnego argumentu i po kropce odniesie się do \_\_init\_(parametr\_1, parametr\_2...), ale bez self - to już ogarnia ta właśnie funkcja. Nadpisując \_\_init\_ następnie dodaj interesujące cię parametry w normalny sposób. Jednocześnie pamiętaj, że w definiowaniu przy nadpisywaniu nadal wpisujesz parametry z klasy dziedziczonej + nowe parametry. Funkcja super pobiera jedynie zapisy poszczególnych parametrów.

Np.

class Czlowiek:

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

        self.imie = imie

        self.nazwisko = nazwisko

    def info(self):

        print(f'Imię tej osoby to {self.imie}, a nazwisko to {self.nazwisko}.')

class Pilkarz(Czlowiek):

    def \_\_init\_\_(self, imie, nazwisko, klub):

        super().\_\_init\_\_(imie, nazwisko)

        self.klub = klub

    def info\_zawodnik(self):

        print(f'Zawodnik {self.imie} {self.nazwisko} z {self.klub}.')

Wywoływanie metod z klasy dziedziczonej i dziedziczącej --> możesz wykorzystywać metody zdefiniowane z obu klasach, nawet jeśli znajdują się w nich te same obiekty.

Np.

class Czlowiek:

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

        self.imie = imie

        self.nazwisko = nazwisko

    def info(self):

        print(f'Imię tej osoby to {self.imie}, a nazwisko to {self.nazwisko}.')

class Pilkarz(Czlowiek):

    def \_\_init\_\_(self, imie, nazwisko, klub):

        super().\_\_init\_\_(imie, nazwisko)

        self.klub = klub

    def info\_zawodnik(self):

        print(f'Zawodnik {self.imie} {self.nazwisko} z {self.klub}.')

    def klub(self):

        print(f'Klub: {self.klub}')

# %%

pilkarz\_1 = Pilkarz('Robert', 'Lewandowski', 'Bayern')

pilkarz\_2 = Pilkarz('Krzysztof', 'Piątek', 'AC Milan')

# %%

pilkarz\_1.info()

pilkarz\_2.info\_zawodnik()

**Nadawanie tej samej nazwy metodom w klasie dziedziczącej i dziedziczonej** → zadziała taka metoda wtedy, ale jest to błędne działanie, łatwo się pomylić, bo Python wtedy ignoruje metodę klasy dziedziczonej i przechodzi najpierw do tej "późniejszej", czyli dziedziczącej.

Łączenie metod i funkcja super → funkcję super możesz też wykorzystać do łączenia metod zdefiniowanych w klasie dziedziczonej i dziedziczącej. Funkcja ta wpisana w wybranym fragmencie definicji nowej metody, która łączy inne, wybrane metody będzie wykonywać cokolwiek jedna z tych metod, które wybrałeś.

Np.

class Czlowiek:

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

        self.imie = imie

        self.nazwisko = nazwisko

    def info(self):

        print(f'Imię tej osoby to {self.imie}, a nazwisko to {self.nazwisko}.')

class Pilkarz(Czlowiek):

    def \_\_init\_\_(self, imie, nazwisko, klub):

        super().\_\_init\_\_(imie, nazwisko)

        self.klub = klub

    def info\_cale(self):

        super().info()

        print(f'Klub: {self.klub}')

# %%

pilkarz\_1 = Pilkarz('Robert', 'Lewandowski', 'Bayern')

pilkarz\_2 = Pilkarz('Krzysztof', 'Piątek', 'AC Milan')

# %%

pilkarz\_1.info\_cale()

pilkarz\_2.info\_cale()

**Lekcja\_45\_Dziedziczenie wielokrotne klas**

**Dziedziczenie wielokrotne klas** → polega na dziedziczeniu przez klasę z dwóch lub więcej klas. Podawanie klas dziedziczonych, w nawiasie, po przecinku. SUkcesja wykorzystywania metod: najpierw python będzie ich szukał w klasie, a potem przejdzie do każdej kolejnej klasy dziedziczonej w nawiasie. To samo tyczy się wartościom przypisanym obiektom - jeśli są obiekty nazywające się tak samo, to pierwszeństwo ma ten w klasie "wyższej". Należy zwracać uwagę na kolejność podawanych klas.

Np.

class Czlowiek:

    pochodzenie = 'Ziemia'

    imie = 'Jack'

class Polak:

    kraj = 'Polska'

    imie = 'Piotr'

class Pilkarz(Czlowiek, Polak):

    def info(self):

        print(f'Utworzony obiekt pochodzi z planety {self.pochodzenie}.\n'

              f'Kraj pochodzenia: {self.kraj}.\n'

              f'Nazwa obiektu: {self.imie}.')

# %%

pilkarz\_1 = Pilkarz()

pilkarz\_1.info()

**Lekcja\_46\_Dziedziczenie wielokrotne klas - problem**

**Dziedziczenie wielokrotne klas - problem** → problem został opisany i rozwiązany na poniższym przykładzie. Chodzi o problem dziedziczenia. Wszystkie poniższe klasy mają te same metody. Klasa B i C jednak dziedziczą po klasie A, a klasa D po klasie B i C (w tej kolejności - co nie jest bez znaczenia!). Zwróć uwagę, że wykorzystana została klasa D w tworzeniu obiektu. Jako pierwsza zatem wykorzystana będzie metoda  z klasy D. Ale po zrobieniu pass, ponieważ w atrybutach klasy D jest dziedziczenie z klasy B i C, to w takiej kolejności wykorzystywane będą kolejne metody, a przy kolejnych pass, będzie to dopiero wtedy metoda z klasy A.

class A:

    def metoda(self):

        print('Metoda klasy A.')

class B(A):

#    def metoda(self):

#        print('Metoda klasy B.')

    pass

class C(A):

#    def metoda(self):

#        print('Metoda klasy C.')

    pass

class D(B, C):

#    def metoda(self):

#        print('Metoda klasy D.')

    pass

# %%

d = D()

d.metoda()

**Lekcja\_47\_Dziedziczenie wielopoziomowe klas**

**Dziedziczenie wielopoziomowe klas** → polega na przekazywaniu atrybutów i metod/funkcji w klasach, gdzie istnieje dziedziczenie na więcej niż jednym poziomie, a więc: klasa\_1, klasa\_2(1), klasa\_3(2) - klasa\_3 nadal będzie mieć obiekty z klasy 1, mimo, że jest między nimi klasa 2. Po prostu są one połączone.

Np.

class Czlowiek:

    pochodzenie = 'Ziemia'

    imie = 'Jack'

class Polak(Czlowiek):

    kraj = 'Polska'

#    imie = 'Piotr'

class Programista (Polak):

    technologia = 'Python'

#    imie = 'Krzysztof'

    def info(self):

        print(f'Pochodzenie: {self.pochodzenie}\n'

              f'Kraj: {self.kraj}\n'

              f'Technologia: {self.technologia}\n'

              f'Imię: {self.imie}')

# %%

programista\_1 = Programista()

programista\_1.info()

**Metody i super** → nadpisywanie metod za pomocą super, a zatem dopisywanie czegoś do nich wymaga tylko po super(). napisania nazwy metody, którą nadpisujesz i do tego dodanie elementów, które cię interesują.

# %%

class Weather:

        def \_\_init\_\_(self, city, temp, humidity):

            self.city = city

            self.temp = temp

            self.humidity = humidity

        def report(self):

            return (

                f'The weather in {self.city} is currently:\n'

                f'\t - {self.temp} degrees\n\t - {self.humidity}% humidity'

            )

class SunnyWeather(Weather):

        def \_\_init\_\_(self, city, temp, humidity, uv\_index):

            super().\_\_init\_\_(city, temp, humidity)

            self.uv\_index = uv\_index

        def report(self):

            return super().report() + '\n\t - sunny'

# %%

w = SunnyWeather('Warsaw', 23, 60, 80)

print(w.report())

**Lekcja\_48\_Zmienne publiczne, chronione i prywatne**

**Zmienne publiczne, chronione i prywatne** → możemy mówić o trzech typach  zmiennych: public: zmienna (publiczna); protected: \_zmienna (chroniona); private: \_\_zmienna (prywatna). Ich sposób zapisu różni się pokazanymi podkreślnikami. Różnice: zmienna publiczna dostępna jest z poziomu klasy, klasy pochodniej (po której dziedziczę) i wszędzie poza pochodną; zmienna protected dostępna jest z poziomu klasy i klasy pochodnej; zmienna private jest dostępna tylko z poziomu klasy, którą definiuję. Istnieje umowa niepisana, że zmiennych prywatnych nie należy dotykać, chronionych nie należy dotykać poza klasą i klasą pochodną - będąc deweloperem Python trzeba być tego świadomym. Decydowanie o typie zmiennych, a zatem ich zapisanie odbywa się w konstruktorze po self.

Np.

class Spolka:

    def \_\_init\_\_(self, rodzaj, rynek, gielda):

        self.rodzaj = rodzaj

        self.\_rynek = rynek

        self.\_\_gielda = gielda

class KGHM(Spolka):

    def \_\_init\_\_(self, rodzaj, rynek, gielda, nazwa):

        super().\_\_init\_\_(rodzaj, rynek, gielda)

        self.nazwa = nazwa

        print(f'Atrybut publiczny: {self.rodzaj}')

        print(f'Atrybut chroniony: {self.\_rynek}')

        print(f'Atrybut prywatny: {self.\_\_gielda}')

**Korzystanie z atrybutów publicznych, chronionych, prywatnych** → Python nie pozwoli na wykorzystanie, nawet w print atrybutu prywatnego, ale pozwoli na skorzystanie z publicznego/chronionego. Z prywatnych po prostu nie powinno się korzystać.

Np.

class Spolka:

    def \_\_init\_\_(self, rodzaj, rynek, gielda):

        self.rodzaj = rodzaj

        self.\_rynek = rynek

        self.\_\_gielda = gielda

class KGHM(Spolka):

    def \_\_init\_\_(self, rodzaj, rynek, gielda, nazwa):

        super().\_\_init\_\_(rodzaj, rynek, gielda)

        self.nazwa = nazwa

        print(f'Atrybut publiczny: {self.rodzaj}')

        print(f'Atrybut chroniony: {self.\_rynek}')

        print(f'Atrybut prywatny: {self.\_\_gielda}')

# %%

spolka = Spolka('Spółka Akcyjna', 'Główny', 'GPW w Warszawie')

print(f'Atrybut publiczny: {spolka.rodzaj}')

print(f'Atrybut chroniony: {spolka.\_rynek}')

#print(f'Atrybut prywatny: {spolka.\_\_gielda}')

**Wykorzystywanie zmiennych prywatnych na siłę** → te zmienne nie powinny mnie interesować, ale jeśli sam napisałem je i koniecznie chcę z nimi pracować to mogę dostać się do nich i wykorzystać je odwołując się dodając jeden podkreślnik klasę po której dziedziczę i dopiero potem z dwoma podkreślnikami nazwa atrybutu prywatnego: zmienna.\_Klasa-źródłowa\_\_atrybut-prywatny.

Np.

class Spolka:

    def \_\_init\_\_(self, rodzaj, rynek, gielda):

        self.rodzaj = rodzaj

        self.\_rynek = rynek

        self.\_\_gielda = gielda

class KGHM(Spolka):

    def \_\_init\_\_(self, rodzaj, rynek, gielda, nazwa):

        super().\_\_init\_\_(rodzaj, rynek, gielda)

        self.nazwa = nazwa

        print(f'Atrybut publiczny: {self.rodzaj}')

        print(f'Atrybut chroniony: {self.\_rynek}')

#        print(f'Atrybut prywatny: {self.\_\_gielda}')

# %%

kghm = KGHM('Spółka Akcyjna', 'Główny', 'GPW w Warszawie', 'KGHM')

print(f'Atrybut prywatny: {kghm.\_Spolka\_\_gielda}')

**Lekcja\_49\_Metody magiczne**

**Metody magiczne** → metody magiczne to metody, które na liście metod mają dwa podkreślniki po obu stronach. Python "pod spodem" wykonuje  magiczne metody. Np. jak coś dodajesz 1 + 2 to Python wykonuje metodę int.\_\_add\_\_(1, 2).

Np.

1+1

# %%

type(1)

# %%

dir(1)

# %%

int.\_\_add\_\_(1, 2)

**Najczęściej wykorzystywane metody magiczne** → lista poniżej.

1.Operatory:

Dodawanie: + : \_\_add\_\_(self, other)

Odejmowanie: - : \_\_sub\_\_(self, other)

Mnożenie: \* : \_\_mul\_\_(self, other)

Dzielenie: / : \_\_truediv\_\_(self, other)

Dzielenie, które obcina resztę: // : \_\_floordiv\_\_(self, other)

Dzielenie modulo: % : \_\_mod\_\_(self, other)

Potęgowanie: \*\* : \_\_pow\_\_(self, other)

2. Operatory porównawcze:

Mniej niż: < : \_\_lt\_\_(self, other)

Więcej niż: > : \_\_gt\_\_(self, other)

Mniejszy lub równy<= : \_\_le\_\_(self, other)

Większy lub równy>= : \_\_ge\_\_(self, other)

Równy: == : \_\_eq\_\_(self, other)

Różny od: != : \_\_ne\_\_(self, other)

**Nadpisywanie metod magicznych** → nie jest to rekomendowane, ale na własne potrzeby można nadpisać klasy/metody wykonujące obliczenia na inne niż zaproponowane przez Python. Można dziedziczyć po klasach Python w takim przypadku, np. po klasie int, czyli po liczbach całkowitych.

Np.

class SpecialInt(int):

    def \_\_init\_\_(self, special\_int):

        self.special\_int = special\_int

    def \_\_add\_\_(self, other):

        return self.special\_int + other.special\_int + 10

    def \_\_sub\_\_(self, other):

        return self.special\_int - other.special\_int - 10

# %%

s\_1 = SpecialInt(1)

s\_2 = SpecialInt(2)

s\_1 + s\_2

# %%

s\_1 - s\_2

**Nadpisywanie metod i upraszczanie kodu** →  jak w przykładzie \_\_str\_\_ i \_\_add\_\_ i \_\_sub\_\_ i \_\_lt\_\_. Upraszczanie kodu natomiast polega na tym, że jeśli coś mi się powtarza wiele razy (w przykładzie liczenie pola kwadratu) to to również możesz zdefiniować.

Np.

class Kwadrat:

    def \_\_init\_\_(self, dlugosc\_boku):

        self.dlugosc\_boku = dlugosc\_boku

    def \_\_str\_\_(self):

        return f'Kwadrat o boku {self.dlugosc\_boku}cm.'

    def pole(self):

        return self.dlugosc\_boku \*\*2

    def \_\_add\_\_(self, other):

#        return self.dlugosc\_boku \*\* 2 + other.dlugosc\_boku \*\* 2

         return self.pole() + other.pole()

    def \_\_sub\_\_(self, other):

#        return self.dlugosc\_boku \*\* 2 - other.dlugosc\_boku \*\* 2

         return self.pole() - other.pole()

    def \_\_lt\_\_(self, other):

#        return self.dlugosc\_boku \*\* 2 < other.dlugosc\_boku \*\* 2

         return self.pole() < other.pole()

# %%

k1 = Kwadrat(3)

k2 = Kwadrat(10)

# %%

print(k1)

print(k2)

# %%

k1 + k2

# %%

k1 - k2

# %%

k1 < k2

**Różnica między \_\_str\_\_ i \_\_repr\_\_** → Metoda \_\_str\_\_() zwraca czytelny dla człowieka, czyli nieformalny, ciąg znaków reprezentujący obiekt. Ta metoda jest wywoływana przez wbudowane funkcje print(), str() i format(). Jeśli nie zdefiniujesz metody \_\_str\_\_() dla klasy, wbudowana implementacja obiektu wywoła zamiast tego metodę \_\_repr\_\_(). Metoda \_\_repr\_\_() zwraca bardziej bogatą w informacje, czyli oficjalną, ciągową reprezentację obiektu. Ta metoda jest wywoływana przez wbudowaną funkcję repr(). Jeśli to możliwe, zwrócony ciąg znaków powinien być prawidłowym wyrażeniem języka Python, którego można użyć do odtworzenia obiektu. We wszystkich przypadkach ciąg znaków powinien mieć charakter informacyjny i jednoznaczny. Ogólnie rzecz biorąc, ciąg \_\_str\_\_() jest przeznaczony dla użytkowników, a ciąg \_\_repr\_\_() jest przeznaczony dla programistów. Przykład użycia klasy wbudowanej: klasa datetime.datetime jest wbudowaną klasą języka Python, która ma domyślną implementację metod \_\_str\_\_() i \_\_repr\_\_(). Poniższy przykładowy kod przedstawia ciągi znaków zwracane przez domyślną implementację metod \_\_str\_\_() i \_\_repr\_\_() dla obiektu datetime.datetime:

import datetime

mydate = datetime.datetime.now()

print("\_\_str\_\_() string: ", mydate.\_\_str\_\_())

print("str() string: ", str(mydate))

print("\_\_repr\_\_() string: ", mydate.\_\_repr\_\_())

print("repr() string: ", repr(mydate))

#Inny przykład

# %%

class Rectangle:

    def \_\_init\_\_(self, width, height):

        self.width = width

        self.height = height

    def \_\_str\_\_(self):

        return f'Rectangle({self.width}, {self.height})'

    def \_\_repr\_\_(self):

        return f'Rectangle({self.width}, {self.height})'

    def \_\_add\_\_(self, other):

        new\_width = self.width + other.width

        new\_height = self.height + other.height

        return Rectangle(new\_width, new\_height)

**Lekcja\_50\_Projekt: funkcja kwadratowa**

**Funkcja kwadratowa - projekt** → klasa reprezentująca wszystkie funkcje kwadratowe. Zaimportowana biblioteka numpy będzie potrzebna do wykonywania testów. W przykładzie brane są pod uwagę wszystkie możliwości przy wykonywaniu funkcji kwadratowej. Ponadto na końcu kodu znajduje się else raise Error, co oznacza, że jeśli nie wzięte zostały pod uwagę wszystkie scenariusze albo popełniony został błąd, to ta metoda samokontroli to wskaże.

Np.

import numpy as np

class FunkcjaKwadratowa:

    def \_\_init\_\_(self, a, b, c):

        self.a = a

        self.b = b

        self.c = c

    def rownanie(self):

        print(f'{self.a} \* x \*\* 2 + {self.b} \* x \*\* 2 + {self.c}')

    def rozwiaz(self):

        if self.a == 0 and self.b == 0 and self.c == 0:

            print('Równanie tożsamosciowe.')

        elif self.a == 0 and self.b == 0 and self.c != 0:

            print('Równanie sprzeczne.')

        elif self.a == 0 and self.b != 0:

            x = -self.c / self.b

            print(x)

        elif self.a != 0:

            delta = self.b \*\* 2 - 4 \* self.a \* self.c

            if delta > 0:

                    sqrt\_delta = np.sqrt(delta)

                    x1 = (-self.b - sqrt\_delta) / (2 \* self.a)

                    x2 = (-self.b + sqrt\_delta) / (2 \* self.a)

                    print (x1, x2)

            elif delta == 0:

                x = - self.b / (2 \* self.a)

                print(x)

            else:

                print('Brak rozwiązań.')

        else:

            raise Error

# %%

fk\_1 = FunkcjaKwadratowa(3, 1, 4)

fk\_2 = FunkcjaKwadratowa(3, 1, -4)

fk\_1.rownanie()

fk\_1.rozwiaz()

fk\_2.rownanie()

fk\_2.rozwiaz()

**Testowanie poprawności funkcji** → przykładowa funkcja kwadratowa może mieć błędy spowodowane nieprzewidzianymi scenariuszami, co wyświetli znajdująca się opcja error. Żeby sprowokować ją (jej hipotetyczne istnienie) należy w losowy sposób i odpowiednią ilość kombinacji wykonać obliczenia.Ostatecznie należy podać, że jeśli skrypt będzie uruchamiany bezpośrednio uruchomiona zostanie funkcja.

Np.

import numpy as np

class FunkcjaKwadratowa:

    def \_\_init\_\_(self, a, b, c):

        self.a = a

        self.b = b

        self.c = c

    def rownanie(self):

        print(f'{self.a} \* x \*\* 2 + {self.b} \* x \*\* 2 + {self.c}')

    def rozwiaz(self):

        if self.a == 0 and self.b == 0 and self.c == 0:

            print('Równanie tożsamosciowe.')

        elif self.a == 0 and self.b == 0 and self.c != 0:

            print('Równanie sprzeczne.')

        elif self.a == 0 and self.b != 0:

            x = -self.c / self.b

            print(x)

        elif self.a != 0:

            delta = self.b \*\* 2 - 4 \* self.a \* self.c

            if delta > 0:

                    sqrt\_delta = np.sqrt(delta)

                    x1 = (-self.b - sqrt\_delta) / (2 \* self.a)

                    x2 = (-self.b + sqrt\_delta) / (2 \* self.a)

                    print (x1, x2)

            elif delta == 0:

                x = - self.b / (2 \* self.a)

                print(x)

            else:

                print('Brak rozwiązań.')

        else:

            raise Error

# %%

def main():

    for i in range(10000):

        a = np.random.choice(range(-100, 100))

        b = np.random.choice(range(-100, 100))

        c = np.random.choice(range(-100, 100))

        fun = FunkcjaKwadratowa(a, b, c)

        fun.rozwiaz()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Hermetyzacja** → w kontekście klas w języku Python, hermetyzacja oznacza technikę programowania, w której atrybuty klasy są ukrywane przed dostępem spoza klasy.

**Polimorfizm** → w kontekście języka Python, polimorfizm oznacza technikę programowania, w której obiekty danej klasy posiadają różne metody o takiej samej nazwie.

**Klasa abstrakcyjna** → w kontekście języka Python, klasa abstrakcyjna oznacza klasę, która nie może tworzyć własnych obiektów. Klasa ta ponadto posiada tylko metody abstrakcyjne.

**SEKCJA 11. TESTY JEDNOSTKOWE (UNIT TEST)**

**Lekcja\_51\_Wyłapywanie błędów**

**Wyłapywanie błędów** → lista najbardziej typowych błędów w Python:

Np.

1 / 0

#ZeroDivisionError: division by zero (nie można dzielić przez zero)

# %%

4 + '4'

#TypeError: unsupported operand type(s) for +: 'int' and 'str' (łączenie

#zmiennych różnego typu)

# %%

int('f')

#ValueError: invalid literal for int() with base 10: 'f' (znaku nie da się

#skonwertować na liczbę)

# %%

float('sd')

#ValueError: could not convert string to float: 'sd' (znaku nie da się

#skonwertować na liczbę)

**Klauzula try-except** → pozwala ona spróbować wykonać fragment kodu, który jeśli wyświetli błąd to spróbuje zrobić coś innego, na przykład komunikat o błędzie własnego autorstwa - zawsze wygląda to lepiej niż błąd wyskakujący w środku uruchamiania programu. Dwukropek za except oznacza, że dany autorski komunikat będzie obstawiać wszystkie błędy.

Np.

try:

    1 / 0

except:

    print('Nie dzieli się przez zero!')

**Klauzula try-except przy rozbudowanej funkcji** → działając przy rozbudowanej funkcji i nie mogąc kontrolować kilku rodzajów błędów naraz, standardem jest w klauzuli try-except, za except nie stawiać dwukropka, tylko nazwy konkretnego błędu, który klauzula ma weryfikować. Jest tak, ponieważ w rozbudowanych funkcjach mogą pojawić się różne błędy. Żeby poznać jaki error zwróci dany błąd i móc go wpisać do except, po prostu wpisz błędny fragment kodu do konsoli.

Np.

try:

    1 / 0

except ZeroDivisionError:

    print('Nie dzieli się przez zero!')

except TypeError:

    print('Zły typ!')

# %%

try:

    4 + '4'

except TypeError:

    print('Nie można dodawać tekstu do liczby!')

# %%

try:

    int('sd')

except ValueError:

    print('Zły tekst.')

**Występowanie dwóch błędów jednocześnie** →  jeśli stworzysz pętlę, która zatrzyma się po spełnieniu określonego warunku, to tak długo jak pojawiać się będzie błąd i twój komunikat, tak długo pętla będzie wykonywana - aż nie podasz prawidłowej wartości.

Np.

while True:

    try:

        liczba = int(input('Podaj liczbę: '))

        break

    except ValueError:

        print('Musisz podać liczbę.')

**FileNotFoundError** → błąd przy otwieraniu nieistniejącego pliku.

Np.

try:

    with open('test.txt', 'r') as file:

        for line in file:

            print(line)

except FileNotFoundError:

    print('Plik nie istnieje.')

**Instrukcja raise** → istnieje możliwość podnoszenia błędów samemu. Można tą instrukcję umieścić w każdym miejscu w kodzie. Działa to tak, że podnosi się błąd oraz dopisek własny dotyczący tego błędu.

Np.

raise TypeError('Błąd.')

# %%

raise ValueError('Błąd wartosci.')

**Obsługiwanie wielu błędów naraz** → przewidując wiele różnych błędów, kolejne klauzule try-except będą wprowadzane jedna po drugiej.

def divide(x, y):

    try:

        x = int(x)

        y = int(y)

        return x / y

    except ZeroDivisionError:

        print('Nie można dzielić przez zero!')

    except ValueError:

        print('To muszą być liczby!')

# %%

divide(10,2)

# %%

divide(10,0)

# %%

divide('10', '2')

# %%

divide('10', 2)

# %%

divide (10, 'qwe')

**Lekcja\_52\_Wyrażenie assert**

**Wyrażenie assert** → przy diagnozowaniu błędów może pomóc w Python wyrażenie assert. Jest to wyrażenie, które podnosi się kiedy napotka false. Po wyrażeniu można wpisać komentarz, podpowiadający dla użytkownika. Komentarz ten musi znaleźć się po przecinku. Jeśli false się nie pojawi to konsola nie zwróci nic.

Np.

assert 1 == 1

# %%

assert 1 == 2

# %%

def test\_sum():

    assert sum([1, 2, 3]) == 6, 'Powinno być 6!'

test\_sum()

# %%

def test\_sum():

    assert sum([1, 2, 3]) == 5, 'Powinno być 6!'

test\_sum()

**Uruchamianie skryptu testowego bezpośrednio** → wykorzystanie specjalnej zmiennej \_\_name\_\_ przy testach jednostkowych jest przydatne ponieważ przy sprawdzaniu czy \_\_name\_\_ jest równe \_\_main\_\_ sprawdzasz czy plik jest  uruchamiany bezpośrednio czy przez import.

Np.

def calculate\_income\_tax(amount, tax\_rate, age):

    """

    Calculates the income tax based on the given amount, tax rate,

    and age.

    :param amount: The amount of income.

    :param tax\_rate: The tax rate as a decimal.

    :param age: The age of the taxpayer.

    :return: The amount of income tax.

    """

    if age <= 18:

        return int(min(amount \* tax\_rate, 5000))

    elif age <= 65:

        return int(amount \* tax\_rate)

    else:

        return int(min(amount \* tax\_rate, 8000))

def test\_calculate\_income\_tax():

    calculate\_income\_tax(60000, 0.15, 10) == 5000

    calculate\_income\_tax(60000, 0.15, 18) == 5000

    calculate\_income\_tax(60000, 0.15, 19) == 9000

    calculate\_income\_tax(60000, 0.15, 65) == 9000

    calculate\_income\_tax(60000, 0.15, 66) == 8000

# %%

if \_\_name\_\_ == '\_\_main\_\_':

    test\_calculate\_income\_tax()

**Lekcja\_53\_Pakiet unittest - krok po kroku**

**Testy jednostkowe** → testy, które pozwalają testować tworzone oprogramowanie przez dokonywanie testów weryfikujących poprawność działania pojedynczych elementów, tzw. jednostek programu. Zaletą ich jest możliwość wykonywania ich na bieżąco i w pełni zautomatyzowanych testów, często na elementach  programu, często modyfikowanych. Umożliwia to szybkie wychwycenie błędów i ich lokalizację oraz proces debuggingu - jest to bardziej proste. W Python jest pakiet unittest, który pozwala te testy przeprowadzać. Etapy tworzenia testu jednostkowego (plan każdego testu jednostkowego):

1. Zaimportowanie unittest

2. Zdefiniowanie funkcji do testowania

3. Stworzenie przypadku testowego używając klasy unittest.TestCase

4. Zdefiniowanie testu jako metody klasy TestCase (metoda musi zacząć się od słowa "test")

5. Call assert function

6. Assert function wywoła błąd assertionError jeżeli otrzymamy błąd

7. Wywołaj funkcję main() z modułu unittest

Np.

#1.

import unittest

#2.

def add(x, y):

    return x + y

#3.

class SimpleTest(unittest.TestCase):

    #4.

    def test\_add(self):

        #5.

        self.assertEqual(add(3, 4), 6, msg='Powinno być 7')

#6. --> zostanie wywołany w konsoli błąd

#7.

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

**Uruchamianie skryptów testowych w PowerShell** → uruchamiasz po prostu plik ze skryptem w PowerShell. Na przykład: PS C:\Users\Leszek> python Pakiet\_unittest-krok\_po\_kroku.py.

**Lekcja\_54\_Podstawowe metody asercji**

**Podstawowe metody asercji** → metody asercji to metody testowania. Lista podstawowych metod asercji:

1. assertEqual: sprawdza czy dwa elementy są równe

2. assertNotEqual: sprawdza czy dwa elementy nie są równe

3. assertTrue: sprawdza czy wyrażenie/element jest prawdą

4. assertFalse: sprawdza czy wyrażenie/element nie jest prawdą

5. assertIn: sprawdza przynależnosć (czy należy)

6. assertNotIn:  sprawdza przynależnosć (czy nie należy)

Na początku należy zaimportować unittest, po czym utworzyć klasę, która dziedziczy po klasie TestCase. Wstaw testy, które chcesz wykonać wybierając odpowiednie metody asercji. Uruchom testy bezpośrednio, a zatem za pomocą unittest.main(). Można testy wykonywać z poziomu PowerShell.

Np.

import unittest

class SimpleTest(unittest.TestCase):

    def suma(self):

        self.assertEqual(3 + 7, 9)

    def odejmij(self):

        self.assertNotEqual(3 + 7, 10)

    def prawda(self):

        self.assertTrue(3 + 7 == 11)

    def falsz(self):

        self.assertFalse(3 + 7 == 10)

    def w(self):

        self.assertIn(5, [1, 2, 3, 4])

    def niew(self):

        self.assertNotIn(3, range(4))

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

**Lekcja 55\_Metody asercji - kolekcje**

**Metody asercji - kolekcje** → są cztery podstawowe metody asercji dotyczące kolekcji:

1. assertListEqual: sprawdza czy dwie listy są równe

2. assertTupleEqual: sprawdza czy dwie tuple są równe

3. assertSetEqual: sprawdza czy dwie zbiory są równe

4. assertDictEqual: sprawdza czy dwie słowniki są równe

Wykonuj testy również w PowerShell.

Np.

import unittest

class SimpleTest(unittest.TestCase):

    def test\_1(self):

        self.assertListEqual([1, 2, 3], [1, 2, 4])

    def test\_2(self):

        self.assertTupleEqual((1, 2), (1, 3))

    def test\_3(self):

        self.assertSetEqual({4, 5}, {4, 6})

    def test\_4(self):

        self.assertDictEqual({'a': 1, 'b': 2}, {'a': 1, 'b': 3})

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

**assertRaises** → sprawdza, czy zgłaszany jest wyjątek, gdy wywoływana jest funkcja z dowolnymi argumentami pozycyjnymi lub kluczowymi, które są również przekazywane do funkcji assertRaises(). Test kończy się pomyślnie, jeśli zostanie zgłoszony wyjątek, oznacza błąd, jeśli zostanie zgłoszony inny wyjątek, lub zakończy się niepowodzeniem, jeśli nie zostanie zgłoszony żaden wyjątek. Aby przechwycić dowolną grupę wyjątków, jako wyjątek można przekazać tuplę zawierającą klasy wyjątków.

Np.

import unittest

def calculate\_average(numbers):

    if not numbers:

        return None

    return sum(numbers) / len(numbers)

    def test\_calculate\_average\_invalid\_input(self):

        self.assertRaises(TypeError, calculate\_average, [1, 2, "3", 5])

        self.assertRaises(TypeError, calculate\_average, ["a", "b", "c"])

        self.assertRaises(TypeError, calculate\_avZYerage, [None, None])

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

**Lekcja 56\_Pomijanie pewnych testów**

**Pomijanie pewnych testów** → aby pominąć jakiś test należy wewnątrz metody umieścić metodę skipTest + ewentualnie w argumencie komentarz dlaczego test jest pomijany. Można też użyć dekoratora @unittest.skip, ale ta pierwsza metoda wydaje się lepsza z punktu widzenia hierarchiczności kodu.

Np.

import unittest

def add(x, y):

    return x + y

def sub(x, y):

    return x - y

class SimpleTest(unittest.TestCase):

    def test\_add(self):

        self.skipTest('Pomiń')

        self.assertEqual(add(1, 5), 9)

    @unittest.skip('Pomiń')

    def test\_sub(self):

        self.assertEqual(sub(10, 8), 2)

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

**Lekcja 57\_Warunkowe pomijanie testów**

**Warunkowe pomijanie testów** → polega na pomijaniu testów w zależności od pewnego warunku. Najważniejsze metody warunkowego pomijania testów:

1. unittest.skip(reason): pomija oznaczony test

2. unittest.skipIf(condition, reason): pomija oznaczony test jeśli warunek jest prawdziwy

3. unittest.skipUnless(condition, reason): pomija oznaczony test chyba, że warunek jest prawdziwy

4. unittest.expectedFailure: oznacza test jako błąd, jeśli test będzie niepowodzeniem nie zostanie policzony jako błąd. Jeśli okaże się, że  nie ma błędu, konsola zwróci komunikat "unexpected successes", w przeciwnym przypadku odda "expected failures"

Np.

import unittest

class SimpleCase(unittest.TestCase):

    x = 6

    y = 8

    @unittest.skip('Pomiń')

    def test\_add(self):

        wynik = self.x + self.y

        self.assertEqual(wynik, 8)

    @unittest.skipIf(x < y, 'Pomiń')

    def test\_sub(self):

        wynik = self.x - self.y

        self.assertEqual(wynik, 4)

    @unittest.skipUnless(y == 0, 'Pomiń')

    def test\_div(self):

        wynik = self.x / self.y

        self.assertEqual(wynik, 3.0)

    @unittest.expectedFailure

    def test\_mul(self):

        wynik = self.x \* self.y

        self.assertEqual(wynik, 12)

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

# %%

#Zadanie\_1

import sys

import unittest

class Container:

    def \_\_init\_\_(self):

        if sys.platform.startswith('win'):

            self.code = 'XC-win'

        else:

            self.code = f'XC-{sys.platform}'

class TestContainer(unittest.TestCase):

    @unittest.skipUnless(sys.platform.startswith('win'), 'Requires Windows.')

    def test\_code\_is\_XC\_win\_on\_Windows(self):

        c = Container()

        self.assertEqual(c.code, 'XC-win', 'Expected code to be XC-win.')

    @unittest.skipUnless(sys.platform.startswith('linux'), 'Requires Linux.')

    def test\_code\_starts\_with\_XC\_on\_Linux(self):

        c = Container()

        self.assertEqual(

            c.code, 'XC-linux', 'Expected code to be XC-linux.'

        )

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

**Lekcja 58\_Testowanie błędu**

**Testowanie błędu** → pakiet unittest pozwala na testowanie czy dany fragment kodu zwróci błąd. Jeśli błąd zostanie wyrzucony to test zostanie zdany.

**assertRaises** → metoda przyjmuje argumenty (exception, callable, \*args, \*\*kwargs) i testuje czy błąd zostanie zwrócony. Test jest zdany jeśli oczekiwany błąd zostanie podniesiony, w przeciwnym razie test jest nie zdany. Zatem w argumentach wpisujesz jaki błąd oczekujesz, że będzie zwrócony, dla jakiej funkcji, jakich argumentów.

Np.

import unittest

def div(a, b):

return a / b

class RaiseTest(unittest.TestCase):

def test\_raise(self):

     self.assertRaises(ZeroDivisionError, div, 1, 0)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**Lekcja 59\_Testowanie dzięki dokumentacji - doctest**

**Testowanie dzięki dokumentacji - doctest** → Python zawiera w sobie jeszcze pakiet doctest, który pozwala testować przypadki, które podajemy w dokumentacji naszej funkcji. Jest to użyteczny pakiet jeśli chodzi o pisanie jednolitych kodów, które z dokumentacją są spójne. Aby pakiet doctest przetestował to co jest w dokumentacji, należy trzymać się hierarchii. Podajemy to w taki sposób jakbyśmy podawali do interpretera w konsoli. Znak zachęty >>> wywołanie funkcji i w następnej linijce wynik. To będę czytać jako dokumentacja, która daje przykład jak funkcję wywoływać a także dzięki takiej dokumentacji jestem w stanie sprawdzić czy to co podawane w dokumentacji jest prawdą. Należy też dodać, że uruchamiam skrypt bezpośrednio to zaimportowany będzie doctest i z pakietu doctest użyta będzie metoda testmod(). Jeśli nic nie zostanie zwrócone do konsoli to oznacza, że testy przebiegły pomyślnie.

def add(x, y):

"""Zwraca sumę dwóch liczb.

>>> add(3, 4)

7

>>> add(2, 8)

100

"""

return x + y

if \_\_name\_\_ == '\_\_main\_\_':

import doctest

doctest.testmod()

**Testowanie za pomocą doctest z pliku** → za pomocą metody testfile, w ramach której przekazujesz nazwę pliku gdzie są te testy. Pamiętaj, że moduły muszą się nazywać od tekstu i być tekstem, niech istnieje plik, z którego importowany jest moduł do treści pliku tekstowego.

Np.

#Treść pliku tekstowego:

Using 'add'

-----------

>>> from doc\_test import add

>>> add(4,5)

9

>>> add(2,4)

8

>>> add(10,13)

23

def add(x, y):

"""Zwraca sumę dwóch liczb.

>>> add(3, 4)

7

>>> add(2, 8)

100

"""

return x + y

if \_\_name\_\_ == '\_\_main\_\_':

import doctest

doctest.testfile('test.txt')

**SEKCJA MOJA\_1. INNE**

**Lekcja\_60\_Data scraping\_przykład**

**Teoria:**

**Upewnij się, że masz odpowiednią wersję Python.**

Command line (cmd): python --version

**Upewnij się, że masz zainstalowany pip.** Pip czyli Python Package Index, czyli najpopularniejsze narzędzie do instalowania pakietów Python.

Command line (cmd): pip --version

**Możliwe, że musisz zainstalować odpowiedni pakiet Python.**

Command line (cmd): pip install package\_name

Np.

pip install beautifulsoup4

**Lista wszystkich pakietów Python.**

List of all libraries (cmd): pip list

**Requests**: biblioteka request jest integralną częścią Pythona służącą do wysyłania żądań HTTP do określonego adresu URL. Niezależnie od tego, czy są to interfejsy API REST, czy Web Scraping, należy poznać żądania, aby móc dalej korzystać z tych technologii. Kiedy ktoś wysyła żądanie do URI, zwraca odpowiedź.

**Html5lib**: biblioteka w czystym Pythonie do analizowania kodu HTML. Został zaprojektowany tak, aby był zgodny ze specyfikacją HTML WHATWG, implementowaną przez wszystkie główne przeglądarki internetowe.

**Bs4**: obiekt BeautifulSoup jest dostarczany przez Beautiful Soup, który jest frameworkiem do scrapingu stron internetowych dla Pythona. Web scraping to proces wydobywania danych ze strony internetowej za pomocą zautomatyzowanych narzędzi w celu przyspieszenia tego procesu.

**Pandas**: biblioteka utworzona na bazie biblioteki NumPy, która udostępnia różne struktury danych i operatory do manipulowania danymi numerycznymi.

**Import as** → skorzystaj z aliasu żeby przywołując określoną bibliotekę korzystać z niej szybciej.

**SOAP (ang. Simple Object Access Protocol)**: protokół komunikacyjny wykorzystujący XML do kodowania wywołań i najczęściej protokół HTTP do ich przenoszenia, możliwe jest jednak wykorzystanie innych protokołów do transportu danych.

**Respond.text zwraca treść odpowiedzi w formacie Unicode.** Zasadniczo odnosi się do treści Binary Response. Żądania Pythona są zwykle używane do pobierania treści z określonego identyfikatora URI zasobu. Ilekroć wysyłamy żądanie do określonego URI za pośrednictwem Pythona, zwraca on obiekt odpowiedzi. Teraz ten obiekt odpowiedzi będzie używany w celu uzyskania dostępu do pewnych funkcji, takich jak treść, nagłówki itp.

**Z czego składa się strona internetowa?** Kiedy wszedłeś na stronę z tym artykułem to Twoja przeglądarka wysłała żądanie do serwera internetowego, by pobrać zawartość.  Następnie serwer odesłał Ci odpowiednie pliki, by prawidłowo wyświetlić stronę. Pliki można z grubsza podzielić na:

* HTML – zawierają główną część strony,
* CSS – tutaj jest zdefiniowany wygląd (np. style czcionki, akapitów itp.),
* Zdjęcia – po prostu wszelkiego rodzaju grafiki, aby umożliwić wyświetlanie obrazów,
* JS – pliki Javascript dodające interaktywność na stronie.

**HTML to język, w którym napisana jest większość witryn internetowych.** HTML służy do tworzenia stron i zapewnienia ich funkcjonalności. Kod użyty do uczynienia ich wizualnie atrakcyjnymi jest znany jako CSS i nie będę się tutaj na nim skupiał. W tym momencie ważniejsza jest nauka budowania domów (HTML), a nie projektowania wnętrz (CSS).

**Biblioteka request.**

**Pierwszą rzeczą, jaką musimy zrobić, aby zeskrapować stronę internetową, jest po prostu jej pobranie.** Możemy pobierać strony za pomocą biblioteki request w Python. Biblioteka wyśle żądanie GET do serwera, który pobierze dla nas zawartość HTML z danej strony.

**BeautifulSoup ku pomocy rozprasowania HTML’a!**

BeautifulSoup to biblioteka Pythona, której nazwa pochodzi z „Alicji w Krainie Czarów”, od wiersza Lewisa Carrolla o tej samej nazwie. Krótko mówiąc, BeautifulSoup to pakiet, który analizuje kod HTML (lub XML!) oraz pomaga organizować i formatować dane internetowe w bardziej przyjazne struktury.

**Teraz wykorzystajmy bibliotekę BeautifulSoup, aby przeanalizować pobraną stronę HTML i wyodrębnić tekst ze znacznika p**. Najpierw import biblioteki, a następnie należy utworzyć klasę BeautifulSoup, aby przeanalizować nasz dokument. Wyświetlmy jeszcze raz to co wyżej:

**Metoda text() interfejsu Response pobiera strumień Response i wczytuje go do końca.** Zwraca obietnicę, która jest rozwiązywana za pomocą ciągu znaków. Odpowiedź jest zawsze dekodowana przy użyciu UTF-8.

**Przykładowy projekt:**

Źródło: <https://www.youtube.com/watch?v=8dTpNajxaH0&pp=ygUUcHl0aG9uIGRhdGEgc2NyYXBpbmc%3D>

**Poniżej opis całego procesu:**

1. **Zacznij od zaimportowania z biblioteki bs4 BeautifulSoup**
2. **Zaimportuj requests**
3. **Definiowanie URL**
4. **Zdobywanie informacji, zdobywanie obiektu typu response**
5. **Definiowanie soup w formie tekstu przy wykorzystania parsera (analizatora składniowego) html.**

KOD

from bs4 import BeautifulSoup

import requests

url = 'https://en.wikipedia.org/wiki/List\_of\_largest\_companies\_in\_the\_United\_States\_by\_revenue'

page = requests.get(url)

soup = BeautifulSoup(page.text, 'html')

print(soup)

**Skonkretyzuj o jakie dane ci chodzi po przygotowaniu tzw. 'soup'**. Żeby to zrobić wejdź przez przeglądarkę na stronę www i zrób inspekcję, użyj opcji "wybierz element ze strony" i zobacz jaka jest klasa danej tabeli, która cię interesuje. Skopiuj jako zewnętrzne HTML i zostaw informację o tabeli. W przykładzie to <table class="wikitable sortable jquery-tablesorter">. W przykładzie są dwie albo więcej tabele o tej samej klasie. Wykorzystane zostanie find.

Może okazać się, że wyciągnięta zostanie nie ta tabela co trzeba bo każda ma tą samą klasę.

KOD

soup.find('table')

**W związku z powyższym wykorzystaj find\_all i wyciągnij wszystkie**. Znajdź w kodzie słowa, które są w tabeli, która cię interesuje (przykład: "Rank"). Zwróć uwagę, że w konsoli pojawi się długi kod, który reprezentuje find, a dopiero "Out[1]:" zwraca interesujący cię w find\_all zakres. Skorzystaj z indeksowania żeby wyciągnąć daną w kolejności tabelę.

KOD

# %%

soup.find\_all('table')[1]

**Definiowanie zmiennej z zawartością tabeli**. Najpierw definiujesz jako całą zmienną tabelę, po czym bierzesz nagłówki - znowu korzystasz z typów przypisanych do nagłówków (w przykładzie to "<th>") soup.find('table', class\_ = 'wikitable sortable jquery-tablesorter') - niepotrzebne. Pamiętaj, że nie szukasz "th" dla całego soup, tylko dla zmiennej table utworzonej wcześniej.

KOD:

# %%

table = soup.find\_all('table')[1]

print(table)

# %%

world\_titles = table.find\_all('th')

print(world\_titles)

**Ponieważ chcesz wyciągnąć sam tekst z kodu html, wystarczy, że skorzystasz z list comprehension i iteracji po obiektach listy, którą jest twoja utworzona zmienna**.

KOD:

# %%

world\_table\_titles = [title.text for title in world\_titles]

print(world\_table\_titles)

**strip()** → poza białymi znakami wytnie również znaki akapitu.

KOD:

# %%

world\_table\_titles = [title.text.strip() for title in world\_titles]

print(world\_table\_titles)

**Tworzenie ramy danych (dataframe albo df) z Pandas.**

**Pandas dataframe** → Pandas DataFrame to dwuwymiarowa struktura danych, przypominająca dwuwymiarową tablicę lub tabelę z wierszami i kolumnami.

Ustawiasz, że nazwami kolumn są dane ze zmiennej z nazwami kolumn ze strony.

import pandas as pd

KOD:

# %%

df = pd.DataFrame(columns = world\_table\_titles)

print(df)

**Wyciąganie danych do tabeli** → zwróć uwagę w kodzie HTML strony, że <tr> reprezentują wiersze (kolejne wiersze), a <td> reprezentuje dane zawarte w tych wierszach. Zauważ, że to logiczne: r = rows, d = data.

**Wyciągnij z interesującej cię tabeli wszystkie tr i przypisz do zmiennej.** Teraz kiedy masz treści wierszy, wyciągnij z nich dane. Przeiteruj po każdym wierszu i w ramach iteracji wyciągnij z każdego zestaw danych, na razie żeby je zobaczyć. Przypisuj do zmiennej wszystkie dane, które znajdziesz w wierszu, po czym odwołując się do tej zmiennej utwórz kolejną, która iterując po już elementach listy (a listą jest teraz wiersz z danymi) będzie czyścić każdy element i wyciągać tekst. W tej chwili do zmiennej przypisuje się jednak tylko dane z ostatniego wiersza. Przy okazji pozbądź się pierwszej listy (czyli wiersza), która pojawia się w przykładzie ponieważ jest pusta, poprzez rozpoczęcie iteracji od indeksu 1, a nie 0.

KOD:

# %%

column\_data = table.find\_all('tr')

print(column\_data)

for row in column\_data[1:]:

row\_data = row.find\_all('td')

individual\_row\_data = [data.text.strip() for data in row\_data]

print(individual\_row\_data)

**df.loc** → atrybut Pandas DataFrame.loc umożliwia dostęp do grupy wierszy i kolumn według etykiet lub tablicy logicznej w danej ramce Pandas DataFrame. Inaczej location. Sprawdza indeks w DataFrame. W przykładzie liczysz długość twojego DataFrame, ile jest wierszy. Zwróć uwagę, że nazwy kolumn nie są nadpisane, bo je wcześniej ustaliłeś. Teraz na końcu iteracji wstawiasz df.loc co automatycznie dokonuje append danych z każdego wiersza.

KOD:

# %%

for row in column\_data[1:]:

row\_data = row.find\_all('td')

individual\_row\_data = [data.text.strip() for data in row\_data]

length = len(df)

df.loc[length] = individual\_row\_data

print(df)

**Przenoszenie danych do pliku CSV.**

**to\_csv** → zwraca ciąg CSV z określoną wartością struktury. Postaw znak "r" przed podaniem ścieżki gdzie chcesz umieścić ten plik i na końcu nadaj mu nazwę dodając do ścieżki backslash i nazwa.csv (\nazwa.csv). W celu uniknięcia kopiowania indeksu, w ostatnim argumencie to\_csv wpisz index = False.

Plik, który otrzymujesz to CSV. Oznacza to, że twoja wersja MS Excel może nie od razu przekształcić go na tabelę, tylko umieścić wszystko w jednej kolumnie. W celu przekonwertowania wykorzystaj opcje Dane, Z tekstu, Importuj.

KOD:

# %%

df.to\_csv(r'C:\Users\Leszek\Downloads\Przykład.csv', index = False)

**Lekcja\_61\_Data scraping\_kurs**

Teoria: <https://www.youtube.com/watch?v=XVv6mJpFOb0>

**Język html** → w tym języku cały kod jest tworzony za pomocą tagów. Te tagi są odpowiedzialne za prezentowanie różnych informacji. Na początku kodu może pojawić się tag, który będzie zawierać metainformacje, po czym będzie  tag zawierający zawartość strony.

**tag <div>** → podstawowy tag, który będzie tworzyć inne tagi o różnych stylach. Zawierać może atrybuty typu "class", który może decydować jaki styl będzie tekstu.

**tag <h5>** → tag ten może określać typ nagłówka.

**tag <p>** → akapit.

**tag <a>** → odwołanie do innej strony, którą można odwiedzić.

**tag <script>** → odpowiedzialne za przywołanie skryptów javascript, ale to mnie nie interesuje.

**Praca z BeautufulSoup** → pracując z tą biblioteką musisz doprecyzować jaką metodą będziesz dokonywać analizować (parse) pliki html na obiekty python.

**lxml parser** → domyslny parser html może nie radzić sobie z uszkodzonym kodem html więc lepiej zainstalować bibliotekę lxml parser.

Np.

Command line (cmd): pip install lxml

**Data scraping z kopii strony w html w pliku** → możesz mieć stronę w kodzie html w pliku, a zatem żeby dokonać scrapingu musisz korzystać z metod pozyskiwania danych z pliku. Korzystasz z opcji read ('r'). Przypisz zawartość otwieranego pliku nowej zmiennej. Następnie żeby przeczytać zawartość wchodzisz od razu w utworzoną zmienną i przypisz tą zawartość innej zmiennej.

Np.

from bs4 import BeautifulSoup

with open ('home.html', 'r') as html\_file:

    content = html\_file.read()

print(content)

**Tworzenie soup i prettify()** → teraz tworzysz zmienną w ramach której analizujesz zmienną zawierającą całą interesującą cię zawartość. Prettify natomiast to metoda, która upiększa kod html, dzięki czemu jest bardziej czytelny.

Np.

soup = BeautifulSoup(content, 'lxml')

print(soup.prettify())

**Wyciąganie informacji** → za pomocą find albo find\_all. Szukasz ostatniego lub wszystkich interesujących cię tagów, które podasz jako pierwszy argument. Przeiteruj po zebranych tagach, które znajdują się na liście i za pomocą metody text wyciągnij sam tekst.

Np.

courses\_tags = soup.find\_all('h5')

print(courses\_tags)

for course in courses\_tags:

    print(course.text)

**Inspect** → opcja w twojej przeglądarce pozwalająca na podejrzenie kodu html. Jeśli szukasz kodu jakiegoś odnośnika (np. linku, przycisku) to skorzystaj  z prawym na to i inspect.

**Filtrowanie, wyciąganie informacji znajdującej się w obszerniejszym tagu** → dla find\_all wykorzystaj drugi argument czyli filtr. Filtrując po klasie czyli class, co będzie się zdarzać, pamiętaj żeby dać na końcu podkreślnik, ponieważ jest to jednocześnie instrukcja do definiowania klasy. Za pomocą split możesz wyciągnąć określony element listy, która powstaje. Wybierz ten element korzystając z indeksowania.

Np.

course\_cards = soup.find\_all('div', class\_ = 'card')

for c in course\_cards:

    course\_name = c.h5.text

    course\_price = c.a.text.split()[-1]

    print(course\_name)

    print(course\_price)

    print(f'{course\_name} costs {course\_price}')

**Get** → metoda z biblioteki request, która prosi o dane z określonej strony www.

from bs4 import BeautifulSoup

import requests

html\_text = requests.get('www.timesjobs.com/candidate/job-search.html?searchType=personalizedSearch&from=submit&txtKeywords=python&txtLocation=')

soup = BeautifulSoup(html\_text.text, 'lxml')

**Replace** → wykorzystuj metodę replace w celu pozbycia się niepotrzebnych znaków (np. białych znaków) z rezultatów scrapingu.

Np.

company\_name = job.find('h3', class\_ = 'joblist-comp-name').text.replace(' ', '')

**Wyciąganie pojedynczych obiektów na początku** → dobrą praktyką jest na początku nie korzystać z find\_all tylko z find, zdobyć wszystkie dane pojedyncze, po czym wprowadzić iterację na początku z find\_all.

**Pętla w scrapingu, a kolejność** → zwróć uwagę, że jeśli tworzysz sobie  warunki w wyciąganiu danych to dla pętli ma znaczenie co będzie działo się pierwsze, drugie, etc. Pierwsze warunki to te filtrujące. Czasami wystarczy , że do danego wcięcia dodasz warunek if 'tekst' in zmiennej.

Np.

for job in jobs:

    published\_date = job.find('span', class\_ = 'sim-posted').span.text

    if 'few' in published\_date

**Span w spanie** → może zdarzyć się sytuacja, że element span znajduje się w innym spanie. W takim przypadku skorzystaj z metody span.

Np.

published\_date = job.find('span', class\_ = 'sim-posted').span.text

**Dostawanie się do tagów ukrytych w tagach** → jeżeli chcesz się odwołać do tagów znajdujących się w tagach, znajdujących się w tagach, itd., pisz ich nazwy po kropce. Pierwszym obiektem, może jednak być zmienna, do której nawiązujesz (w przykładzie: 'job')

Np.

more\_info = job.header.h2.a

**Odwołanie się do atrybutu tagu** → może się zdarzyć, że w tagu po spacji jego częścią będzie określony atrybut. Odwołaj się do niego za pomocą nawiasu kwadratowego. W ten sposób dostaniesz się do wartości danego atrybutu.

Np.

more\_info = job.header.h2.a['href']

**Uruchamianie programu scrapującego co x minut** → po pierwsze zbierz cały program i umieść go w jednej metodzie (przy okazji przesuwając wszystko o jeden tabulator). Następnie chcesz żeby program działał bez końca -  do czego użyjesz warunku if \_\_name\_\_ == '\_\_main\_\_'. Potem wykorzystaj metodę speel i ustaw tyle sekund ile chcesz (zaimportuj wcześniej bibliotekę time.

Np.

def find\_jobs():

#Tutaj cała metoda

if \_\_name\_\_ = '\_\_main\_\_':

    while True:

        find\_jobs()

        time\_wait = 10

        print(f'Waiting {time\_wait} minutes...')

        time.sleep(time\_wait \* 60)

**Umieszczane informacji pojawiających się w iteracjach czasowych w plikach tekstowych** → żeby co x czasu w nowym pliku pojawiała się nowa iteracja  informacji wykorzystaj with open przed printowaniem informacji w konsoli (które to printowanie zaraz zamieni się na wpisywanie ich do plików tekstowych). Musisz ponadto zmienić założenie pętli for ze zwykłego iterowania po liniach html, na dodanie iterowania po indeksie i kodzie (tak żeby każda wartość miała swój numer indeksu). Użyj ponadto enumerate na  iterowanym obiekcie, ponieważ wtedy masz w iteracji numer indeksu oraz  iterowany obiekt. Daj znak akapitu na końcu dla lepszego wyglądu pliku  tekstowego.

Np.

for index, job in enumerate(jobs):

#kod scrapingu

with open(f'posts/{index}.txt', 'w') as f:

    f.write(f'Company Name: {company\_name.strip()} \n')

    #kolejne linie tekstu wyciągane

print(f'File saved: {index}')

**Lekcja\_62\_Odczytywanie CSV i SQL**

**Kurs**: https://www.youtube.com/watch?v=q5uM4VKywbA

**Plik CSV** → (ang. comma-separated values, wartości rozdzielone przecinkiem) – format przechowywania danych w plikach tekstowych i odpowiadający mu typ  MIME text/csv.

**Rozgranicznik** → inaczej delimiter, czyli cokolwiek co oddziela od siebie wartości, np. przecinek.

**Biblioteka csv** → ułatwia odczytywanie plików csv. Teoretycznie możliwe jest wykorzystanie metod zwykłego Pythona, chociażby split, ale one nie będą miały wszystkich istotnych funkcjonalności.

**Reader** → metoda biblioteki csv, pozwala na odczytywanie plików csv, co robi się zwykłym sposobem odczytywania plików tak poza tym. Do metody można dodać czym są odseparowane dane (jakie są rozgraniczniki), standardowo, docelowo są to przecinki. Należy ponadto żeby zobaczyć dane przeiterować po każdym akapicie - wynikiem są listy akapitów. Zwróć uwagę, że możesz wyciągać dane kolumny, które reprezentowane są za pomocą indeksów - wstaw odpowiedni numer indeksu w print, żeby uzyskać taki efekt.

Np.

import csv

with open('names.csv', 'r') as csv\_file:

    csv\_reader = csv.reader(csv\_file)

    for line in csv\_reader:

        print(line[2])

        #print(line)

**Odczytywanie bez nagłówków** → skorzystaj z funkcji next w celu pominięcia pierwszej listy, która zwykle stanowi nagłówki.

Np.

import csv

with open('names.csv', 'r') as csv\_file:

    csv\_reader = csv.reader(csv\_file)

    next(csv\_reader)

    for line in csv\_reader:

        print(line)

**Zapisywanie danych do nowego pliku csv** → w przykładzie to przepisywanie danych z iterowanego pliku csv do drugiego, ale przedzielone za pomocą myślników. Żeby to zrobić otwórz plik csv, ten oryginalny i tylko przypisz go do zmiennej. Następnie stwórz plik i przeprowadź potrzebne operacje. Pamiętaj, że myślniki nie są jednak dobrym rozdzielnikiem - poza przecinkami warto korzystać z tabów (\t)

**Writer i dodatkowy argument zmieniający rozgranicznik** → dodatkowy argument  działa również w przypadku metody reader. Metoda write to metoda zapisu do pliku csv. Można dodać argument delimiter= żeby określić jak separowane  mają być wartości. Zwróć uwagę, że ze zmiennej reader korzystasz dopiero po utworzeniu pliku nowego i nadaniu mu rozgraniczników.

**Writerow i wynik** → metoda służąca do wypełniania kolejnych linii akapitów w  trakcie iteracji. Zwróć uwagę, że ponieważ skorzystałeś z biblioteki csv, Python wstawił cudzysłów dla danych, które już zawierały myślnik, np. osoby dwojga nazwisk.

Np.

import csv

with open('names.csv', 'r') as csv\_file:

    csv\_reader = csv.reader(csv\_file)

    with open('new\_names.csv', 'w') as new\_file:

        #csv\_writer = csv.writer(new\_file, delimiter='-')

        csv\_writer = csv.writer(new\_file, delimiter='\t')

        for line in csv\_reader:

            csv\_writer.writerow(line)

# %%

import csv

with open('new\_names.csv', 'r') as csv\_file:

    csv\_reader = csv.reader(csv\_file, delimiter='\t')

    for line in csv\_reader:

        print(line)

**Alt+4** → skomentowanie wielu linii naraz wybierając zakres i wykorzystując ten skrót.

**Praca z plikami csv wykorzystując słownikowy reader** → wykorzystaj metodę DictReader. To sprawi, że każda wartość jest teraz zawarta w słowniku, a jednocześnie nie ma już tytułów kolumn - te stały się kluczami wartości. Większa przejrzystość tego podejścia polega na tym, że teraz chcąc wyciągać daną kolumnę wartości odniosę się nie do iteracji (która może być nieznana dla osoby czytającej kod, co kryje się pod 1 albo 2), ale do nagłówka/klucza.

Np.

import csv

with open('names.csv', 'r') as csv\_file:

    csv\_reader = csv.DictReader(csv\_file)

    for line in csv\_reader:

        #print(line)

        print(line['first\_name'])

**Praca z plikami csv wykorzystując słownikowy writer i writerheader** → w tym przypadku przed iteracją i wstawianiem kolejnych linii tekstu do pliku należy podać zmienną z nagłówkami danych w formie listy, następnie podać tą zmienną jako jeden z argumentów (fieldnames) do metody DictWriter. Jeśli chcę żeby mój plik miał nagłówek to przed iteracją należy jeszcze wykorzystać metodę writeheader.

Np.

import csv

with open('names.csv', 'r') as csv\_file:

    csv\_reader = csv.DictReader(csv\_file)

    with open('new\_names.csv', 'w') as new\_file:

        fieldnames = ['first\_name', 'last\_name', 'email']

        csv\_writer = csv.DictWriter(new\_file, fieldnames=fieldnames)

        csv\_writer.writeheader()

        for line in csv\_reader:

            csv\_writer.writerow(line)

**Przewaga DictWriter nad zwykłym tworzeniem pliku csv** → jak w przypadku DictReader teraz można bardziej przejrzyście w kodzie podawać określone kolumny, które mnie interesują - zrób to poprzez wybór interesujących cię kolumn w zmiennej fieldnames połączonym z usuwaniem nie interesujących cię kolumn w iteracji za pomocą polecenia del.

Np.

import csv

with open('names.csv', 'r') as csv\_file:

    csv\_reader = csv.DictReader(csv\_file)

    with open('new\_names.csv', 'w') as new\_file:

        fieldnames = ['first\_name', 'last\_name']

        csv\_writer = csv.DictWriter(new\_file, fieldnames=fieldnames)

        csv\_writer.writeheader()

        for line in csv\_reader:

            del line['email']

            csv\_writer.writerow(line)

**Kurs**: https://www.youtube.com/watch?v=YHG\_596\_zOE

**Importowanie danych z SQL do Python** → za pomocą przekształcenia danych z SQL na CSV i przerobienie danych CSV w Python.

**Przekształcanie danych w SQL na CSV** → prawym na bazę danych, Tasks, Export Data, wybierz źródło danych (w przykładzie mój Native Client),   nazwę serwera, bazę danych, wybierz transfer za pomocą zapytania, napisz interesujące cię zapytanie i stwórz plik.

Np.

import csv

with open('Test\_CSV.csv', 'r') as test\_file:

    test\_reader = csv.reader(test\_file)

    for line in test\_reader:

        print(line[0])

**Lekcja\_63\_Połączenie z SQL Server za pomocą Python**

**Kurs**: <https://www.youtube.com/watch?v=aF552bMEcO4>

**Biblioteka PyODBC** → zainstaluj tą bibliotekę w cmd.

Np.

#Komenda: pip install pyodbc

**Definiowanie połączenia z bazą SQL** → za pomocą metody connect z biblioteki pyodbc. Zdefiniuj 4 zmienne: Driver, Server, Database i Trusted\_Connection. Driver to będzie SQL Server Native Client 11.0, który jest zainstalowany w SQL 2016. Nazwę serwera weźmiesz z Management Studio - jest to pierwszy, główny obiekt na liście (zielone kółko). Nazwa bazy danych następnie. Potwierdź też trusted connection.

Np.

import pyodbc

conn = pyodbc.connect(

'Driver = {SQL Server Native Client 11.0};'

'Server = DESKTOP-FLCPP4I;'

'Database = AdventureWorks;'

'Trusted\_Connection = yes;')

**CRUD** → to skrót od CREATE, READ, UPDATE i DELETE, czyli działania, które możesz wykonać. W podanym przykładzie będziesz wykonywać właśnie połączenia w celu przeprowadzenie tych operacji. Po ich przeprowadzeniu wyłączysz to połączenie conn.close().

**Cursor** → klasa kursor. Pozwala kodowi Pythona na wykonanie polecenia SQL Server w sesji bazy danych. Przykład poniżej.

**Read** → jesli chcesz odczytać coś z bazy danych zdefiniuj zmienną read i przypisz jej przygotowane połączenie, wyprintuj, że chodzi o read, przygotuj zmienną, która będzie miała metodę cursor, a potem wykonaj na niej metodę execute, która w argumencie będzie miała polecenie SQL. Weź pod uwagę, że podajesz w klauzuli FROM nazwę tabeli. Następnie iterujesz po każdym wierszu i printujesz. Zwróć uwagę, że składnia w definiowaniu connection jest bardzo istotna i, że znaki równosci muszą przylegać do poszczególnych elementów. Za wszystkimi definicjami umieść read (twoje connection).

Np.

import pyodbc

conn = pyodbc.connect(

'Driver={SQL Server Native Client 11.0};'

'Server=DESKTOP-FLCPP4I;'

'Database=AdventureWorks;'

'Trusted\_Connection=yes;')

def read(conn):

print('Read')

cursor = conn.cursor()

cursor.execute('SELECT \* FROM dummy')

for row in cursor:

     print(f'row = {row}')

print()

read(conn)

conn.close()

**Create** → jeżeli chcesz dodać rekord wykorzystaj składnię z commit. Wykorzystaj zdefiniowaną zmienną connection i zrób na niej formułę create. Wyprintuj co chcesz zrobić. Zdefiniuj cursor jako wynikający z twojego connection i w execute wpisz polecenie 'INSERT INTO tabela (kolumna, kolumna), VALUES (?, ?);', (wartość, wartość). Potwierdź operację metodą commit. Dopisz też przygotowane wcześniej read, żeby widzieć zmiany. Za wszystkimi definicjami umieść create (twoje connection).

Np.

import pyodbc

conn = pyodbc.connect(

'Driver={SQL Server Native Client 11.0};'

'Server=DESKTOP-FLCPP4I;'

'Database=AdventureWorks;'

'Trusted\_Connection=yes;')

def read(conn):

print('Read')

cursor = conn.cursor()

cursor.execute('SELECT \* FROM dummy')

for row in cursor:

     print(f'row = {row}')

print()

def create(conn):

print('Create')

cursor = conn.cursor()

cursor.execute(

     'insert into dummy(a,b) values(?,?);',

     (3232, 'catzzz'))

conn.commit()

read(conn)

read(conn)

create(conn)

conn.close()

**Update** → podobne do create. Def create(twoje connection). Printujesz, że update. Cursor i execute bez zmian. W definicji zwróć szczególną uwagę na składnię i kolejność. UPDATE Tabela SET zmieniana kolumna = ? WHERE kolumna narzucająca warunek = ?; i po przecinku w liście najpierw wartość, która pojawi się nowa, zaktualizowana, a dopiero potem wartość narzucająca warunek. Za wszystkimi definicjami umieść update (twoje connection).

Np.

import pyodbc

conn = pyodbc.connect(

'Driver={SQL Server Native Client 11.0};'

'Server=DESKTOP-FLCPP4I;'

'Database=AdventureWorks;'

'Trusted\_Connection=yes;')

def read(conn):

print('Read')

cursor = conn.cursor()

cursor.execute('SELECT \* FROM dummy')

for row in cursor:

     print(f'row = {row}')

print()

def create(conn):

print('Create')

cursor = conn.cursor()

cursor.execute(

     'insert into dummy(a,b) values(?,?);',

     (3232, 'catzzz'))

conn.commit()

read(conn)

def update(conn):

print('Update')

cursor = conn.cursor()

cursor.execute(

     'UPDATE dummy SET a = ? WHERE b = ?;',

     (1234, 'dogzzz'))

conn.commit()

read(conn)

read(conn)

create(conn)

update(conn)

conn.close()

**Delete** → podobne do update/create. delete, printowanie, cursor, execute. W metodzie execute umieść składnię DELETE FROM twoja tabela WHERE warunek;. Po czym umieść commit. Za wszystkimi definicjami umieść delete (twoje connection).

Np.

import pyodbc

conn = pyodbc.connect(

'Driver={SQL Server Native Client 11.0};'

'Server=DESKTOP-FLCPP4I;'

'Database=AdventureWorks;'

'Trusted\_Connection=yes;')

def read(conn):

print('Read')

cursor = conn.cursor()

cursor.execute('SELECT \* FROM dummy')

for row in cursor:

     print(f'row = {row}')

print()

def create(conn):

print('Create')

cursor = conn.cursor()

cursor.execute(

     'insert into dummy(a,b) values(?,?);',

     (3232, 'catzzz'))

conn.commit()

read(conn)

def update(conn):

print('Update')

cursor = conn.cursor()

cursor.execute(

     'UPDATE dummy SET a = ? WHERE b = ?;',

     (1234, 'dogzzz'))

conn.commit()

read(conn)

def delete(conn):

print('Delete')

cursor = conn.cursor()

cursor.execute(

     'DELETE FROM dummy WHERE a > 5;'

     )

conn.commit()

read(conn)

read(conn)

create(conn)

update(conn)

delete(conn)

conn.close()

**Uruchamianie w Powershell** → python nazwa pliku.py

**Lekcja\_64\_Jupyter notebook**

**Kurs**: <https://www.youtube.com/watch?v=HW29067qVWk>

**Jupyter Notebook** → służy do tworzenia interaktywnych dokumentów w notatniku, które mogą zawierać kod na żywo, równania, wizualizacje, multimedia i inne wyniki obliczeń . Jupyter Notebook jest często używany przez programistów, analityków danych i studentów do dokumentowania i demonstrowania przepływów pracy związanych z kodowaniem lub po prostu do eksperymentowania z kodem.

**Instalacja Jupyter** → instaluje się w pakiecie Anaconda i tam można go otworzyć.

**Nowy notebook** → New, wybierz język programowania.

**Podpowiedzi** → listę skrótów klawiszowych i opis podstawowych funkcjonalności znajdziesz w Help.

**Edit mode / command mode** → dwa tryby, edytowania i komend.

**Uruchamianie kodu** → wykorzystaj opcję Cell. Są różne opcje. Uruchamianie komórki, uruchamianie i przechodzenie do kolejnego wiersza, uruchamianie i tworzenie kolejnego wiersza.

**Numery inkrementacji** → czyli numery w In/Out w nawiasach kwadratowych. Nie tylko pokazują kolejność pisania kodu, ale również determinują kolejność wykonywania kodu.

**Uruchamianie skryptu jak w Spyder** → jeżeli chcesz, żeby skrypt uruchamiał się od góry do dołu w sposób z którego korzystałeś dotychczas to albo pisz kod w taki sposób albo skorzystaj z opcji Run All umieszczonej w Cell.

**Dodawanie wiersza kodu** → Insert, below/above.

**Markdown** → tekst, który tłumaczy się na HTML. Dodajesz go do wiersza kodu i execute.

**Specjalne komendy** → korzystasz z Jupyter Notebook jak z interpretera Python, ale są różne specyficzne komendy.

**Jupyter i bash** → możesz pisać komendy jak w bashu, zaczynając kod od wykrzyknika.

Np.

! pip list

**Magics** → specjalne polecenia, zaczynają się od % albo %%. Jeden symbol % oznacza, że argumenty będą pochodzić z jednej linii, a %% oznacza, że cała komórka będzie wykorzystana jako argumenty. Wyświetlanie komend magics.

Np.

%lsmagic

%pwd

%ls

**Czyszczenie jądra/kernela** → Kernel, Restart and clear output. Przydatne żeby oczyścić kod z wyników.

**%matplotlib inline** → magics, który tworzy w notebook wykres. Podaj komendę magics po czym wstaw kod tworzący wykres. Korzystanie z Jupyter Notebook pozwala na szybkie uruchomienie i wyświetlenie wykresu.

Np.

%matplotlib inline

import matplotlib.pyplot as plt

import numpy as np

x = np.linspace(0, 3\*np.pi, 500)

plt.plot(x, np.sin(x\*\*2))

plt.title('A simple chirp');

**%%html** → komenda pozwalająca renderować HTML bezpośrednio. Ponieważ to magisc cell, to cała komórka zostanie wyrenderowana jako HTML. Zwróć uwagę, że żeby kod zadziałał, między komendą, a kodem html musi być enter.

Np.

%%HTML

<iframe src= "https://media.geeksforgeeks.org/wp-content/uploads/20240206111438/uni2.html" height="370" width="400">

**%%timeit** → funkcja sprawdzająca ile czasu zajmie przeprowadzenie podanej operacji.

Np.

%%timeit

square\_names = [n\*n for n in range(1000)]

**Jupyter i Pandas** → możesz natychmiast wyświetlać tabele tworzone za pomocą biblioteki Pandas.

Np.

import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.randn(10,5))

df

**Eksportowanie Notebook** → file, download as.

**Tworzenie wielu kerneli (jądra) używając wiele wersji Python** → file, new notebook.

**Jupyter galleries** → możesz znaleźć online wiele galerii Jupyter, chociażby na GitHub (<https://github.com/ipython/ipython/blob/main/examples/IPython%20Kernel/Trapezoid%20Rule.ipynb>).

**Lekcja\_65\_Biblioteka numpy**

Kurs: <https://www.youtube.com/watch?v=QUT1VHiLmmI>

Kurs prowadzony na Jupyter Notebook.

**NumPy** → jest biblioteką, która obsługuje jedno, dwu, trzy rzędowe (wymiarowe) dane, czyli X, X/Y, X/Y/Z.

**Różnica między NumPy, a listami** → NumPy jest szybsze. Wykorzystuje mniej pamięci. Podczas iterowania obiektów w NumPy nie ma sprawdzania jakie to typy.

#Tworzenie szyku/szeregu liczb --> jedno lub dwuwymiarowe. Za pomocą metody array w argumencie, w głównym nawiasie kwadratowym umieszczasz jedną lub dwie listy liczb.

Np.

import numpy as np

array\_1 = np.array([1,2])

print(array\_1)

array\_2 = np.array([[0.1, 0.5, 1], [2.5, 1.5, 2.4]])

print(array\_2)

**Sprawdzanie ilości wymiarów** → metoda ndim.

Np.

array\_1.ndim

**Sprawdzanie kształtu danych** → to znaczy ile ma wierszy, a ile kolumn. Metoda shape, najpierw pokazuje ile wierszy, potem kolumny.

Np.

array\_2.shape

**Ustalanie ile miejsca ma zająć obiekt i sprawdzenie jego typu i ilość miejsca, które zabiera** → przy definiowaniu zmiennej za pomocą metody array, w drugim argumencie wpisz dtype = typ\_danych. Sprawdzenie typu danych za pomocą metody dtype. Żeby sprawdzić ile zabiera obiekt miejsca metoda itemsize.

Np.

array\_3 = np.array([1, 2, 3], dtype = 'int16')

print(array\_3)

array\_3.dtype

array\_3.itemsize

**Oś NumPy** → kolejne elementy w nawiasach kwadratowych stanowią wiersze. W metodach NumPy można korzystać z odwołań do osi x/y. Oś 0 stanowi pierwszy wymiar, czyli wiersze, a oś 1 drugi czyli kolumny.

Np.

stats = np.array([[1,2,3],[4,5,6],[7,8,9]])

print(stats)

np.min(stats, axis = 0)

np.min(stats, axis = 1)

**Wyciąganie elementu z wierszy/kolumn** → określasz indeks wiersza i kolumny - w tej kolejności. Pamiętaj, że koordynaty podajesz na podstawie indeksowania, czyli liczysz wiersze/kolumny od zera.

Np.

array\_4 = np.array([[1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14]])

array\_4[1, 5]

**Wyciąganie całej kolumny/wiersza** → w pierwszym/drugim argumencie podaj tylko ":".

Np.

array\_4[0, :]

array\_4[:, 0]

**Wyciąganie co któryś element** → wzór to [wiersz, kolumna czyli startindex:endindex:stepsize], czyli start, stop i co ile.

Np.

array\_4[0, 1:-1:2]

**Zamiana pojedynczych danych i całych kolumn/wierszy** → poprzez wskazanie pozycji obiektu albo całego wiersza/kolumny.

Np.

array\_4[1,2] = 20

print(array\_4)

array\_4[0,2] = 5

print(array\_4)

array\_4[1,:] = 7

print(array\_4)

**Praca nad danymi 3D** → podajesz w nawiasie kwadratowym od najwyższego poziomu do najniższego.

Np.

array\_5 = np.array([[[1,2], [3,4]], [[5,6], [7,8]]])

print(b)

array\_5[1,0,0]

**Tworzenie matrycy złożonej z zer, jedynek albo innych liczb** → jeśli chcę stworzyć taką matrycę wykorzystam jedną z trzech metod: np.zeros, np.ones i np.full. W przypadku zeros i ones w nawiasach zwykłych wpisz ile ma być wierszy/kolumn + możesz dodać argument określający typ danych, dtype = typ\_danych. Jeśli chcesz stworzyć matrycę z inną liczbą niż zero/jeden, podaj ją w metodzie full, w drugim argumencie, za układem matrycy.

Np.

np.zeros((2,3))

np.ones((4,2,2), dtype = 'int32')

np.full((2,2), 99)

**Full\_like** → tworzy matrycę z liczbami mojego wyboru, ale która wygląda jak inna matryca, która już istnieje. Wzór: np.full\_like(nazwa\_matrycy, wybrana\_liczba)

Np.

array\_1 = np.array([1,2])

print(array\_1)

np.full\_like(array\_1, 99)

**Tworzenie matrycy z losową zawartością** → za pomocą określonych przeze mnie warunków albo jako kopia kształtu innej matrycy. Wzór: np.random.rand(4,2,3) albo np.random.random\_sample(nazwa\_matrycy.shape).

Np.

np.random.rand(4,2,3)

np.random.random\_sample(array\_1.shape)

**Matryca z losowymi danymi typu integer** → możesz określić najniższą wartość w pierwszym argumencie, najwyższą w drugim. Wzór: np.random.randint(-4, 8, size=(3.3))

Np.

np.random.randint(-4, 8, size=(3,3))

**Tworzenie matrycy o kształcie kwadratu** → czyli zawsze 1x1, 2x2, 3x3, 4x4, 5x5, i tak dalej. Wzór: np.identity(5)

Np.

np.identity(5)

**Powtarzanie szyku wielokrotnie** → tak, żeby dany szyk znalazł się jeden pod drugim. Wzór: np.repeat(powtarzany\_szyk, ile\_razy, axis=0)

Np.

arr = np.array([[1,2,3]])

print(arr)

r1 = np.repeat(arr, 3, axis=0)

print(r1)

**Tworzenie skomplikowanej matrycy** → na przykładzie. Najpierw tworzysz obiekt, którego wielkość (wiersze/kolumny) będzie taka jak pożądana wielkość. Tworzysz dodatkowe, osobne obiekty, które znajdą się w większym obiekcie. Od najmniejszego obiektu przypisujesz coraz to kolejnym, aż w końcu to największy obiekt (jego fragment określony jako wiersze/kolumny) zostanie wypełniony.

Np.

output = np.ones((5,5))

z = np.zeros((3,3))

z[1,1] = 9

output[1:-1, 1:-1] = z

print(output)

**Problem z kopiowaniem w NumPy i metoda copy** → jeśli będziesz chciał w tradycyjny sposób zrobić kopię jakiejś zmiennej (a = b) i potem zmienisz wartość tej kopii, to zmieni się też wartość oryginału. Żeby temu zapobiec wykorzystaj metodę copy: b = a.copy()

Np.

a = np.array([1,2,3])

b = a.copy()

b[0] = 100

print(a)

print(b)

**Dodawanie/odejmowanie/mnożenie/dzielenie** → będzie działać tak jak w Python.

Np.

a = np.array([1,2,3,4])

a - 1

a + 1

a / 2

a \* 2

b = np.array([1,0,1,0])

a + b

a \*\* 2

**Statystyka w NumPy** → min/max, przy podaniu osi (axis) jako 1 poszuka w każdym wierszu tych wartości.

Np.

stats = np.array([[1,2,3],[4,5,6],[7,8,9]])

print(stats)

np.min(stats, axis=0)

np.max(stats, axis=1)

**Sum** → metoda zwracająca sumę matrycy lub podanego wiersza/kolumny. Wzór: np.sum(matryca, axis=0)

Np.

stats = np.array([[1,2,3], [4,5,6]])

np.sum(stats, axis=0)

**Reshape** → metoda służąca do zmiany kształtu matrycy. Działać będzie tylko jeśli nowy kształt będzie zgadzać się z liczbą obiektów w starym kształcie, to jest stary kształt to 4/2, to nowy może być 2/4, ale nie 2/5. Wzór: zmienna.reshape((nowy kształt)).

Np.

before = np.array([[1,2,3,4], [5,6,7,8]])

print(before)

after\_1 = before.reshape((4,2))

print(after\_1)

after\_2 = before.reshape((1,8))

print(after\_2)

**Vstack** → Vertical stacking, czyli układanie pionowe. Pozwala na łączenie wierszy jedno na drugim. Wzór: np.vstack([wiersz\_1,wiersz\_2,wiersz\_3])

Np.

v1 = np.array([1,2,3,4])

v2 = np.array([5,6,7,8])

np.vstack([v1, v2, v1, v2])

**Hstack** → Horizontal stack, układanie poziome, czyli dodawanie elementów na końcu określonego innego elementu. Będą się dodawać zgodnie z kolejnością podawania argumentów. Wzór: np.hstack((poziom\_1, poziom\_2)).

Np.

h1 = np.ones((2,4))

h2 = np.zeros((2,2))

np.hstack((h1, h2))

**Załadowanie danych z pliku txt** → za pomocą metody genfromtxt. Podajesz nazwę i rozszerzenie pliku oraz co rozdziela obiekty (delimiter). Możesz od razu zmienić typ danych za pomocą metody astype.

Np.

filedata = np.genfromtxt('data.txt', delimiter = ',')

filedata = filedata.astype('int32')

print(filedata)

**Prezentacja danych jako PRAWDA/FAŁSZ** → korzystając z prostego większe/mniejsze możesz zmienić dane na boolean (czyli P/F).

Np.

filedata > 50

**Wyciąganie wyników danej operacji** → dokonujesz wyodrębnienia i prezentacji tych wyników za pomocą nawiasu kwadratowego. Wzór: zmienna[zmienna > liczba]

Np.

filedata[filedata > 50]

**Indeksowanie danych z listy** → mając listę, można wyciągać określone obiekty za pomocą odwołań na zasadzie indeksu, ale należy to zrobić w podwójnym nawiasie kwadratowym.

Np.

a = np.array([1,2,3,4,5,6,7,8,9])

a[[0,2,8]]

**Any/all** → metody, które pozwalają na określenie czy którekolwiek/wszytkie wartości znajdujące się w matrycy lub jej kolumnach (axis = 0) czy wierszach (axis = 1) spełniają określony warunek. Wzór: np.all(zmienna > warunek, axis = 0). Można dodawać warunki za pomocą znaku &, ale wtedy warunki muszą być w osobnych nawiasach okrągłych. Możesz też zrobić odwrotność (czyli NOT) dodają znak ~ przed rozpoczęciem warunków.

Np.

np.any(filedata > 50)

np.any(filedata > 0, axis = 0)

np.all(filedata < 200, axis = 1)

((filedata > 50) & (filedata < 100))

(~((filedata > 50) & (filedata < 100)))

**Lekcja\_66\_Biblioteka Seaborn**

**Kurs**: <https://www.youtube.com/watch?v=LnGz20B3nTU>

**Przykładowe dane**: <https://github.com/mwaskom/seaborn-data/blob/master/exercise.csv>

**Matplotlib** → biblioteka do tworzenia wykresów dla języka programowania Python i jego rozszerzenia numerycznego NumPy.

**Seaborn** → biblioteka, która używa Matplotlib do rysowania wykresów. Będzie używana do wizualizacji rozkładów losowych.

Np.

import seaborn as sb

print(sb.get\_dataset\_names())

**sb.get\_dataset\_names()** → funkcja wyciągająca pakiety danych do wykorzystania w bibliotece Seaborn.

Np.

import seaborn as sb

print(sb.get\_dataset\_names())

**Wczytywanie danych i wyświetlanie pięciu pierwszych wierszy** → możliwe jest chociażby wczytywanie danych z CSV, za pomocą wzoru: sb.load\_dataset(plik). Wyświetl pięć pierwszych wierszy za pomocą zmienna.head().

Np.

data = sb.load\_dataset('titanic')

print(data.head())

**Wykres liczenia** → inaczej countplot lub po prostu wykres kolumnowy. Jest to wykres liczbowy. W argumencie: sb.countplot(data=zmienna,os x=wybrana\_kolumna\_danych). W przypadku osi x kolumny są pionowe, w przypadku y są poziome. Za pomocą argumentu hue mogę dodać kolejne elementy/dane do wykresu.

Np.

sb.countplot(data=data, x='sex')

# %%

sb.countplot(data=data, y='sex')

# %%

sb.countplot(data=data, y='sex', hue='alive')

**Histogram** → histogram, wykres ciągły. Wzór: sb.histplot(data=data, x='kolumna')

Np.

data = sb.load\_dataset('exercise')

print(data.head())

sb.histplot(data=data, x='pulse')

**Boxplot** → wykres pudełkowy, wykres ciągły. Wzór: sb.boxplot(data=data, x='kolumna')

Np.

data = sb.load\_dataset('exercise')

print(data.head())

sb.boxplot(data=data, x='pulse')

**Boxenplot** → inny wykres pudełkowy, wykres ciągły. Wzór: sb.boxenplot(data=data, x='kolumna')

Np.

data = sb.load\_dataset('exercise')

print(data.head())

sb.boxenplot(data=data, x='pulse')

**Violinplot** → wykres skrzypcowy, wykres ciągły. Wzór: sb.violinplot(data=data, x='kolumna')

Np.

data = sb.load\_dataset('exercise')

print(data.head())

sb.violinplot(data=data, x='pulse')

**Rugplot** → wykres kreskowy, wykres ciągły. Wzór: sb.rugplot(data=data, x='kolumna')

Np.

data = sb.load\_dataset('exercise')

print(data.head())

sb.rugplot(data=data, x='pulse')

**KDE Plot** → kernel density estimation, wykres gęstości, wykres ciągły. Wzór: sb.kdeplot(data=data, x='kolumna')

Np.

data = sb.load\_dataset('exercise')

print(data.head())

sb.kdeplot(data=data, x='pulse')

**Strip Plot** → wykres kropkowy, wykres ciągły. Wzór: sb.stripplot(data=data, x='kolumna')

Np.

data = sb.load\_dataset('exercise')

print(data.head())

sb.stripplot(data=data, x='pulse')

**Swarm Plot** → wykres kropkowy, wykres ciągły. Wzór: sb.swarmplot(data=data, x='kolumna')

Np.

data = sb.load\_dataset('exercise')

print(data.head())

sb.swarmplot(data=data, x='pulse')

**Ecdfplot** → wykres liniowy, ale schodkowy.

Np.

data = sb.load\_dataset('exercise')

print(data.head())

sb.ecdfplot(data=data, x='pulse')

**Łączenie wykresów** → istnieje możliwość pokazania dwóch wykresów na jednej macierzy. Żeby to zrobić należy ustawić dwa polecenia tworzące wykresy, jedno pod drugim.

Np.

data = sb.load\_dataset('exercise')

print(data.head())

sb.rugplot(data=data, x='pulse', hue = 'diet')

sb.boxenplot(data=data, x='pulse', hue = 'diet')

**Wykresy skupiające dwie zmienne, analiza dwuwariantowa** → poniżej lista wykresów.

Np.

data = sb.load\_dataset('flights')

data.head()

sb.lineplot(data=data, x='year', y='passengers')

sb.pointplot(data=data, x='year', y='passengers')

sb.barplot(data=data, x='year', y='passengers')

**Samplowanie danych** → wybieranie ilości rekordów, które mają być wyciągnięte. Wyciągane będą od początku. Wzór: zmienna = data.sample(x\_rekordów)

Np.

data = sb.load\_dataset('diamonds')

data.head()

data = data.sample(3000)

sb.scatterplot(data=data, x='carat', y='price')

# %%

sb.kdeplot(data=data, x='carat', y='price')

# %%

sb.histplot(data=data, x='carat', y='price')

# %%

sb.jointplot(data=data, x='carat', y='price')

**Bardziej skomplikowany przykład** → zawiera dodatkowe elementy graficzne.

Np.

sb.lmplot(data=data, x='carat', y='price', markers = '.',

          line\_kws={'color':'red'})

**Wizualizacja zmian, siatki** → jeśli chcę zobaczyć jak dane się zmieniły względem jednej zmiennej, jak obiekty się zmieniają i zobaczyć ich wiele to wykorzystam argument col, czyli column, czyli kolumna. I wizualizacja również pojawi się w pewnym sensie w kolumnach.

Np.

sb.relplot(data=data, x='carat', y='price', col='cut')

# %%

data = sb.load\_dataset('tips')

data.head()

facet = sb.FacetGrid(data=data, col='smoker', row='sex')

facet.map(sb.boxplot, 'total\_bill')

# %%

sb.pairplot(data)

**Korelacje** → uruchamiane poprzez wykorzystanie metody corr. Wtedy wszystkie zmienne w danych zostaną skorelowane ze wszystkimi. Jeśli chcę zobaczyć korelację jednej zmiennej z resztą muszę ją odizolować poprzez dodanie [['zmienna']] za metodą corr.

data = sb.load\_dataset('diamonds')

data.head()

sb.heatmap(data.corr())

# %%

sb.heatmap(data.corr(), annot=True, linewidths=1)

# %%

sb.heatmap(data.corr()[['price']], annot=True, linewidths=1)

**Lekcja\_67\_Notatka z Data Analyst Portfolio Project(s)**

**Style.use** → matplotlib. Ten skrypt demonstruje różne dostępne arkusze stylów na wspólnym zestawie przykładowych wykresów: wykres punktowy, obraz, wykres słupkowy, łatki, wykres liniowy i histogram. Każdy z tych arkuszy stylów można zaimportować (tj. aktywować) według jego nazwy. W przykładzie dla stylu ggplot.

Np.

import matplotlib

import matplotlib.pyplot as plt

plt.style.use('ggplot')

from matplotlib.pyplot import figure

**Matplotlib.figure.Figure()** → ta klasa jest kontenerem najwyższego poziomu dla wszystkich elementów wykresu. Instancja Figure obsługuje wywołania zwrotne za pośrednictwem atrybutu callbacks, który jest instancją CallbackRegistry.

Np.

import matplotlib

import matplotlib.pyplot as plt

plt.style.use('ggplot')

from matplotlib.pyplot import figure

**plt.figure(figsize=(x,y))** → podczas tworzenia wykresów za pomocą biblioteki Matplotlib domyślny rozmiar rysunku wynosi 6,4 cala dla szerokości i 4,8 cala dla wysokości (w calach). Można zmienić te parametry używając wzoru: plt.figure(figsize=(x,y)).

Np.

%matplotlib inline

matplotlib.rcParams['figure.figsize'] = (12,8)

**Wczytywanie i odczytywanie danych za pomocą Pandas** → wykorzystaj moduł read\_.

Np.

Read in the data

df = pd.read\_csv(r'C:\Users\Leszek\Downloads\Portfolio\_SQL Data Exploration\movies.csv')

df.head()

**mean** → metoda NumPy. Oblicza średnią arytmetyczną podanych danych (elementów tablicy) wzdłuż określonej osi. Możesz w argumencie podać kolumny, po których dokonujesz iteracji.

Np.

import numpy as np

for col in df.columns:

pct\_missing = np.mean(df[col].isnull())

print('{} - {}%'.format(col, pct\_missing))

**columns** → w Pandas DataFrame.columns to atrybut, który zapewnia dostęp do etykiet kolumn ramki danych. Zwraca obiekt Index reprezentujący nazwy kolumn w DataFrame. Ten atrybut jest używany do wyświetlania, manipulowania lub przypisywania nowych etykiet kolumn do Pandas DataFrame, umożliwiając użytkownikom pracę z określonymi kolumnami w strukturze danych tabelarycznych i odwoływanie się do nich. Atrybut Pandas DataFrame.columns zwraca etykiety kolumn danej ramki danych.

Np.

import pandas as pd

import numpy as np

df = pd.read\_csv(r'C:\Users\Leszek\Downloads\Portfolio\_SQL Data Exploration\movies.csv')

for col in df.columns:

pct\_missing = np.mean(df[col].isnull())

print('{} - {}%'.format(col, pct\_missing))

**dropna** → usuwa wszystkie wiersze z wartościami NULL z DataFrame. W przykładzie używasz pliku .csv o nazwie data.csv (Pandas).

Np.

import pandas as pd

df = pd.read\_csv(r'C:\Users\Leszek\Downloads\Portfolio\_SQL Data Exploration\movies.csv')

df = pd.read\_csv('data.csv')

newdf = df.dropna()

**dtypes** → sprawdza typ danych dla kolumn (Pandas).

Np.

import pandas as pd

df = pd.read\_csv(r'C:\Users\Leszek\Downloads\Portfolio\_SQL Data Exploration\movies.csv')

df.dtypes

**astype** → metoda zmieniająca typ danych (Pandas). Wzór: df[‘kolumna’] = df[‘kolumna’].astype(‘typ\_danych’).

Np.

df['budget'] = df['budget'].astype('int64')

df['gross'] = df['gross'].astype('int64')

**Wyciąganie ciągu liczb za pomocą wzoru** → za pomocą extract i pat. Wyodrębnij ciągi znaków w wyrażeniu regularnym pat jako kolumny w DataFrame. Dla każdego ciągu w serii wyodrębnij znaki z pierwszego dopasowania wyrażenia regularnego pat. Extract w tym kontekście służy do wyciągania danych, pat natomiast do wskazania wzoru zachowania-wyciągania. Jeśli chodzi ci o liczby podaj dla przykładu wzór: pat = '([0-9]{4})'), co oznacza cyfry od 0 do 9, gdzie składają się w 4-cyfrową liczbę.

Np.

df['yearcorrect'] = df['released'].str.extract(pat = '([0-9]{4})').astype(int)

**sort\_values** → Pandas. Wzór: dataframe.sort\_values(by, axis, ascending, inplace, kind, na\_position, ignore\_index, key). Tylko by jest niezbędne, reszta jest opcjonalna. By: po czym sortujesz. Ascending: True/False. Inplace: False zwróci nowe DataFrame, co jest opcjonalne, to znaczy tak się stanie jeśli niczego nie zmienisz.

Np.

df = df.sort\_values(by = ['gross'], inplace = False, ascending = False)

**Wyświetlanie wszystkich danych** → Pandas. Za pomocą pd.set\_option('display.max\_rows', None).

Np.

pd.set\_option('display.max\_rows', None)

df

**Usuwanie duplikatów** → za pomocą drop\_duplicates().

Np.

df.drop\_duplicates()

**Scatterplot** → wykres rozproszony. Przygotowanie tego typu wykresu w prostej wersji wymaga zaimportowania matplotlib.pyplot as plt. Wykorzystaj plt.scatter do określenia osi x = df[‘kolumna’], y = df[‘kolumna’]. Dodaj tytuł-title, nazwę osi x/y-x/ylabel. Pokaż wykres: plt.show.

Np.

import matplotlib.pyplot as plt

plt.scatter(x=df['budget'], y=df['gross'])

plt.title('Budget vs Gross Earnings')

plt.xlabel('Gross Earnings')

plt.ylabel('Budget for Film')

plt.show

**Regplot** → seaborn. Ta metoda jest używana do dopasowania modelu regresji liniowej. Powstanie wykres z kropkami wskazującymi na rozmieszczenie danych w matrycy oraz linią pokazująca kierunek danych. Wzór: sns.regplot(x=’kolumna\_1’, y=’kolumna\_2’, data=df, scatter\_kws czyli kolor kropek={"color": "red"}, line\_kws czyli linia={"color":"blue"}).

Np.

import seaborn as sns

sns.regplot(x='budget', y='gross', data=df, scatter\_kws={"color": "red"}, line\_kws={"color":"blue"})

**Corr** → metoda określa jaka jest korelacja między danymi liczbowymi z poszczególnych kolumn. Tylko dla danych liczbowych. Docelowo wykorzystuje metodę Pearsona i tak będzie jeśli zostawisz tylko nawias. Dodając argument method=’inna\_metoda’, możesz użyć metody kendall albo spearman.

Np.

df.corr(method='pearson')

df.corr(method='kendall')

df.corr(method='spearman')

df.corr()

**Heatmap →** seaborn. Metoda tworzy matrycę, która wskaże wizualnie potencjalne korelacje między danymi z poszczególnych kolumn.

Np.

import seaborn as sns

import matplotlib.pyplot as plt

correlation\_matrix = df.corr()

sns.heatmap(correlation\_matrix, annot=True)

plt.title('Correlation Matric for Numeric Features')

plt.xlabel('Movie Features')

plt.ylabel('Movie Features')

plt.show()

**Cat.codes** → metoda wykorzystywana do nadawania obiektom tekstowym randomowych liczb. Jednocześnie te same wyrażenia będą miały nadane te same kody-liczby. Można wykorzystać w pętli, która najpierw zamienia kolumny mające typ “object” na “category”, a potem użyć cat.codes.

Np.

df\_numerized = df

for col\_name in df\_numerized.columns:

if(df\_numerized[col\_name].dtype == 'object'):

     df\_numerized[col\_name] = df\_numerized[col\_name].astype('category')

     df\_numerized[col\_name] = df\_numerized[col\_name].cat.codes

df\_numerized

**Unstack** → sprawdzanie wysokiej korelacji za pomocą tej metody. jeśli przez wysoką korelację rozumiesz większą niż 0,5 współczynnika korelacji to należy: utworzyć nową zmienną z matrycą korelacji wszystkich, za pomocą metody unstack przekonwertuj tą tabelę na listę połączonych wszystkich wartości osi x i y, sortuj te wartości i ostatecznie wyodrębnij te, które są większe od 0,5.

Np.

correlation\_mat = df\_numerized.corr()

corr\_pairs = correlation\_mat.unstack()

corr\_pairs.sort\_values()

sorted\_pairs = corr\_pairs.sort\_values()

sorted\_pairs

high\_corr = sorted\_pairs[(sorted\_pairs) > 0.5]

high\_corr

**Umieszczanie pliku Jupyter do GitHub** → zamień wszystkie wyświetlenia danych na .head, bo plik zabierze za dużo miejsca.

**import smtplib** → biblioteka pozwalająca na napisanie kodu wysyłającego maile.

**import time** → biblioteka pozwalająca na odmierzanie czasu, np. w pętli, żeby dokonywało się jakieś działanie.

**Span i get\_text** → wyciągając dane ukryte w span wpisz w argument find jako pierwszy “span”, a potem dopiero “class\_”). Reszta to metoda get\_text.

Np.

price = soup2.find('span', class\_ = 'a-offscreen').get\_text()

**Strip (zaawansowane)** → metoda strip poza tym, że usuwa białe znaki, może usuwać specyficzne znaki/ciągi znaków jeśli wpiszesz je do argumenntu.

Np.

price.strip().strip('zł')

**Newline i encoding** → newline sprawia, że przy umieszczaniu danych nie ma przestrzeni między poszczególnymi CSV. Encoding domyślny to UTF8. Uruchamiając dane w Excel, skorzystaj z data, tekst jako kolumny, rozdziel przecinkami.

Np.

import csv

header = ['Title', 'Price', 'Color']

data = [title, price, color]

with open('AmazonWebScraperDataset.csv', 'w', newline='', encoding='UTF8') as f:

writer = csv.writer(f)

writer.writerow(header)

writer.writerow(data)

**Umieszczanie daty powstania danych w tworzonym pliku** → jest to dobra praktyka, taka informacja może się przydać. Wykorzystaj bibliotekę datetime.

Np.

price = price.strip().strip('zł')

title = title.strip()

color = color.strip()

today = datetime.date.today()

print(title)

print(price)

print(color)

print(today)

import csv

header = ['Title', 'Price', 'Color', 'Date']

data = [title, price, color, today]

with open('AmazonWebScraperDataset.csv', 'w', newline='', encoding='UTF8') as f:

writer = csv.writer(f)

writer.writerow(header)

writer.writerow(data)

**Read\_csv** → skorzystaj z tej metody Pandas jeśli nie chcesz za każdym razem otwierać pliku CSV żeby sprawdzać postęp, ale żeby widzieć go w konsoli.

Np.

import pandas as pd

df = pd.read\_csv(r'C:\Users\Leszek\AmazonWebScraperDataset.csv')

print(df)

**Appending the data** → dodawanie danych do istniejącego pliku CSV. W with open wykorzystaj argument “a+” i nie twórz już nagłówków (header). Korzystasz z metody writer.

Np.

import csv

header = ['Title', 'Price', 'Color', 'Date']

data = [title, price, color, today]

with open('AmazonWebScraperDataset.csv', 'w', newline='', encoding='UTF8') as f:

writer = csv.writer(f)

writer.writerow(header)

writer.writerow(data)

with open('AmazonWebScraperDataset.csv', 'a+', newline='', encoding='UTF8') as f:

writer = csv.writer(f)

writer.writerow(data)

**Dodawanie danych automatycznie do x czasu** → w pierwszej kolejności zdefiniuj cały kod od podawania bibliotek do polecenia dodającego, append, dane do pliku CSV. W następnej kolejności puść pętlę while, która co x sekund, gdyż wartość podaje się w sekundach, będzie dodawać dane. Pamiętaj, że definiując z def wszystko musi znaleźć się stopień niżej.

Np.

def check\_price():

from bs4 import BeautifulSoup

import requests

import smtplib

import time

import datetime

import pandas as pd

import csv

URL = 'https://www.amazon.pl/Power-Czarna-Koszulka-Unisex-Lu%C5%BAnym/dp/B0DMWK9X5W?crid=1PFQJAR99AY1D&dib=eyJ2IjoiMSJ9.BZ9GU-e\_nsJbusSCS57UmPZ9LgdIE1mriB5uekB5C9vZrM2EwrzkklOO8-AOgysOI7POmCt2KyFnKYq8vx66lhspeRTHbz-9IirxS\_pxKI\_afACI2kIOu2j4GdaO2cqbkMJieIvzDH\_cMPUciM7DCEuAaPvD98ye95ptGwIiyUaGsZwp-KuB7zDSglrorPCgKKLm\_WGs2ou5Iknlmv7KL3RE89mUWIMht7BUUmqeqf\_zRhu93xM1iCH\_M8NbJRWkuS9C6HOaIyIXI6FGkGZeMa7X24scDnicE563oCKC9DM.0mirq3QlKRbNdCvYdrSMGvHfOqZ7eyYhNxkadFQ5QaY&dib\_tag=se&keywords=got+data+shirt&qid=1732944907&sprefix=got+data%2Caps%2C90&sr=8-12'

headers = {"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:124.0) Gecko/20100101 Firefox/124.0"}

page = requests.get(URL, headers=headers)

soup1 = BeautifulSoup(page.content, "html.parser")

soup2 = BeautifulSoup(soup1.prettify(), "html.parser")

title = soup2.find(id='productTitle').get\_text()

price = soup2.find('span', class\_ = 'a-offscreen').get\_text()

color = soup2.find('span', class\_ = 'selection').get\_text()

price = price.strip().strip('zł')

title = title.strip()

color = color.strip()

today = datetime.date.today()

today = datetime.date.today()

header = ['Title', 'Price', 'Color', 'Date']

data = [title, price, color, today]

with open('AmazonWebScraperDataset.csv', 'a+', newline='', encoding='UTF8') as f:

     writer = csv.writer(f)

     writer.writerow(data)

while(True):

check\_price()

time.sleep(86400)

**Wysyłanie maila w pętli jeśli spełniony warunek** → podłączanie się pod maila, w przykładzie gmail, logowanie się za pomocą loginu i hasła, pisanie tematu, zawartości.

Np.

def check\_price\_mail():

from bs4 import BeautifulSoup

import requests

import smtplib

import time

import datetime

import pandas as pd

import csv

URL = 'https://www.amazon.pl/Power-Czarna-Koszulka-Unisex-Lu%C5%BAnym/dp/B0DMWK9X5W?crid=1PFQJAR99AY1D&dib=eyJ2IjoiMSJ9.BZ9GU-e\_nsJbusSCS57UmPZ9LgdIE1mriB5uekB5C9vZrM2EwrzkklOO8-AOgysOI7POmCt2KyFnKYq8vx66lhspeRTHbz-9IirxS\_pxKI\_afACI2kIOu2j4GdaO2cqbkMJieIvzDH\_cMPUciM7DCEuAaPvD98ye95ptGwIiyUaGsZwp-KuB7zDSglrorPCgKKLm\_WGs2ou5Iknlmv7KL3RE89mUWIMht7BUUmqeqf\_zRhu93xM1iCH\_M8NbJRWkuS9C6HOaIyIXI6FGkGZeMa7X24scDnicE563oCKC9DM.0mirq3QlKRbNdCvYdrSMGvHfOqZ7eyYhNxkadFQ5QaY&dib\_tag=se&keywords=got+data+shirt&qid=1732944907&sprefix=got+data%2Caps%2C90&sr=8-12'

headers = {"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:124.0) Gecko/20100101 Firefox/124.0"}

page = requests.get(URL, headers=headers)

soup1 = BeautifulSoup(page.content, "html.parser")

soup2 = BeautifulSoup(soup1.prettify(), "html.parser")

title = soup2.find(id='productTitle').get\_text()

price = soup2.find('span', class\_ = 'a-offscreen').get\_text()

color = soup2.find('span', class\_ = 'selection').get\_text()

price = price.strip().strip('zł')

title = title.strip()

color = color.strip()

today = datetime.date.today()

today = datetime.date.today()

header = ['Title', 'Price', 'Color', 'Date']

data = [title, price, color, today]

with open('AmazonWebScraperDataset.csv', 'a+', newline='', encoding='UTF8') as f:

     writer = csv.writer(f)

     writer.writerow(data)

if(price < 100):

     send\_mail()

while(True):

check\_price\_mail()

time.sleep(86400)

def send\_mail():

server = smtplib.SMTP\_SSL('smtp.gmail.com',465)

server.ehlo()

#server.starttls()

server.ehlo()

server.login('dekiert.leszek@gmail.com','xxxxxxxxxxxxxx')

subject = "T-shirt jest teraz tańszy niż 100 zł!\_cena"

body = "T-shirt jest teraz tańszy niż 100 zł!"

msg = f"Temat: {subject}\n\n{body}"

server.sendmail(

     'dekiert.leszek@gmail.com',

     msg

)

**API** → Application Programming Interface, Interfejs programowania aplikacji jest to zbiór reguł ściśle opisujący, w jaki sposób programy lub podprogramy komunikują się ze sobą. API ma klienta i serwer. Osoba zbierająca dane to serwer, a aplikacja, która odpowiada na zapytanie to klient. API stanowi pośrednika między użytkownikiem, a posiadaczem danych. API to zbiór reguł i danych, do których można mieć dostęp jako użytkownik. Metafora: idziesz do rzeźnika i wybierasz co chcesz, po czym rzeźnik idzie za ladę i wybiera to dla ciebie. Ty nie masz dostępu za tym co za ladą, widzisz tylko to co jest dostępne. Jeśli strona pozwala na takie działanie (w przykładzie <https://coinmarketcap.com/>), to będzie udostępnione chociażby na samym dole. Dobrze jest zwrócić uwagę na dokumentację API, bo nie każdy dostęp i zasady są jednakowe. Możliwe, że udostępniony jest kod, którym można wyciągać dane.

**Uruchamianie API** → wykorzystaj kod udostępniony przez stronę, z której czerpiesz dane. Podmień w kodzie klucz, który uzyskujesz ze strony. Patrz dalej w instrukcję, wykorzystaj w kodzie sugerowaną domenę, ale nie podmieniaj całości: podmień '<https://sandbox-api.coinmarketcap.com/v1/cryptocurrency/listings/latest>' na '<https://pro-api.coinmarketcap.com/v1/cryptocurrency/listings/latest>'.

**Błąd IOPub data rate exceeded** → należy zwiększyć “data rate limit”. Znajdź i uruchom program “Anaconda prompt”. Wpisz kod zwiększający data rate: jupyter notebook --NotebookApp.iopub\_data\_rate\_limit=1e10. Potem w komunikacie znajdź informację: “To access the notebook, open this file in a browser” i uruchom podany link.

**Formatowanie danych** → dane pojawią się w różnym formacie. W przykładzie to format json jako biblioteki zagnieżdżone. Wykorzystaj metodę normalize z biblioteki pandas.

Np.

import pandas as pd

pd.json\_normalize(data['data'])

**Wszystkie kolumny** → żeby zobaczyć wszystkie kolumny należy skorzystać z metody pandas max\_columns. A potem wyświetlić dane.

Np.

import pandas as pd

pd.set\_option('display.max\_columns', None)

pd.json\_normalize(data['data'])

**Timestamp** → przygotowując automatyzację warto dodać kolumnę z informacją kiedy te dane został wyciągnięte.

Np.

df = pd.json\_normalize(data['data'])

df['timestamp'] = pd.to\_datetime('now')

df

**Ustawianie limitu w przykładzie** → w przekazanym przez źródło kodzie. Mień “top” na cokolwiek chcesz.

**Skrypt automatyzujący pozyskiwanie i zapisywanie danych** → zdefiniuj funkcję dodającą dane do obiektu dataframe (df) za pomocą skryptu źródła danych oraz połączenia normalize/to\_datetime/append. Zaimportuj bibliotekę os razem z funkcjami time/sleep i przygotuj pętlę uruchamiającą kod x razy za pomocą range oraz zatrzymującą działanie pętli na x czasu za pomocą sleep w sekundach. Pamiętaj o dodaniu wszystkich bibliotek do zdefiniowanej funkcji, z której korzystasz. Musisz też od razu pod def oznaczyć df, czyli dataframe, czyli zmienną do której umieszczasz dane jako global.

Np.

def api\_runner():

global df

from requests import Request, Session

from requests.exceptions import ConnectionError, Timeout, TooManyRedirects

import json

url = 'https://pro-api.coinmarketcap.com/v1/cryptocurrency/listings/latest'

parameters = {

   'start':'1',

   'limit':'15',

   'convert':'USD'

}

headers = {

   'Accepts': 'application/json',

   'X-CMC\_PRO\_API\_KEY': 'e9369a3d-e46d-42d0-819c-ef8f78b01df0',

}

session = Session()

session.headers.update(headers)

try:

   response = session.get(url, params=parameters)

   data = json.loads(response.text)

   print(data)

except (ConnectionError, Timeout, TooManyRedirects) as e:

   print(e)

import pandas as pd

pd.set\_option('display.max\_columns', None)

pd.set\_option('display.max\_rows', None)

pd.json\_normalize(data['data'])

df = pd.json\_normalize(data['data'])

df['timestamp'] = pd.to\_datetime('now')

df2 = pd.json\_normalize(data['data'])

df2['Timestamp'] = pd.to\_datetime('now')

df = df.append(df2)

df

import os

from time import time

from time import sleep

for i in range(100):

api\_runner()

print('API Runner completed')

sleep(10)

exit()

**Skrypt automatyzujący i umieszczający dane w CSV** → podobny do powyższego z tym, że na końcu sprawdza czy istnieje plik CSV z tymi danymi, a jeśli tak to dodaje nowe dane do istniejącego pliku. Jeśli nie, to stworzy ten plik. Zmień też kod dokonujący operację dodawania danych, nie jest już ci potrzebna zmienna df2 ani nawet metoda append, ponieważ to się dzieje w klauzuli if. Możesz dodać kod czytający w konsoli zawartość pliku CSV.

Np.

def api\_runner():

global df

from requests import Request, Session

from requests.exceptions import ConnectionError, Timeout, TooManyRedirects

import json

url = 'https://pro-api.coinmarketcap.com/v1/cryptocurrency/listings/latest'

parameters = {

   'start':'1',

   'limit':'15',

   'convert':'USD'

}

headers = {

   'Accepts': 'application/json',

   'X-CMC\_PRO\_API\_KEY': 'e9369a3d-e46d-42d0-819c-ef8f78b01df0',

}

session = Session()

session.headers.update(headers)

try:

   response = session.get(url, params=parameters)

   data = json.loads(response.text)

   print(data)

except (ConnectionError, Timeout, TooManyRedirects) as e:

   print(e)

import pandas as pd

pd.set\_option('display.max\_columns', None)

pd.set\_option('display.max\_rows', None)

pd.json\_normalize(data['data'])

df = pd.json\_normalize(data['data'])

df['timestamp'] = pd.to\_datetime('now')

df = pd.json\_normalize(data['data'])

df['Timestamp'] = pd.to\_datetime('now')

df

if not os.path.isfile(r'C:\Users\Leszek\Downloads\Portfolio\_Alex the Analyst\API.csv'):

     df.to\_csv(r'C:\Users\Leszek\Downloads\Portfolio\_Alex the Analyst\API.csv', header='column\_names')

else:

     df.to\_csv(r'C:\Users\Leszek\Downloads\Portfolio\_Alex the Analyst\API.csv', mode='a', header=False)

import os

from time import time

from time import sleep

for i in range(100):

api\_runner()

print('API Runner completed')

sleep(10)

exit()

import pandas as pd

df\_read = pd.read\_csv(r'C:\Users\Leszek\Downloads\Portfolio\_Alex the Analyst\API.csv')

df\_read

**Prezentowanie całych wartości zamiast fragmentów** → jeśli chcę widzieć całe wartości w konsoli zamiast ich skróconej wersji użyję wyrażenia lambda.

Np.

import pandas as pd

pd.set\_option('display.float\_format', lambda x: '%.5f' % x)

**Grupowanie zebranych wyników w Python** → za pomocą metody groupby. Grupuj za pomocą wybranej zmiennej różnicującej, chociażby nazwie, ustal czy chcesz sortować dane. W podwójnym nawiasie kwadratowym umieść kolumny, które chcesz zaobserwować oraz zastosuj na nich sposób grupowania, chociażby średnią lub medianę.

Np.

df\_group = df.groupby('name', sort=False) [['quote.USD.percent\_change\_1h', 'quote.USD.percent\_change\_24h', 'quote.USD.percent\_change\_7d', 'quote.USD.percent\_change\_30d', 'quote.USD.percent\_change\_60d', 'quote.USD.percent\_change\_90d']].mean()

df\_group

**Piwot danych i indeksowanie** → żeby skorzystać z danych w przykładzie, w formie wizualizacji należy ułożyć wartości w wierszach zamiast kolumnach. Dokonanie piwotu wymaga podjęcia kroków: skorzystania z metody stack, żeby ułożyć dane jedna pod drugą; zamienić obiekt na typ dataframe za pomocą metody to\_frame, ponieważ tylko z tego obiektu możliwe będzie przygotowanie wizualizacji; sprawdzenie ilości wartości w obiekcie; przygotowanie wartości indeksu, która będzie zawierać ilość wartości w obiekcie; nadanie indeksu wartościom; zresetowanie indeksów dla całego dataframe. Możesz zmienić nazwę nagłówka jednej lub więcej kolumn.

Np.

df\_stack = df\_group.stack()

df\_stack

df\_df = df\_stack.to\_frame(name='values')

df\_df

df\_df.count()

import pandas as pd

df\_index = pd.Index(range(90))

df\_df\_values = df\_df.set\_index(df\_index)

df\_df\_values

df\_df\_values = df\_df.reset\_index()

df\_df\_values

df\_complete = df\_df\_values.rename(columns={'level\_1': 'percent\_change'})

df\_complete

**Zmiany długości tekstu i wizualizacja** → przed wizualizacją warto przygotować dane, tak aby wartości nie były za długie, też w legendzie. Za pomocą metody replace zamień za długie wartości na krótsze. Skorzystaj z biblioteki seaborn oraz matplotlib żeby przygotować wykres liniowy z kropkami. Wybierz co będzie w osi x i y. Co będzie w legendzie, z jakich danych skorzystasz oraz czy na liniach mają być kropki.

Np.

df\_complete['percent\_change'] = df\_complete['percent\_change'].replace(['quote.USD.percent\_change\_1h', 'quote.USD.percent\_change\_24h', 'quote.USD.percent\_change\_7d', 'quote.USD.percent\_change\_30d', 'quote.USD.percent\_change\_60d', 'quote.USD.percent\_change\_90d'],['1h', '24h', '7d', '30d', '60d', '90d'])

df\_complete

import seaborn as sns

import matplotlib.pyplot as plt

sns.catplot(x='percent\_change', y='values', hue='name', data=df\_complete, kind='point')

**Wykres liniowy wybranego elementu** → jeśli chcesz zaprezentować wykres zmian tylko jednego elementu, możesz go wyciągnąć zapytaniem, czyli query, które piszesz w podwójnym cudzysłowiu.

Np.

df\_bitcoin = df[['name', 'quote.USD.price','timestamp']]

df\_bitcoin = df\_bitcoin.query("name == 'Bitcoin'")

df\_bitcoin

sns.set\_theme(style="darkgrid")

sns.lineplot(x='timestamp', y='quote.USD.price', data = df\_bitcoin)