WPROWADZENIE DO JĘZYKA PYTHON

Opracowanie: Krzysztof Ropiak

Fragmenty kodu zaprezentowane w materiałach są zgodne z 32-bitową wersją Python 3.5.0

SPIS TREŚCI

1. HISTORIA PYTHONA

Twórcą Pythona jest holender Guido van Rossum a sama nazwa, co niektórych zapewne nie dziwi, pochodzi od popularnego serialu BBC „Latający Cyrk Monty Pythona". Prace nad pierwszym interpreterem Pythona rozpoczęły się w 1989 roku jako następca języka ABC. Wszystkie wersje aż do 1.2 powstawały w CWI (Centrum Matematyki i Informatyki) w Amsterdamie gdzie Guido wówczas pracował. Od wersji 2.1Python był udostępniany jako projekt Open Source przez niedochodową organizację Python Software Foundation (PSF). Obecnie nad rozwojem Pythona pracuje wiele osób, ale Guido wciąż jest zaangażowany w ten proces. Sam twórca w 1995 roku wyemigrował do USA gdzie w latach 2005 – 2013 pracował dla Google a obecnie pracuje dla firmy Dropbox.

Ważnym momentem w historii Pythona było utworzenie drugiej głównej gałęzi – Pythona 3 w roku 2008. Od tego momentu wersja 2 oraz 3 były rozwijane oddzielnie, ale czas wersji 2 zaczyna mijać o czym świadczy ogłoszony już termin zakończenia wsparcia na 12 kwietnia 2020 roku. Rozwój języka jest prowadzony przy wykorzystaniu PEP (Python Enhancement Proposal). Dokumenty te to propozycje rozszerzeń lub zmian w języku w postaci artykułu, który jest poddawany pod dyskusję wśród programistów Pythona. Każdy dokument zawiera opis proponowanego rozwiązania, uzasadnienie oraz aktualny status. Po osiągnięciu konsensusu propozycje są przyjmowane lub odrzucane.

2. WYBRANE CECHY JĘZYKA PYTHON

Python jest językiem ogólnego przeznaczenia, którego ideą przewodnią jest czytelność i klarowność kodu źródłowego. Standardowa implementacja języka jest CPython (napisany w C), ale istnieją też inne, np. Jython (napisany w Javie), CLPython napisany w Common Lisp, IronPython (na platformę .NET) i PyPy napisany w Pythonie.

Python nie wymusza jednego stylu programowania dając możliwość programowania obiektowego, programowania strukturalnego oraz programowania funkcyjnego.

Inne cechy języka Python:

- Typy sprawdzane są dynamicznie (w przeciwieństwie np. do Javy),
- Do zarządzania pamięcią używany jest garbage collector,
- Wszystkie wartości przekazywane są przez referencję,
- Jest czasem kwalifikowany jako język skryptowy,
- Nie ma enkapsulacji, jak to ma miejsce w C++ czy Javie, ale istnieją mechanizmy, które pozwalają na osiągnięcie podobnego efektu,
- Możliwe jest tworzenie funkcji ze zmienną liczbą argumentów,
- Możliwe jest tworzenie funkcji z argumentami o wartościach domyślnych.

3. PYTHON I DATA SCIENCE

Python nie jest jedynym ani też jednoznacznie najlepszym językiem dla Data Science. Jego największym konkurentem w tej dziedzinie jest R, który od samego początku był tworzony z myślą o statystyce, która jak wiadomo w dziedzinie sztucznej inteligencji ma szerokie zastosowanie. Trwają niezliczone spory i porównania próbujące udowodnić wyższość jednego rozwiązania nad drugim. Skoro jednak będziemy zajmować się Pythonem przytoczę kilka jego zalet pod kątem Data Science:

a) Python jest **językiem ogólnego przeznaczenia** co powoduje, że oprócz możliwości wykorzystania specjalistycznych bibliotek np. pod Data Science można bez konieczności integracji z innymi rozwiązaniami **zbudować kompletną aplikację desktopową lub webową**,

b) Python jako **język skryptowy**, i dodatkowo w połączeniu z **Jupyter** (**IPython) notebook**, pozwala na bardzo szybkie testowanie i prototypowanie poprzez pisanie kodu „na bieżąco", co powoduje brak konieczności kompilowania kodu i jego późniejszego uruchamiania co znacznie przyspiesza proces stworzenia działającego rozwiązania,

c) **Bogata paleta bardzo dobrej jakości bibliotek** dla AI (Artificial Intelligence) i DS (Data Science). Za przykład mogą tutaj posłużyć **NumPy**, **Pandas**, **SciPy, matplotlib** czy **scikit-learn**, które zostaną zaprezentowane na zajęciach „Zaawansowany Python".

d) Społeczność – jako język ogólnego zastosowania społecznośćs Pythona jest bardzo duża co przekłada się na łatwość uzyskania odpowiedzi na pytania, sporą ilość dobrej dokumentacji oraz rozbudowaną listę bibliotek i dodatków.

Polecam dość zwięzły artykuł, który rzuci nieco światła na to dlaczego warto używać Pythona w Data Science oraz czego wciąż w nim brakuje, aby mógł stać się jeszcze lepszym narzędziem.

https://www.quora.com/Why-is-Python-a-language-of-choice-for-data-scientists

4. PREZENTACJA ŚRODOWISKA PYCHARM COMMUNITY EDITION

W trakcie zajęć prezentacja przykładów kodu oraz ćwiczenia będą wykonywane z wykorzystaniem środowiska PyCharm Community, które jest darmową wersją tego narzędzia. Wersja Community pozbawiona jest wielu udogodnień, które mogą przydać się na dalszym etapie pracy z językiem Python np. w trakcie wytwarzania aplikacji webowych z wykorzystaniem m.in. Django, Flask, JavaScript oraz wielu innych. Mamy natomiast do dyspozycji „inteligentny" edytor, graficzny debugger, inspekcję kodu, wsparcie dla systemów kontroli wersji oraz pakiet narzędzi naukowych.

Bardziej szczegółowe porównanie można znaleźć pod adresem:

https://www.jetbrains.com/pycharm/features/editions_comparison_matrix.html

Firma JetBrains, która jest autorem m.in. oprogramowania PyCharm oferuje pełen pakiet oprogramowania w wersji Professional za darmo dla studentów oraz nauczycieli. Aby taką licencję uzyskać należy się zarejestrować i aplikować o przyznanie takiej licencji. Jedyne ograniczenie to wykorzystywanie tego oprogramowania do celów komercyjnych. Więcej tutaj: https://www.jetbrains.com/student/ .

5. ORGANIZACJA KODU WEDŁUG PEP8

Jak zostało już wspomniane w 1 zmiany w specyfikacji języka odbywają się poprzez system PEP. Dokument o numerze PEP8 jest jedną (ale nie jedyną) propozycją organizacji kodu języka Python. Oryginalna treść dokumentu dostępna jest pod adresem https://www.python.org/dev/peps/pep-0008/.

5.1. WCIĘCIA

W kodzie języka Python nie znajdziemy znanych z PHP, Javy czy C# klamerek do separacji bloków kodu, określania ram ciała metody czy klasy lub zakresu operacji w pętli. Tutaj do tego celu służą odpowiednio ustawione wcięcia i puste linie między w/w elementami. Dla osób, które nigdy wcześniej nie miały do czynienia z taką organizacją kodu może to być zaskakujące, ale dość szybko staje się zrozumiałe i intuicyjne.

Przykład:

```
if score >= 100:
    print("Zwycięstwo !")
```

Każdy kolejny poziom zagnieżdżenia w bloku kodu poprzedza odstęp w postaci wielokrotności 4 spacji (pojedyncza wartość wcięcia). Dopuszczalne jest również stosowanie tabulatorów jako wcięć, ale zalecane są spacje a dodatkowo w wersji Python 3 użycie jednocześnie spacji i tabulatorów jako wcięć nie jest dozwolone.

Wcięcia używamy również w sytuacjach, w których linia kodu jest zbyt długa i musi być złamana na większą ilość wierszy. Zalecana długość linii według PEP8 to 79 znaków.

Przykład:

```
wyslane = wyslij_wiadomosc(e_mail_odbiorcy, temat,
                           wiadomosc)
```

W takim przypadku wcięcie sięga znaku otwarcia deklaracji listy atrybutów wywoływanej metody lub funkcji. Dopuszczalne są jednak odstępstwa od tej reguły pozwalające na zastosowanie mniejszego wcięcia dla kolejnych linii.

Przykład:

```
wyslane = wyslij_wiadomosc(e_mail_odbiorcy,
    temat, wiadomosc)
```

Jednak sama dokumentacja mówi o tym, że jest to opcjonalne formatowanie, więc należy go używać tylko z konieczności a nie jako regułę.

Deklaracja zmiennych takich jak lista, tablica, krotka czy słownik dzięki wcięciom często poprawia ich czytelność co jest główną zasadą, która kierowano się określając reguły formatowania kodu w Pythonie.

Przykład:

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

W przypadku łamania linii i operatorów np.. arytmetycznych obowiązuje zasada przenoszenia operatora do nowej linii.

Przykład:

```
zysk = (przychod
        - koszty
        - podatki)
```

5.2. PUSTE LINIE

Funkcje najwyższego rzędu oraz definicje klas oddzielamy od pozostałych bloków kodu dwiema pustymi liniami.

Przykład:

```
def zrob_cos():
    return "zrobione"


def tez_cos_zrob():
    return "też zrobione"
```

Metody klas oraz funkcje lokalne oddzielone są natomiast pojedynczą pustą linią.

Przykład:

```
class Osoba:

    def __init__(self, imie, nazwisko, plec):
        self.imie = imie
        self.nazwisko = nazwisko
        self.plec = plec

    def przedstaw_sie(self):
        print("Nazywam się {0} {1}".format(self.imie, self.nazwisko))

    def moj_wiek(self):
        print("Moja płeć to: {0}.".format(self.plec))


os = Osoba("Krzysztof", "Ropiak", "mężczyzna")
os.przedstaw_sie()
os.moj_wiek()
```

Przykład:

```
def funkcja_top_level():

    def funkcja_lokalna():
        pass

    def kolejna_funkcja_lokalna():
        pass
```

Pojedyncze puste linie mogą być również stosowane wewnątrz funkcji aby odseparować od siebie logiczne sekcje funkcji.

5.3. IMPORT

Poszczególne instrukcje importu powinny być rozdzielone na oddzielne linie.

Przykład:

Tak

```
import os
import sys
```

Nie

```
import sys,os
```

Poprawny jest natomiast taki sposób definiowania importu:

```
from subprocess import Popen, PIPE
```

Inne zasady dotyczące organizacji importów.

Importy powinny być umieszczane na początku pliku tuż za ewentualnymi komentarzami dla modułu i elementami docstring. Kolejnośc importów ma również znaczenie. Oto zalecany porządek:

- import bibliotek standardowych
- import powiązanych bibliotek zewnętrznych (ang. third party imports)
- import lokalnych aplikacji/bibliotek

Zalecane jest również dodawanie pustej linii po każdej z w/w grup importów. Jako, że Python umożliwia zarówno import całej biblioteki lub tylko wybranych jej modułów często trzeba dobrać odpowiedni sposób do sytuacji, ale z reguły zaleca się wykonywanie importu i dodanie aliasu lub import modułu zamiast np. konkretnej klasy z tego modułu co zmniejsza ryzyko wystąpienia konfliktów w przestrzeni nazw. Więcej informacji oraz przykłady znajdują się w rozdziale poświęconym zarządzaniu i importowi pakietów.

5.4. INNE ZALECENIA

1. Zmienne typu string można umieszczać zarówno w cudzysłowie lub w apostrofach, gdyż w przypadku Pythona nie ma to znaczenia. Natomiast PEP8 nie zaleca żonglowania tym zapisem i trzymania się jednej z opcji. Sytuacją, w której dozwolone jest użycie obu jednocześnie jest ciąg tekstowy, który sam już zawiera cudzysłów lub apostrof – wtedy należy użyć drugiego z nich.

   Przykład:

```
artykul = 'Recenzja "Władcy Pierścieni".'
```

Ale można również tak:

```
artykul = "Recenzja \"Władcy Pierścieni\"."
```

lub tak

```
artykul = """Recenzja "Władcy Pierścieni"."""
```

2. Spacje w wyrażeniach i definicjach są pożądane, ale nie należy ich nadużywać.

   Przykład:

```
dobrze: zakupy(szynka[1], {jajka: 2})
        x = 1
        lista[index]
        lista[1:4]

źle:    zakupy( szynka[ 1 ], { jajka: 2 } )
        x=1
        lista [index]
        lista[1: 4]
```

Wszystkie operatory binarne powinny być otoczone pojedynczą spacją.

Ćwiczenia

Zasad, które opisane są w dokumencie PEP8 jest więcej, ale nie wszystkie będą tutaj przytoczone. Poświęć ok. 15 minut na zapoznanie się z dokumentem PEP8 (link na początku rozdziału) i przejrzyj pokazane tam przykładowe fragmenty kodu. Zwróć uwagę na informacje w sekcji „Comments" oraz „Naming Conventions".

Na sam koniec warto dodać, że edytor środowiska PyCharm posiada zaimplementowany mechanizm formatowania kodu wg. PEP8 więc nawet jak złamiemy zasadę wcięć, odstępów lub inną, zostaniemy o tym fakcie poinformowani co umożliwi dokonanie poprawek lub skorzystanie z automatycznego formatowania. Czasem jednak automatyczne formatowanie potrafi się pogubić…

6. PODSTAWOWE TYPY DANYCH

Zanim przejdziemy do omawiania poszczególnych typów danych warto wiedzieć, że Python jest językiem „typowanym dynamicznie". Oznacza to, że typ danych jaki zostanie wykorzystany do przechowania wartości przypisanej do zmiennej często zależy od wartości jaka zostanie do zmiennej przypisana co znacznie różni się od sposobu w jaki typy są przypisywane do zmiennych w Javie czy C++.

Takie rozwiązanie ma zarówno wady jak i zalety. Do wad można zaliczyć to, że pierwotny typ zmiennej może ulec zmianie w dalszej części kodu co wymusza na programiście większą kontrolę tego co dzieje się z tą zmienną i czasami trzeba stosować funkcje, które sprawdzają typ przekazanych danych. Nie możemy też w żaden sposób wymusić przekazania do metody danych określonego typu lub określić jaki typ danych zostanie zwrócony. Zaletą dynamicznego typowania jest większa elastyczność języka i możliwość zmiany typu w locie co eliminuje konieczność jawnego deklarowania nowych zmiennych do przechowywania danych pod postacią innego typu (rzutowanie jawne i niejawne).

Kolejna istotna informacja jest taka, że Python jest językiem zorientowanym obiektowo i wszystko w Pythonie jest obiektem* o czym świadczy chociażby to, że właściwie wszystkie zmienne posiadają metody, które można na nich wykonać.

* to stwierdzenie może nie być prawdą, jeżeli porównać definicje obiektu w innych językach programowania, ale z punktu widzenia Pythona i jego twórców jest prawdą.

6.1. KILKA SŁÓW O OPERATORACH

Zanim omówione zostaną typy danych warto poznać kilka operatorów, które w powiązaniu ze zmiennymi są często używane.

```
# operatory arytmetyczne
suma = 1 + 2 * 3 / 4.0

# modulo czyli reszta z dzielenia
reszta = 12 % 5

kwadrat = 5 ** 2
szescian = 5 ** 3

# operacje na napisach
full_name = "Krzysztof" + " " + "Ropiak"
# ale to ?
spam = "SPAM " * 10
print(spam)

# listy
oceny = [1, 2, 3, 4, 5] * 10
print(oceny)

# operatory porównania
liczba1 = 1
liczba2 = 2

print(liczba1 > liczba2)
print(liczba1 <= liczba2)
print(liczba1 == liczba2)
print(liczba1 != liczba2)

# powyższe porównania zwrócą wartości logiczne czyli True lub False
# na wartościach logicznych możemy również wykonywac operacje

prawda = True
falsz = False

print(prawda and falsz)
print(prawda or falsz)
print(not prawda)
print(not not prawda)
print(bool(prawda or falsz))
```

Python w bardziej złożonych wyrażeniach wykonuje działania w określonej kolejności:

1. najpierw **
2. następnie *, / oraz %
3. a dopiero na końcu + i -

W Pythonie jako fałsz traktowane są:

- liczba zero (0, 0.0, 0j itp.)
- False
- None (null)
- puste kolekcje ([], (), {}, set() itp.)
- puste napisy
- w Pythonie 2 – obiekty posiadające metodę __nonzero__(), jeśli zwraca ona False lub 0
- w Pythonie 3 – obiekty posiadające metodę __bool__(), jeśli zwraca ona False

6.2. TYPY LICZBOWE

Dwa główne typy liczbowe Pythona to liczba całkowita oraz rzeczywiste czyli `integer` i `float`. Jest jeszcze typ complex, który służy do przechowywania wartości liczb zespolonych, ale zapoznanie się z informacjami na jego temat pozostawiam czytelnikowi.

```
całkowita = 5
rzeczywista = 5.6
rzeczywista = float(56)
# powyższy sposób to rzutowanie
# poniżej kolejny przykład
liczba_str = "123"
liczba = int(liczba_str)
print(type(liczba))


# zmienne można również zadeklarować w jednej linii
a, b = 3, 4
```

W przypadku liczb rzeczywistych można również określić precyzję z jaką zostaną wyświetlone, ale stosowny przykład znajduje się w kolejnym podrozdziale.

6.3. TYP TEKSTOWY

We fragmentach kodu w poprzednich rozdziałach znalazło się już kilka przykładów deklaracji zmiennej typu string. Dla przypomnienia:

```
artykul = """Recenzja "Władcy Pierścieni"."""
imie = 'Krzysztof'
hobby = "piłka nożna"
```

Powyższy fragment to tylko przykład różnych metod deklaracji, w trakcie zajęć będzie stosowany sposób zalecany przez PEP8 czyli cudzysłów.

Ciąg tekstowy w Pythonie to tablica znaków co daje z miejsca wiele możliwości manipulacji i dostępu do składowych tego ciągu. Inna ważna cecha stringów to fakt, że po ich zadeklarowaniu nie możemy zmienić zadeklarowanych znaków ciągu. Oczywiście możemy nadpisać zmienną nową wartością lub dokleić do niej kolejny ciąg tekstowy, ale pierwotny fragment jest niezmienny. Poniżej kilka przykładów.

```
imie = "Krzysztof"
nazwisko = "Ropiak"

# string to tablica znaków więc możemy odwołać się do jej elementów
print(imie[0])

# indeks elementu możemy również określać jako pozycja od końca ciągu
print(imie[-1])

# można również pobrać fragment ciągu (slice) określając jako indeks
# element początkowy i końcowy. Zwróć uwagę na wartość tych indeksów.
print(imie[0] + imie[-2] + imie[4:6])
# można również określic tylko jeden z dwóch indeksów
print(imie + nazwisko[3:])

# inny sposób złączania ciągów
print(imie + " " + nazwisko)

# Elementów ciągu nie można zmieniać więc poniższa instrukcja zwróci błąd.
# nazwisko[0] = "P"

# Potwierdzeniem tego, że ciąg tekstowy jej również obiektem jest możliwość
# wykonania na nim metod dla tego typu zdefiniowanych. Metoda count() zlicza
# ilość wystąpień danego ciągu w wartości przechowywanej przez zmienną.
print(imie.count("z"))
# Co ciekawe w Pythonie możemy wywoływać funkcje dla danego obiektu już podczas
deklaracji
# co na pierwszy rzut oka może wyglądać dość egzotycznie.
print("Jesteś szalona!".count("a"))

# Potwierdzeniem niezmienności zadeklarowanego stringa może być również poniższy
kod
print(imie.lower())
print(imie)

# Aby zwrócić długość ciągu tekstowego należy posłużyć się wbudowaną funkcją
len()
print(len(nazwisko))
```

Ciągi tekstowe bardzo często są „dekorowane” innymi wartościami przed ich wydrukowaniem na konsolę. Ciąg wyjściowy może być zlepkiem innych ciągów i/lub liczb, ale nie możemy tak po prostu wykonać poniższej operacji:

```
print("Ala ma " + 4 + " lata")
```

Interpreter Pythona zwróci błąd z informacją, że nie może wykonać niejawnej konwersji z typu int na typ string. Poniżej listing z możliwościami jakie mamy do dyspozycji.

```
# formatowanie znane z Pythona 2.x
wyznanie = "Lubię %s" % "Pythona"
print(wyznanie)

wonsz = "Python"
print("Lubię %sa" % wonsz)

print("Lubię %s oraz %sa" % ("Pythona", wonsz))

# %s oznacza, że w miejsce tego znacznika będzie podstawiany ciąg tekstowy
# %i - to liczba całkowita
# %f - liczba rzeczywista lub inaczej zmiennoprzecinkowa
# %x lub #X - liczba całkowita zapisana w formie szesnastkowej

print("Używamy wersji Python %i" % 3)
print("A dokładniej Python %f" % 3.5)
print("Chociaż lepiej to zapisać jako Python %.1f" % 3.5)
print("A kolejną główną wersją Pythona może być wersja %.4f" % 3.6666)
print("A może będzie to wersja %.1f ?" % 3.6666)
print("A może jednak %.f ?" % 3.6666)
wersja = 4
print("A %i w systemie szesnastkowym to %X" % (wersja, wersja))
print("A %i * %i szesnastkowo daje %X" % (wersja, wersja, wersja*wersja))

# Chociaż możliwości przy korzystaniu z mechanizmów powyżej są spore,
# to i kilka wad się również znajdzie. Trzeba pilnować zarówno liczby argumentów
jak
# i ich kolejności. Konieczne jest powielanie tej samej zmiennej jeżeli kilka
# razy jest wykorzystywana w formatowanym ciągu. Spójrzmy na inne możliwości.

print("Lubię %(jezyk)s" % {"jezyk": "Pythona"})
print("Lubię %(jezyk)s a czy Ty lubisz %(jezyk)s ?" % {"jezyk": "Pythona"})
# wadą jest dość duża ilość dodatkowego kodu do napisania, ale nazwy zmiennych
# w ciągu pozwalają na ich szybką identyfikację i wielokrotne wykorzystanie w
# dowolnej kolejności

# poniżej kolejny sposób
print("Lubię język {1} oraz {0}".format("Java", "Python"))

# w nowej wersji języka Python możliwe jest również odwoływanie się do elementów
kolekcji
# lub pól klasy


class Osoba:

    def __init__(self, imie, nazwisko):
        self.imie = imie
        self.nazwisko = nazwisko


kr = Osoba("Krzysztof", "Ropiak")

print("Tą osobą jest {0.imie} {0.nazwisko}".format(kr))
```

Mimo, iż ilość przykładów i sposobów tutaj przedstawionych jest dość duża i wyczerpuje większość najczęstszych przypadków to w dokumentacji można znaleźć jeszcze wiele dodatkowych możliwości. Po więcej przykładów można udać się pod poniższe adresy:

1. https://docs.python.org/3/library/string.html#format-string-syntax
2. https://pyformat.info/

Ćwiczenia

1. Pobierz ze strony https://pl.lipsum.com/ tekst akapitu o tytule „Czym jest Lorem Ipsum” i przypisz go do zmiennej.
2. Wyświetl na konsoli tekst postaci „W tekście jest {liczba_liter1} liter … oraz {liczba_liter2} liter …” . W miejsca { } podstaw zmienne, które będą przechowywały liczbę wystąpień danych liter. Litery, które mają być wyszukane powinny zostać przekazane jako indeks do 3 znaku nazwiska oraz 2 znaku imienia osoby wykonującej ćwiczenie, np. imie = „Krzysztof”, nazwisko = „Ropiak”, litera_1 = imie[2], litera_2 = nazwisko[3].
3. Przejdź na stronę https://pyformat.info/ a następnie zapisz w oddzielnym pliku .py i wykonaj 5 wybranych przykładów formatowania ciągów oznaczonego jako „New”, których nie było w przykładach z tego podrozdziału (np. z wyrównaniem, ilością pozycji liczby, znakiem itp.).
4. Mając dany poniższy fragment kodu (kod umieść w pliku):

```
print(dir(zmienna typu string))
help(zmienna_typu_string.wybrana_funkcja)
```

   Pod `zmienna_typu_string` podstaw dowolny ciąg tekstowy – może być jako odwołanie do zmiennej lub bezpośrednio ciąg tekstowy. Następnie z listy metod, które wyświetli pierwsze polecenie wybierz jedną z funkcji, która nie była omawiana w podrozdziale (wybierz z tych, które w nazwie nie posiadają __ na początku i końcu) i podstaw pod tekst `wybrana_funkcja` w drugim wyrażeniu. Uruchom ten fragment kodu. Pojawi się opis działania wybranej funkcji. Teraz przytrzymaj klawisz CTRL i najedź kursorem myszy na nazwę wybranej funkcji w kodzie i wciśnij lewy klawisz myszy. W ten sposób można szybko przenieść się do pliku źródłowego z definicją funkcji i jej krótkim opisem.
5. Wyszukaj w Internecie pojęcie „extended slice” w kontekście Pythona i wyświetl swoje imię i nazwisko z odwróconą kolejnością znaków z kapitalikami. Np. Fotzsyzrk Kaipor

6.4. LISTY

Lista w języku Python to kolekcja, którą można porównać do tablic w innych językach programowania. Ważną cechą list jest to, że mogą przechowywać różne typy danych. Rozmiar tablicy ograniczony jest możliwościami sprzętu.

Listę możemy zainicjalizować w poniższy sposób:

```
lista = []
lista2 = list()
lista3 = [1, 2, 3]
lista4 = ["a", 5, "Python", 7.8]
```

Do elementów listy odwołujemy się tak samo jak do elementów ciągu tekstowego gdyż tam również mamy do czynienia z listą (chociaż są to obiekty typu string).

Możemy również umieszczać listy w liście, co daje nam listy wielopoziomowe.

```
lista5 = [lista3, lista4]
```

Po wypisaniu takiej listy otrzymamy:

```
[[1, 2, 3], ['a', 5, 'Python', 7.8]]
```

W Pythonie można też w łatwy sposób łączyć ze sobą listy:

```
lista3.extend(lista4)
print(lista3)

wyjście -> [1, 2, 3, 'a', 5, 'Python', 7.8]

# lub w prostszy sposób
lista6 = lista3 + lista4
print(lista6)


Wyjście -> [1, 2, 3, 'a', 5, 'Python', 7.8]
```

Obie metody różnią się od siebie tym, że pierwsza modyfikuje już istniejącą listę, a druga wymaga podstawienia połączonej listy pod zmienną gdyż sama arytmetyczna operacja „+” nie spowoduje zmiany pierwotnej listy.

Niektóre metody, które można wykonać na obiekcie listy wykonują się jako operacje in-place co oznacza, że operacja wykonywana jest bez zwracania nowej wartości przez co nie można zmienionej tablicy przypisać do innej zmiennej. Poniżej przykład z sortowaniem:

```
lista7 = [7, 9, 3, 1]

posortowana = lista7.sort()
print(lista7)
print(posortowana)

# wyjście

[1, 3, 7, 9]

None
```

Wartość **None** w Pythonie odpowiada wartości **Null** w innych językach programowania. Nie możemy też posortować tablicy, w której znajdują się liczby oraz ciągi tekstowe.

Listy mogą być „cięte” (ang. sliced) tak jak ciągi tekstowe przedstawione w poprzednim rozdziale.

Dodawanie, usuwanie i zmiana wartości elementów listy może być wykonywana na wiele sposobów. Poniżej listing z niektórymi z nich.

```
# wstawianie i usuwanie elementów listy

skala = [1, 2, 3, 4, 5]
# dodajemy element na końcu listy
skala.append(6)
print(skala)

# znamy już sposób odwoływania się do elementu listy poprzez indeks więc można
# wartości listy ustawiać w ten sposób, ale nie da się wstawić wartośći na
# indeksie, który nie istnieje
skala[6] = 7
# zostanie zwrócony błąd IndexError: list assignment index out of range
# ale można zrobić to np. tak
skala[6:] = [7]
# lub tak
skala[len(skala):] = [7]

# alternatywnym sposobem jest wywołanie metody insert
skala.insert(6, 7)
print(skala)

# usuwamy element z końca listy co powoduje, że z wykorzystaniem
# tych metod osiągamy funkcję stosu
skala.pop()
print(skala)

# pop może również przyjmować indeks elementu do usunięcia.
# Metoda pop zwraca również wartość elementu usuwanego
skala.pop(2)
print(skala)
```

Do usuwania elementów listy można wykorzystać również wbudowaną funkcję del(), która nie zwraca żadnej wartości. Za jej pomocą można również usuwać zmienne.

W tym rozdziale zostały zaprezentowane podstawy jeżeli chodzi o operacje na listach. Python oferuje wiele bardziej rozbudowanych możliwości generowania, wybierania, sortowania list jednak najpierw muszą zostać wprowadzone takie pojęcia jak pętla czy instrukcja warunkowa. W dokumentacji można również znaleźć przykłady korzystające z instrukcji lambda, ale jej użycie będzie omawiane szerzej podczas zajęć z zaawansowanego Pythona.

Ćwiczenia

1. Stwórz listę z wartościami od 1 do 10. Następnie podziel listę tak, aby pierwsze 5 liczb zostało w oryginalnej liście a pozostałe 5 znalazło się w nowej liście.
2. Połącz te listy ponownie. Dodaj do listy wartość „0” na początku. Utwórz kopię połączonej listy i wyświetl listę posortowaną malejąco.
3. Za pomocą rzutowania stringa na listę wykonaj ćwiczenie 5 z podrozdziału 6.2.

6.5. SŁOWNIKI

Słowniki to tablica mieszająca lub inaczej tablica z haszowaniem, którą można porównać do tablic asocjacyjnych znanych z innych języków programowania. Słowniki przechowują pary klucz:wartość i właśnie po kluczu odbywa się wyszukiwanie. Kluczem w słowniku może być każdy niezmienny typ danych np., string lub liczba. Kluczem może być również krotka, jeżeli zawiera typy niezmienne (string, liczba, krotka). Klucze w słowniku są unikalne a pary elementów nie są uporządkowane w kolejności, w której zostały dodane. Słownik został już wykorzystany we wcześniejszych przykładach w podrozdziale z formatowaniem ciągów tekstowych.

Poniżej fragmenty kodu tworzące słownik oraz pokazujące jak uzyskać dostęp do jego danych.

```
# tworzenie słownika
slownik = {}
slownik = dict([("jeden", 1), ("dwa", 2), ("trzy", 3)])
slownik = dict(jeden=1, dwa=2, trzy=3)
slownik = dict({"jeden": 1, "dwa": 2, "trzy": 3})
slownik = {"jeden": 1, "dwa": 2, "trzy": 3}

print(slownik["jeden"])

# sprawdzenie czy klucz jest w słowniku czy nie
print("jeden" in slownik)
# wypisanie wszystkich kluczy
print(slownik.keys())
# wypisanie wszystkich wartości
print(slownik.values())
# można również sprawdzić czy klucz występuje w słowniku
# w przedstawiony poniżej sposób, ale jest on wolniejszy
print("jeden" in slownik.keys())
# dodanie elementu do słownika
slownik["cztery"] = 4
print(slownik.keys())
```

6.6. KROTKI

Krotki (ang. tuples) są bardzo podobne do list z tą różnicą, że są typem niezmiennym i deklaracja zmiennych zapisywana jest w nawiasach zwykłych a nie kwadratowych. Również mogą przechowywać wiele typów danych jednocześnie.

```
# tworzymy krotke
krotka = (1, 2, "Jacek", "ma")
krotka_liczb = krotka[:2]
print(krotka_liczb)
krotka_stringow = krotka[2:]
print(krotka_stringow)

nowa_krotka = tuple()
najnowsza_krotka = tuple([1, 2, 3])

# możemy również rzutować typy krotka - lista
lista = [1, 2, "Ala", "też", "ma"]
krotka_z_listy = tuple(lista)
nowa_lista = list(krotka_z_listy)

# krotki mogą być zagnieżdżane
duza_krotka = krotka_stringow, krotka_liczb, tuple(nowa_lista)
print(duza_krotka)

# a jeżeli zagnieździmy listę w krotce ?
listokrotka = krotka_z_listy, lista
# to nadal możemy modyfikować elementy listy
listokrotka[1][0] = 0
print(listokrotka)

# pakowanie krotki (tuple packing)
t = 5, 6, 7
print(t)
x, y, z = t
# i rozpakowywanie krotki (tuple unpacking)
# inny sposób łączenia zmiennych różnego typu w string
print("x = " + str(x))
print("y = " + repr(y))
print("z = " + str(z))



# na wyjściu
(1, 2)
('Jacek', 'ma')
(('Jacek', 'ma'), (1, 2), (1, 2, 'Ala', 'też', 'ma'))
((1, 2, 'Ala', 'też', 'ma'), [0, 2, 'Ala', 'też', 'ma'])
(5, 6, 7)
x = 5
y = 6
z = 7
```

6.7. ZBIORY

Zbiór (ang. set) to nieuporządkowana kolekcja, której ważną cechą jest to, że znajdują się w niej unikalne elementy (czyli bez powtórzeń). Zbiory obsługują również matematyczne operacje, które z teorii zbiorów są znane: suma, przecięcie, różnica oraz różnica symetryczna.

```
# inicjalizacja zbiorów
klasa = {"Marek", "Janek", "Ania", "Ewa", "Marek", "Ania"}
print(klasa) # duplikatów już nie ma

# a teraz zbiór znaków ze stringa
czar = set("czabunagunga")
print(czar)
inny_czar = set("abrakadabra")
print(inny_czar)
print(czar - inny_czar) # są w czar, ale nie ma w inny_czar
print(czar.difference(inny_czar)) # to samo, ale inaczej
print(inny_czar - czar) # to nie to samo, jak wiadomo z teorii

print(czar | inny_czar) # znaki w czar lub inny_czar lub obu
print(czar & inny_czar) # przecięcie czyli część wspólna
print(czar.intersection(inny_czar)) # można tak
print(czar ^ inny_czar) # różnica symetryczna
```

Bardzo przydatna staje się własność unikalnych elementów zbioru jeżeli chcemy wyeliminować duplikaty z listy, gdyż wystarczy rzutować listę na zbiór i sprawa załatwiona a następnie, jeżeli na wyjściu potrzebujemy znowu listy to rzutujemy w odwrotną stronę.

6.8. FUNKCJA RANGE

Funkcja (chociaż nazywając precyzyjniej to niezmienna sekwencja) range służy to generowania ciągu liczb według zadanych parametrów. Często przydaje się w pętlach lub podczas tworzenia list lub zbiorów liczb. Funkcja range() i sposób jej użycia zmieniał się w trakcie rozwoju języka i jej zastosowanie w wersji Python 2.x różni się od tego co będzie zaprezentowane tutaj. Zapoznanie się ze szczegółami pozostawiam czytelnikowi.

Poniższe fragmenty kodu zaprezentują sposób działania funkcji range.

```
from decimal import *

# jakim typem jest range ?
liczby = range(5)
print(type(liczby)) # range jest obiektem typu range (w Python 2.x była to lista)

# range może przyjmować 1 parametr, wtedy jest to parametr stop
for i in range(10):
    print(i)

# range może też przyjmować 2 parametry (start, stop)
for i in range(4, 10):
    print(i)

# lub 3 parametry (start, stop, step)
for i in range(4, 10, 2):
    print(i)

# możemy również generować wartości ujemne
for i in range(-5, -1):
    print(i)

for i in range(-5, -10, -2):
    print(i)

# lista z elementów funkcji range
lista = list(range(10))
print(lista)

# funkcja range nie generuje wartości zmiennoprzecinkowych, ale można
# dość łatwo taką funkcję stworzyć
def frange(start, stop, step):
    i = start
    while i < stop:
        yield i
        i += step


for i in frange(0.1, 0.5, 0.1):
    print(i)

# po wyświetleniu wyniku można się zdziwić bo 0.3 to właściwie nie 0.3...
# to może jeszcze raz, ale tak
for i in frange(0.1, 0.5, 0.1):
    print(Decimal(i))

print((0.1 + 0.2) == 0.3)  # kto by się spodziewał ?
print(round((0.1 + 0.2), 2) == round(0.3, 2))  # teraz lepiej
print((Decimal(0.1) + Decimal(0.2)) == Decimal(0.3))
print((Decimal('0.1') + Decimal('0.2')) == Decimal('0.3'))
```

Dla zainteresowanych problemem reprezentacji liczb zmiennoprzecinkowych w systemie dwójkowym odsyłam do dość zrozumiale napisanego artykułu http://www.samouczekprogramisty.pl/liczby-zmiennoprzecinkowe/

7. STEROWANIE WYKONANIEM PROGRAMU

7.1. INSTRUKCJE WARUNKOWE

Język Python posiada tylko jedną wbudowaną instrukcję warunkową i jest nią instrukcja if/elif/else. Nie znajdziemy tutaj konstrukcji case/switch.

Oto najprostsza postać tej instrukcji:

```
liczba1 = 1
liczba2 = 2

if liczba1 > liczba2:
    print("Pierwsza liczba jest większa")
```

Zobaczmy jak wygląda bardziej rozbudowana jej wersja wraz z obsługą danych wprowadzanych z klawiatury.

```
liczba = input("Podaj liczbę całkowitą ")
liczba = int(liczba)

if liczba < 10:
    print("To dość mała liczba")
elif 9 < liczba < 100:  # to jest wersja skrócona warunku
    print("To już całkiem duża liczba")
else:
    print("To musi być wielka liczba")
```

Aby budować bardziej złożone warunki używamy operatorów boolowskich, które zostały przedstawione w podrozdziale Kilka słów o operatorach.

```
if liczba < 10 or liczba > 15:
    print("Liczba nie jest z odpowiedniego przedziału")

# możemy również sprawdzić warunek zawierania się elementu w kolekcji
zbior_dopuszczalny = [1, 3, 5, 7, 9]
if liczba not in zbior_dopuszczalny:
    print("Podana liczba nie znajduje się w zbiorze")
```

Jak zostało już wspomniane wcześniej Python posiada typ None, który odpowiada Null znanemu z innych języków oraz baz danych. Ponownie odsyłam do podrozdziału Kilka słów o operatorach gdzie znajduje się informacja co jest traktowane jako Null w Pythonie.

```
liczba1 = 1
liczba2 = 2

if liczba1 > liczba2:
    print("Pierwsza liczba jest większa")

liczba = input("Podaj liczbę całkowitą ")
liczba = int(liczba)

if liczba < 10:
    print("To dość mała liczba")
elif 9 < liczba < 100:  # to jest wersja skrócona warunku
    print("To już całkiem duża liczba")
else:
    print("To musi być wielka liczba")

if liczba < 10 or liczba > 15:
    print("Liczba nie jest z odpowiedniego przedziału")

# możemy również sprawdzić warunek zawierania sie elementu w kolekcji
zbior_dopuszczalny = [1, 3, 5, 7, 9]
if liczba not in zbior_dopuszczalny:
    print("Podana liczba nie znajduje się w zbiorze")

nic = None
pusty_ciag = ""

if not nic:
    print("None to False")
if not pusty_ciag:
    print("Pusty ciąg to False")
if nic == pusty ciag:
    print("None i pusty ciąg to boolowskie False, ale nie są sobie
równe")
# jeżeli chcemy sprawdzić czy ciąg jest pusty
if pusty_ciag == "":
    print("To jest pusty ciąg")
```

Instrukcję if można również znaleźć w wielu skryptach Pythona sprawdzającą dość tajemniczą własność `if __name__ == „__main__”.` Poniżej wyjaśnienie stosownym przykładem.

```
# Jest również specjalne zastosowanie instrukcji if
# poniższy zapis powoduje, że kod umieszczony wewnątrz tego bloku
# zostanie uruchomiony tylko w przypadku, gdy plik zostanie
# uruchomiony bezpośrednio, tak jak w tym przypadku.
# Jeżeli plik zostanie zaimportowany, to kod nie zostanie uruchomiony

if __name__ == "__main__":
    pass

# możemy też sprawdzić jaką wartość ma zmienna specjalna __name__
print(__name__)
# instrukcja pass nie robi nic, ale jeżeli wymagany jest tutaj kod, żeby
# spełnić wymogi składniowe to możemy jej użyć
```

7.2. PĘTLE

W Pythonie mamy do dyspozycji dwie pętle: **for** oraz **while** przy czym ta pierwsza jest zdecydowanie bardziej „popularna" wśród programistów Pythona. Przykład zastosowania pętli for znalazł się już w podrozdziale 6.8 gdzie opisywana była funkcja range. Dla przypomnienia w poniższych przykładach również pojawi się jej zastosowanie.

```
# for z funkcją range
for i in range(3):
    print(i)

# for dla listy
lista = [4, 5, 6]
for i in lista:
    print(i)

# a gdybyśmy chcieli zwracać również index elementów listy ?
for index, wartosc in enumerate(lista):
    print(str(index) + " -> " + str(wartosc))

# a mozna jeszcze tak, gdyż funkcja enumerate wypakowuje każdy element
listy
# w postaci krotki (index, wartość_z_tablicy)
for krotka in enumerate(lista):
    print(str(krotka[0]) + " -> " + str(krotka[1]))

print(type(krotka[0]))
print(type(krotka[1]))
print(type(krotka))

# pętla for i słowniki
# jeżeli nie wskażemy pętli for czy chcemy iterować po kluczach czy
wartościach
# to domyślnie zostaną wybrane klucze
slownik = {"imie": "Marek", "nazwisko": "Kowalski", "plec": "mezczyzna"}
for key in slownik:
    print(key)

for val in slownik.values():
    print(val)

for key, value in slownik.items():
    print(key + " -> " + value)

for key in slownik:
    print(key + " -> " + slownik[key])
```

Postać Pythonowej pętli while nie różni się od jej sposobu działania w innych językach.

```
# pętla while
counter = 0
while True:
    counter += 1
    if counter > 10:
        break

counter = 0
while counter < 5:
    print(str(counter) + " mniejsze od 5")
    counter += 1

# pętla while nadaje się dobrze w sytuacji kiedy nie wiemy kiedy (nie
# znamy liczby iteracji) się ona zakończy, np. przy pobieraniu danych
# wejściowych w oczekiwaniu na podanie komendy równej warunkowi stopu
pętli

lista = []
print("Podaj liczby całkowite, które chcesz umieścić w pętli.")
print("Wpisz 'stop' aby zakończyć")
while True:
    wejscie = input()
    if wejscie == 'stop':
        break
    lista.append(int(wejscie))

print("Twoja lista -> " + repr(lista))
```

W tym miejscu należy wspomnieć o instrukcji **break** oraz **continue**, które możemy umieszczać wewnątrz pętli. Break powoduje zakończenie pętli (tylko tej, w bloku której znalazła się instrukcja) natomiast continue kończy przebieg aktualnej iteracji pętli (czyli to co jest za continue się nie wykona) i rozpoczyna kolejną iterację.

7.3. PYTHON COMPREHENSIONS

Tytuł podrozdziału postanowiłem pozostawić w języku angielskim, gdyż nie ma chyba dobrego tłumaczenia tego terminu na język polski w odniesieniu do programowania. Mechanizm ten polega na generowaniu kolekcji (lista, słownik, zbiór) na podstawie jednowierszowego zapisu określającego warunki dla zmiennych, które zostaną w danej kolekcji umieszczone. Najlepiej można to oddać za pomocą odpowiednich przykładów.

```
# comprehensions dla list
x = [i for i in range(5)]
print(x)

y = [i for i in range(10) if i % 2 == 0]
print(y)

# możemy wykonać funkcję dla każdej wartości
literoliczby = ["1", "2", "3", "4", "5"]
liczby = [int(i) for i in literoliczby]
print(liczby)

# te instrukcje można również zagnieżdżać, np. dla listy list
# przykład z dokumentacji Pythona
vec_of_vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
single = [num for elem in vec_of_vec for num in elem]
print(single)

# comprehensions i słowniki
# odwrócenie kolejności par klucz:wartość w słowniku
slownik = {1: "Burek", 2: "Azor", 3: "Fafik"}
print({value: key for key, value in slownik.items()})

# comprehensions i zbiory
# zwróć uwagę na nawiast klamrowy jak dla słowników
lista = [1, 2, 2, 3, 4, 4, 4, 5]
zbior = {i for i in lista}
print(zbior)
```

Mechanizm przedstawiony w tym podrozdziale jest bardzo przydatny, chociaż skomplikowane polecenia mogą być wyzwaniem do zinterpretowania względem „tradycyjnego” podejścia. Więcej przykładów można znaleźć w dokumentacji Pythona pod adresem https://docs.python.org/3/tutorial/datastructures.html.

8. DEFINIOWANIE FUNKCJI

Ogólna definicja funkcji mówi, że jest to wydzielony blok kodu, który ma robić możliwie jak najmniej rzeczy na raz, ale ma to robić dobrze. Jest to też niezbędny element metodologii DRY (Don't Repeat Yourself), czyli tam gdzie jakaś funkcjonalność będzie wykorzystywana wielokrotnie możemy zastosować funkcję. Przykładowa deklaracja funkcji w języku Python poniżej.

```
def zainicjalizuj():
    print("inicjalizacja...")
```

Słowo kluczowe def informuje interpreter o tym, że jest to deklaracja funkcji. Funkcja ma swoją nazwę oraz zero lub więcej argumentów. Funkcja może wykonywać jakieś operacje i zmieniać stan obiektów nie zwracając nic na wyjściu (w Javie używamy jako typu zwracanego void) lub może zwracać jakieś wartości. W odróżnieniu od niektórych języków programowania sygnatura funkcji nie mówi nam jakiej wartości możemy się spodziewać.

```
def powiel(tekst, ile_razy):
    return (tekst + " ") * ile_razy


print(powiel("jesień", 5))
```

Należy również pamiętać o tym, że w przypadku gdy deklaracja i wywołanie funkcji odbywają się w tym samym pliku to wywołanie funkcji musi znaleźć się po jej deklaracji.

Podobnie jak np. w języku PHP argumenty funkcji mogą mieć swoje wartości domyślne i zostaną użyte w przypadku gdy programista nie przekaże ich wartości.

```
def powiel(tekst="Hello ", ile_razy=3):
    return (tekst + " ") * ile_razy


print(powiel())
```

W przypadku, gdy funkcja zawiera argumenty opcjonalne w celu uniknięcia konieczności pilnowania kolejności przekazywanych argumentów możemy przekazać wartości argumentów wraz z ich nazwami.

```
print(powiel(ile_razy=5, tekst="Jingle bells"))
```

Funkcje w Pythonie mogą przyjmować właściwie nieograniczoną liczbę argumentów lub argumentów z kluczem (te, których nazwę określamy). Zgodnie z przyjętą konwencją zmienna przechowująca te pierwsze to *args a zmienna dla argumentów z kluczem to **kwargs, ale ich nazwy w naszych funkcjach mogą być dowolne.

```
def robie_duzo_rzeczy(*args, **kwargs):
    print(args)
    print(kwargs)


robie_duzo_rzeczy(3, 4, 5.6, imie="Krzysztof", hobby=["sport",
"fantasy"])
```

Zasięg zmiennych oraz zmienne globalne. Zmienne zadeklarowane wewnątrz funkcji mają zasięg tylko wewnątrz tej funkcji, co oznacza, że próba odwołania się do nich poza funkcją nie powiedzie się. Istnieje jednak możliwość zdefiniowania takiej zmiennej jako zmienna globalna. Mimo, że jest to częścią języka to wykorzystywanie tego mechanizmu nie jest zalecane a często wręcz piętnowane przez innych programistów.

```
def mam_globala():
    global a
    a = 1
    b = 2
    return a + b


def nie_mam_globala():
    c = 3
    return a + c


print(mam_globala())
print(nie_mam_globala())
```

9. PAKIETY I MODUŁY ORAZ ICH IMPORTOWANIE

W środowisku Python moduł to po prostu pojedynczy plik z kodem Pythona, który możemy stworzyć i zapisać. Pakiet to zbiór takich modułów a najczęściej oznacza folder, w którym znajdują się pliki (moduły). Oba te elementy można importować na różne sposoby, kilka przykładów znalazło się w rozdziale poświęconym formatowaniu kodu wg. PEP8.

Zawartość modułu może być zbiorem funkcji lub klas i metod. Taki moduł możemy następnie po zaimportowaniu wykorzystywać w innych naszych programach, modułach lub pakietach. Poniżej prosty przykład modułu i jego wykorzystania w innym skrypcie.

```
# plik matma.py
"""deklaracja funkcji w prostym module"""


def dodaj(a, b):
    return a + b


def odejmij(a, b):
    return a - b


def podziel(a, b):
    return a / b


def pomnoz(a, b):
    return a * b
```

Teraz możemy z poziomu innego pliku w tym samym folderze zaimportować ten moduł.

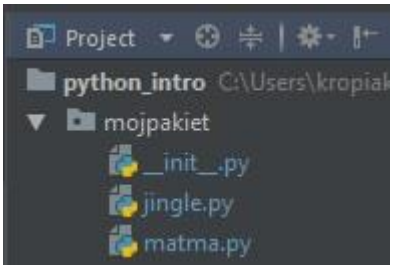
```
import matma

print(matma.dodaj(1, 2))
print(dir(mojpakiet))
print(dir(mojpakiet.matma))
```

Moduły najczęściej importują inne moduły a te importują kolejne. Jeżeli w trakcie tego procesu zostanie napotkany zduplikowany import to zostanie on zignorowany. Polecenie dir(modul|pakiet) wyświetla wszystkie zmienne oraz nazwy modułów, które się w nich znajdują.

Aby stworzyć pakiet trzeba jeszcze dodać specjalny plik do folderu (pakietu) o nazwie __init__.py . Jeżeli korzystamy z zalecanego sposobu importu modułów czyli **`import pakiet.moduł`** to możemy jego zawartość pozostawić pustą chociaż może się tam również znajdować kod inicjalizujący dla pakietu. Natomiast jeżeli chcemy umożliwić import z tego pakietu za pomocą **`from pakiet import *`** to plik __init__.py powinien zawierać zmienną __all__ zawierającą listę modułów, które będą zaimportowane:

```
__all__ = ["matma", "jingle"]
```

Aby sprawdzić sposób działania poniższych listingów przedstawię najpierw strukturę katalogów i plików, które zostały wykorzystane w poniższych przykładach.



Reasumując, został utworzony folder mojpakiet a w nim znajduje się plik matma.py przedstawiony na listingu w tym rozdziale. Zawartość pliku __init__.py przedstawiłem na poprzednim listingu. Została tylko zawartość pliku jingle.py:

```
__all__ = ['sing']


def sing():
    return "Jingle bells, jingle bells..."


def snow():
    return "It's snowing..."
```

Plik, który wykonuje same importy na tym pakiecie znajduje się w tym samym folderze co pakiet. A jego zawartość to:

```
from mojpakiet import *

print(matma.dodaj(1, 2))
print(jingle.sing())
```

Zmienna __all__ służy również do określania listy funkcji w module. Zmienna jest wtedy umieszczana w pliku z definicją klas/funkcji a jej postać jest taka sama jak w przypadku pakietu z taką różnicą, że po zaimportowaniu takiego modułu (pliku) wszystkie jego funkcje i tak będą widoczne bez względu na to czy znalazły się w zmiennej __all__ czy też nie. Określanie __all__ w module najczęściej wykorzystywane jest po to, aby mieć dostęp do listy klas, metod danego modułu np. po to, aby sprawdzić jakie funkcje możemy wywołać. Bardzo dobrze wytłumaczone jest to w artykule pod adresem http://xion.io/post/code/python-all-wild-imports.html .

Struktura pakietów i modułów może być bardziej rozbudowana i rozmieszczona na kolejnych poziomach zagnieżdżenia. Więcej informacji można znaleźć pod adresem https://docs.python.org/3/tutorial/modules.html

Podsumowując, zalecanym sposobem wykonywania importów jest **from pakiet import moduł** lub **import moduł**.

Jako ciekawostkę polecam umieszczenie polecenia **import this** w swoim pliku lub wykonanie w konsoli.

10. ZARZĄDZANIE PAKIETAMI ORAZ VIRTUALENV

Właściwie każdy popularny język programowania posiada mechanizm pozwalający na zarządzanie dodatkowymi bibliotekami czy pakietami. Python również posiada swojego menadżera pakietów o nazwie **pip**. W wersji 3.x nie jest konieczne jego ręczne instalowanie gdyż jest już domyślnie dołączany do tych dystrybucji.

PIP w Windowsowym wierszu poleceń

Aby korzystać z narzędzia PIP w wierszu poleceń systemu Windows musimy się najpierw upewnić, że interpreter Pythona (oraz folder Scripts) znajduje się w zmiennej środowiskowej PATH. Możemy albo wyświetlić zmienną path, albo po prostu w terminalu wykonać polecenie **python –version,** które zwróci wersję Pythona, z której aktualnie korzystamy lub polecenie nie zostanie rozpoznane.

Jeżeli powyższy warunek jest spełniony możemy uruchomić narzędzie PIP i np. wyświetlić listę pakietów z aktualnego środowiska Pythona:

```
python –m pip list
```

Listę poleceń i skromny help uzyskamy po komendzie:

```
python –m pip help
```

Index pakietów, które można zainstalować z indexy PyPi znajdziemy pod adresem https://pypi.python.org/pypi, gdzie aktualnie znajduje się ponad 122 000 pakietów…

Jak wynika z helpa komenda służąca do instalacji pakietu to install nazwa_pakietu. Zainstalujmy pakiet requests.

```
python –m pip install requests
```

Możliwe jest również wybranie konkretnej wersji pakietu, którą chcemy zainstalować.

```
python –m pip install requests==2.18.0
```

Jak nie trudno się domyślić opcja uninstall służy do odinstalowywania pakietów.

Narzędzie PIP umożliwia również instalowanie pakietów na podstawie pliku wymagań, którego przykładowa postać może wyglądać tak:

```
#
####### example-requirements.txt #######
#
###### wymagania bez określonej wersji ######
nose
nose-cov
beautifulsoup4
#
```

```
###### wymagania z okreslonq wersją ######
#   zobacz https://www.python.org/dev/peps/pep-0440/#version-specifiers
docopt == 0.6.1             # dopasowanie wersji. Musi być 0.6.1
keyring >= 4.1.1            # minimalna wersja  4.1.1
coverage != 3.5             # wykluczenie wersji. Wszystko poza wersją 3.5
Mopidy-Dirble ~= 1.1        # Kompatybilna wersja. Rozumiane jako >= 1.1, == 1.*
#
###### odwołuje się do innego pliku z wymaganiami ######
-r other-requirements.txt
#
#
###### konkretny plik z pakietem np. wcześniej pobrany ######
./downloads/numpy-1.9.2-cp34-none-win32.whl
http://wxpython.org/Phoenix/snapshot-builds/wxPython_Phoenix-3.0.3.dev1820+49a8884-cp34-
none-win_amd64.whl
#
###### dodatkowe pakiety bez określania wersji ######
#   Umieszczone tutaj tylko po to, aby pokazać, że kolejność nie ma znaczenia.
rejected
green
#
```

Komedna uruchamiająca instalację pakietów z pliku requirements.txt:

```
python -m pip install -r requirements.txt
```

Są to podstawowe i najczęściej wykorzystywane komendy narzędzia PIP. Po bardziej szczegółową dokumentację wraz z przykładami odsyłam pod adres: https://pip.pypa.io/en/stable/.

Virtualenv

Virtualenv jest skrótem od **virtual environment** co oznacza wirtualne środowisko. Narzędzie to pozwala na tworzenie odrębnych środowisk zawierających interpreter Pythona oraz zestaw pakietów, które chcemy wykorzystać w konkretnym projekcie lub przed aktualizacją pakietów w projekcie produkcyjnym chcemy sprawdzić jak aplikacja będzie się zachowywała w nowym środowisku.

Virtualenv jest pakietem Pythona więc aby z niego skorzystać należy upewnić się, że stosowny pakiet jest zainstalowany i ewentualnie go zainstalować.

Aby stworzyć nowe środowisko wirtualne należy wskazać folder, w którym takie środowisko chcemy stworzyć. Następnie wykonanie komendy:

```
virtualenv nazwa_srodowiska
```

stworzy nowy folder w tym miejscu i skopiuje interpreter Pythona, który był aktualnie ustawiony w zmiennej środowiskowej PATH oraz dołączy kilka skryptów pozwalających m.in. na aktywację i deaktywację środowiska wirtualnego.

Kolejną czynnością, którą trzeba wykonać aby rozpocząć pracę w tym środowisku jest jego aktywacja, która polega na uruchomieniu skryptu z pliku **Scripts\activate** znajdującego się w folderze nowego środowiska. Od teraz aktywnym interpreterem jest ten zawarty w nowym środowisku. Możemy teraz uruchamiać skrypty Pythona, instalować pakiety oraz korzystać z konsoli Python w odniesieniu do tego środowiska.

Na zajęciach na podstawie dokumentacji ze strony https://virtualenv.pypa.io/en/stable/userguide/ zostanie zaprezentowany sposób konfiguracji i korzystania ze środowiska wirtualnego.

11. OBIEKTOWY PYTHON

Programowanie obiektowe (ang. object-oriented programming) – paradygmat programowania, w którym programy definiuje się za pomocą obiektów – elementów łączących stan (czyli dane, nazywane najczęściej polami) i zachowanie (czyli procedury, tu: metody). Obiektowy program komputerowy wyrażony jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.

Podejście to różni się od tradycyjnego programowania proceduralnego, gdzie dane i procedury nie są ze sobą bezpośrednio związane. Programowanie obiektowe ma ułatwić pisanie, konserwację i wielokrotne użycie programów lub ich fragmentów.

Największym atutem programowania, projektowania oraz analizy obiektowej jest **zgodność takiego podejścia z rzeczywistością – mózg ludzki jest w naturalny sposób najlepiej przystosowany do takiego podejścia przy przetwarzaniu informacji.**

Źródło: Wikipedia

Tytułem wstępu przytoczyłem dość zwięźle i konkretnie określoną definicję programowania obiektowego. Projektując obiekty, ich metody i powiązania próbujemy modelować otaczający nas świat i przenosić do rzeczywistości wirtualnej znane nam z życia elementy po to aby rozwiązać jakiś problem lub stworzyć narzędzie do pracy dla nas lub innych ludzi. Zasada projektowania mówi też, że obiekty powinny być możliwie ograniczone w zakresie operacji jakie możemy dzięki nim wykonać. Niech robią stosunkowo niewiele ale za to dobrze. Pomaga to również lepiej rozumieć program i powiązania między obiektami, ponownie je wykorzystywać jak również rozszerzać jego możliwości i łatwiej konserwować kod.

Od ogólnych założeń przechodzimy do deklaracji przykładowej klasy w Pythonie.

```
"""docstring dla modułu"""


class Pojazd:
    """ docstring dla klasy """
    def __init__(self):
        """ konstruktor """
        pass
```

W wersji Python 2.x ta deklaracja wyglądałaby tak:

```
"""docstring dla modułu"""


class Pojazd(object):
    """ docstring dla klasy """
    def __init__(self):
        """ konstruktor """
        pass
```

W wersji 3.x zrezygnowano z jawnego każdorazowego rozszerzania obiektu **object** co jest dość oczywiste i od dawna funkcjonuje np. w Javie.

W przykładzie powyżej została zadeklarowana tylko jedna metoda o nazwie __init__ która jest konstruktorem. Definicja metody (bo tak nazywają się funkcje klasy) nie różni się mocno od definicji funkcji, które już poznaliśmy. Każda metoda przyjmuje specjalny argument o nazwie self, który oznacza odwołanie do obiektu, w którym została zdefiniowana. To odpowiednik **this** znanego z innych języków programowania.

Rozbudujmy nieco naszą klasę.

```
"""docstring dla modułu"""


class Pojazd:
    """ docstring dla klasy """
    def __init__(self, kolor, marka):
        """ konstruktor """
        self.kolor = kolor
        self.marka = marka

    def hamuj(self):
        """
        zatrzymaj samochód
        """
        return "hamuję..."

    def jedz(self):
        """
        jedziemy dalej
        """
        return "%s jedzie dalej" % self.marka


pojazd = Pojazd("niebieski", "Ford")
print(pojazd.jedz())
print(pojazd.hamuj())
```

Rzeczą, którą da się tutaj zauważyć jest na pewno brak modyfikatorów dostępu zarówno dla pól jak i metod klasy gdyż w Pythonie tak naprawdę prywatne metody, ani zmienne nie występują. Natomiast istnieje konwencja, która mówi, że poprzedzenie zmiennej lub metody prefiksem _ lub __ oznacza, że zmienna/metoda powinna być traktowana jako prywatna i inni programiści nie powinni z niej korzystać, bo jest przyzwolenie, aby je zmieniać bez ostrzeżenia. Tak więc nie uznawane są jako część API. Przy nazwach z __ działa też mechanizm, który zamazuje nieco widoczność takiej zmiennej lub metody powodując, że odwołanie do niej jest postaci _nazwaklasy__zmienna. Więcej można doczytać tutaj: https://docs.python.org/3/tutorial/classes.html

Standardowe zmienne klasowe są przechowywane dla każdej instancji klasy, ale Python umożliwia również zdefiniowanie zmiennych, które mogą być współdzielone przez wszystkie instancje danej klasy. Przykład na listingu na kolejnej stronie.

```
"""docstring dla modułu"""


class Pojazd:
    """ docstring dla klasy """

    sygnal = "Piiib piiib"

    def __init__(self, kolor, marka):
        """ konstruktor """
        self.kolor = kolor
        self.marka = marka

    def hamuj(self):
        """
        zatrzymaj samochód
        """
        return "hamuję..."

    def jedz(self):
        """
        jedziemy dalej
        """
        return "%s jedzie dalej" % self.marka


class OpelOmega(Pojazd):

    def hamuj(self):
        return "hamuje dość szybko..."


pojazd = Pojazd("niebieski", "Ford")
print(pojazd.jedz())
print(pojazd.hamuj())

opel = OpelOmega("zielony", "Opel")
print(opel.hamuj())
print(opel.sygnal)
```

Zagadnienia programowania obiektowego to również bardziej złożone mechanizmy takie jak klasy abstrakcyjne czy polimorfizm. Warto tutaj powiedzieć, że Python pozwala na wielokrotne dziedziczenie i dlatego nie posiada możliwości definiowania interfejsów. Te tematy wykraczają jednak poza zakres tego przedmiotu i zostaną omówione na zajęciach z zaawansowanego Pythona.

12. OBSŁUGA PLIKÓW

Przejdźmy od razu do omówienia kilku przykładów.

```
uchwyt = open("plik.txt")
uchwyt = open(r"C:\Users\kropiak\PycharmProjects\python_intro\plik.txt", "r")
```

Pierwsze polecenie otwiera plik, który znajduje się w folderze, w którym jest uruchamiany plik. Domyślnie plik otwierany jest tylko do odczytu. Drugie polecenie przyjmuje ścieżkę bezwzględną i dodatkowo kolejny parametr przekazuje tryb odczytu pliku, który tutaj również jest tylko do odczytu. Litera ‘r’ poprzedzająca ścieżkę informuje Pythona, że ma potraktować ten ciąg tekstowy „dosłownie” czyli nie będą brane pod uwagę ewentualne wystąpienia znaków specjalnych, które trzeba by poprzedzać znakiem „\”.

Podstawowy odczyt danych z pliku można wykonać tak:

```
uchwyt = open("plik.txt")
uchwyt = open(r"C:\Users\kropiak\PycharmProjects\python_intro\plik.txt", "r")
dane = uchwyt.read()
print(dane)
uchwyt.close()
```

I tutaj możemy zauważyć pierwszy problem, jeżeli w pliku tekstowym znajdowały się polskie ogonki. Możemy temu zaradzić dodając dodatkowy parametr określający jak powinny być kodowane odczytywane znaki. Pamiętajmy również o zamykaniu uchwytu do pliku po odczytaniu danych.

```
uchwyt = open(r"C:\Users\kropiak\PycharmProjects\python_intro\plik.txt",
"r", encoding="utf-8")
```

Możemy również odczytywać plik linia po linii z pomocą pętli:

```
uchwyt = open("plik.txt", "r", encoding="utf-8")

for linia in uchwyt:
    print(linia)

uchwyt.close()
```

W tym przypadku może pojawić się sytuacja, gdzie po każdej wyświetlonej linii na wyjściu będzie wypisywana nowa linia. To dlatego, że funkcja print dodaje na końcu znak ‘\n’, który oznacza nową linię, a jeżeli taki znak został również odczytana z pliku to mamy odpowiedź dlaczego tak się dzieje. Aby to zmienić można ustalić wartość parametru ‘end’ funkcji print na inną niż domyślna wartość.

Możemy również określić jakiej wielkości fragmenty pliku wyrażone w bajtach. Tym razem z pomocą pętli while:

```
uchwyt = open("plik.txt", "r", encoding="utf-8")
while True:
    dane = uchwyt.read(1024)
    print(dane, end="")
    if not dane:
        uchwyt.close()
        break
```

Teraz kolej na zapisywanie do pliku.

```
uchwyt = open("plik2.txt", "w", encoding="utf-8")
uchwyt.write("Zapisuję do pliku.")
uchwyt.close()
```

Istnieje bardziej nowoczesna metoda dostępu do plików, której wykorzystanie zwalnia nas z obowiązku pamiętania o zamknięciu uchwytu do pliku.

```
with open("plik.txt", "r", encoding="utf-8") as file_reader:
    for linia in file_reader:
        print(linia, end="")
```

Na koniec jeszcze przykład z obsługą wyjątków, o której więcej również w zaawansowanej części Pythona.

```
try:
    with open("plik.txt", "r", encoding="utf-8") as file_reader:
        for linia in file_reader:
            print(linia, end="")
except IOError:
    print("Wystapił wyjątek IOError")
```

13. JUPYTER (IPYTHON)

13.1. CZYM JEST JUPYTER NOTEBOOK

Jupyter Notebook to interaktywne środowisko obliczeniowe pozwalające użytkownikom na tworzenie dokumentów zawierających kod, sformatowany tekst, wykresy, równania, obrazy i filmy video. Te dokumenty, nazywane notatnikami (ang. notebook) zawierają zapis kodu i wyniku jego działania, pozwalając na udostępnianie notatnika innym użytkownikom np. poprzez e-mail, Dropbox, systemy kontroli wersji lub serwis nbviewer.jupyter.org.

Komponenty środowiska Jupyter Notebook

- **Aplikacja WWW** pozwalająca na interaktywne uruchamianie kodu wybranego języka oraz zarządzanie notatnikami.
- **Kernels (jądra)** – są to procesy uruchamiane do wykonywania kodu notatnika a następnie zwracania wyniku do aplikacji.
- **Notatniki** – samodzielne dokumenty, które zawierają sekwencję wprowadzonego kodu i zwracanych wyników, notatki itp. Każdy dokument posiada swoje jądro (kernel).

Aplikacja webowa Jupytera pozwala na pisanie i uruchamianie kodu języka tak jak w konsoli, wraz z wyjściem, które również jest prezentowane w przeglądarce. Rezultaty obliczeń można łączyć z wieloma formatami takimi jak **HTML, LaTeX, PNG, SCG, PDF** i inne. Wbudowana została również możliwość pisania własnych widgetów w języku **JavaScript**, które mogą łączyć opcje interfejsu graficznego aplikacji z jądrem. Aplikacja umożliwia również wykorzystanie języka znaczników **Markdown** używanego m.in. do pisania prostych dokumentów, często używanego do opisywania projektów na GitHubie. W notatnikach można również umieszczać równania za pomocą składni **LaTeX**, które są wyświetlane w przeglądarce za pośrednictwem **MathJax**.

Chociaż podstawowym jądrem wykorzystywanym przez notatniki jest jądro z interpreterem Pythona, do dyspozycji użytkowników są również:

- Python(https://github.com/ipython/ipython)
- Julia (https://github.com/JuliaLang/IJulia.jl)
- R (https://github.com/IRkernel/IRkernel)
- Ruby (https://github.com/minrk/iruby)
- Haskell (https://github.com/gibiansky/IHaskell)
- Scala (https://github.com/Bridgewater/scala-notebook)
- node.js (https://gist.github.com/Carreau/4279371)
- Go (https://github.com/takluyver/igo)

Dokumenty (notebook) zawierają wszystkie instrukcje wpisane podczas jego edycji, czyli kod oraz efekty jego uruchomienia, które mogą być wykresami, filmem video itd. Pliki notatników posiadają rozszerzenie **.ipynb**. Komendy wpisywane w notatnikach są zorganizowane w komórki, których podstawowe rodzaje to:

- **komórki kodu** – dotyczy to zarówno kodu wejściowego jak i wyjściowego,
- **komórki Markdown** – tekst, który może zawierać znaczniki Markdown jak i równania LaTeX,
- **komórki nagłówków** – tak jak w HTML mamy do dyspozycji 6 poziomów nagłówków do lepszego zorganizowania dokumentu,
- **„surowe" komórki** – niesformatowany tekst, który bez modyfikacji jest konwertowany na inne formaty z wykorzystaniem nbconvert.

Jako, że notatniki są zapisywane w formacie JSON (mimo innego rozszerzenia pliku), to można je w dość łatwy sposób odczytywać za pomocą wielu istniejących języków programowania i bibliotek. Mimo, że istnieje wiele gotowych rozwiązań do konwersji (HTML, LaTeX, PDF, pokazu slajdów poprzez reveal.js) to dzięki ich elastyczności można dodać nowe.

13.2. URUCHAMIANIE JUPYTER NOTEBOOK

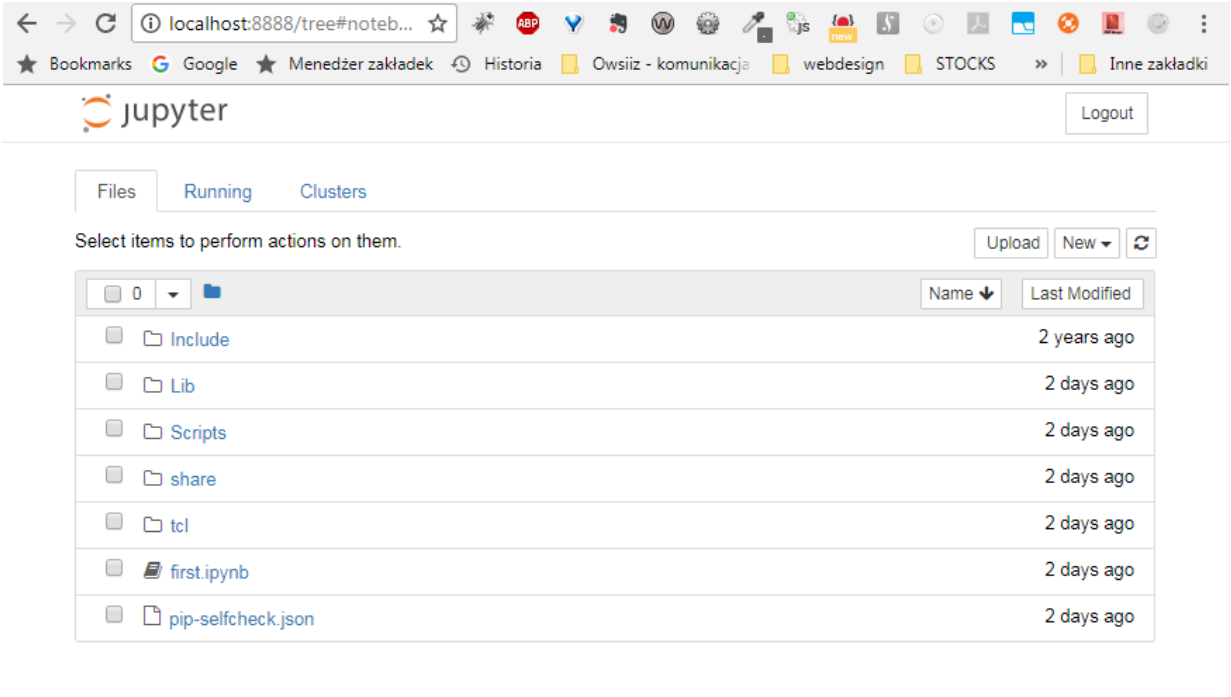
Jupyter Notebook nie jest dołączany do standardowej dystrybucji Pythona więc trzeba go zainstalować samodzielnie. Można to zrobić poprzez narzędzie PIP:

```
pip install jupyter
```

Notatnik uruchamiamy poleceniem:

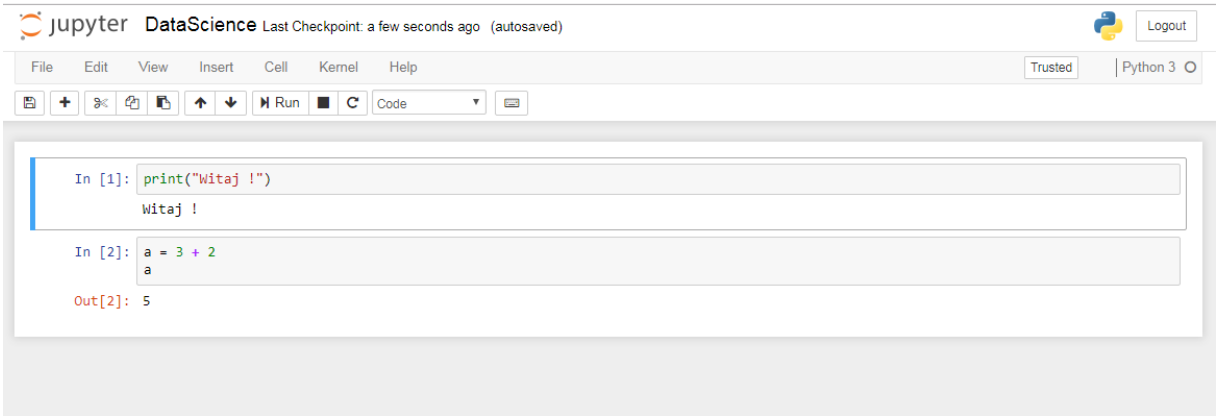
```
jupyter notebook
```

Tym poleceniem uruchamiamy serwer, który następnie wywoła okno przeglądarki z adresem panelu do zarządzania notatnikami. Zrzut ekranu znajduje się na kolejnej stronie.



Z tego miejsca możemy tworzyć nowe notatniki, uruchamiać wcześniej zapisane lub pobrać (upload) istniejące notatniki z innego źrodła. Po zaznaczeniu notatnika pojawiają się dodatkowe opcje, które umożliwiaja zmiane jego nazwy, edycję czy usunięcie.

Widok edycji prezentuje się następująco:



Komórki mogą znajdować się w jednym z dwóch stanów – trybie edycji i trybie zarządzania. Stan ten można odróżnić po wyglądzie ramki. Zielone obramowanie po lewej stronie oznacza tryb edycji (dodatkowo karetka jest ustawiona w aktywnej komórce) a niebieska tryb zarządzania. W trybie edycji możemy zmieniać lub wpisywac polecenia, tekst, itp. a w trybie zarządzania możemy zmieniać ustawienia samego notatnika.

Poniżej znajduje się zbiór przydatnych poleceń, które można uruchomić za pomocą klawiatury.

Enter	Przejście w tryb edycji
Esc	Przejście w tryb zarządzania
Shift + Enter	Wstawienie nowej komórki
Alt + Enter	Uruchamia aktualna komórkę i wstawia nową poniżej
Ctrl + Enter	Uruchamia aktualną komórke i przechodzi w tryb zarządzania
↑ lub k, ↓ lub j	Poruszanie się po komórkach góra-dół
s	Zapisanie notatnika
y, m, 1-6, t	Zmiana typu komórki
a, b	Dodanie komórki przed lub po aktualnie zaznaczonej
x, c, v, d, z	Usuwanie, kopiowanie, wklejanie, usuwanie (wciskamy 2 razy), cofanie
i, 0 (zero)	Przerwanie wykonania kodu przez jądro, restart jądra (po potwierdzeniu)

Pod adresami: https://jupyter-notebook.readthedocs.io/en/latest/examples/Notebook/Working%20With%20Markdown%20Cells.html

oraz https://jupyter-notebook.readthedocs.io/en/latest/examples/Notebook/Typesetting%20Equations.html

można zobaczyć kilka przykładów zastosowania w/w formatów komórek.

Ćwiczenia

1. Stwórz swój własny notatnik, dodaj kilka dowolnych komórek starając się wykonywać jak najwięcej komend przy użyciu klawiatury (można wypróbować przykłady z linków powyżej). Zapisz notatnik pod dowolną nazwą. Pobierz notatnik jako HTML i sprawdź jak wygląda. Zamknij notatnik.
2. Odzwiedź adres https://github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks#social-data i wybierz kilka interesujących Cię notatników. Większość z nich uruchamia się w aplikacji nbviewer co pozwala na łatwe pobieranie całych notatników. Pobierz jeden, zapisz na komputerze lokalnym i uruchom na swoim lokalnym komputerze.
   UWAGA! Większość notatników wymaga dodatkowych bibliotek, które mogą nie być aktualnie zainstalowane. Zaleca się uruchamianie Jupytera w oddzielnym środowisku wirtualnym i pobranie niezbędnych bibliotek przed uruchomieniem notatnika. Informacje o niezbędnych bibliotekach powinny znajdować się na początku notatnika.
3. Zapisz wybrany notatnik jako plik PDF oraz Markdown i zobacz jak wygląda w każdym z tych formatów.

14. DEBUGOWANIE IREFAKTORYZACJA KODU W ŚRODOWISKU PYCHARM

Sposób uruchamiania debuggera oraz część jego możliwości zostanie zaprezentowany w trakcie zajęć a po więcej informacji odsyłam pod adres https://www.jetbrains.com/help/pycharm/debugging.html.

Dodatkowe możliwości Pycharm co do refaktoryzacji kodu można znaleźć w dokumentacji pod adresem https://www.jetbrains.com/help/pycharm/refactoring-source-code.html oraz https://www.jetbrains.com/help/pycharm/pycharm-refactoring-tutorial.html.

15. PROJEKT

Formą zaliczenia przedmiotu jest wykonanie projektu, którego opis znajduje się poniżej. Nie jest konieczne wykonanie 100% opisanej funkcjonalności, aby otrzymać ocenę pozytywną. Praca będzie oceniana pod kątem możliwości danego studenta i jego zaangażowania.

Wymagania projektu

Projekt polega na stworzeniu modułu, który może być zbiorem funkcji i/lub obiektów z metodami i ta druga metoda wydaje się lepszym rozwiązaniem. Moduł będzie operował na wybranym przez studenta datasecie pobranym ręcznie przez studenta na dysk. Zbiory danych można znaleźć pod adresem http://archive.ics.uci.edu/ml/datasets.html . Moduł powinien umożliwiać jego zaimportowanie w interaktywnej konsoli i uruchomienie jego funkcji z wiersza poleceń.

Funkcjonalność, którą ma dostarczać moduł:

- **Wczytanie datasetu** – funkcja, która po podaniu ścieżki (nazwa pliku, jeżeli w tym samym folderze) wczyta dane z pliku do listy. Dodatkowo funkcja przyjmuje parametr określający czy pierwszy wiersz pliku zawiera etykiety kolumn czy nie. Jeżeli tak to etykiety wczytywane są do oddzielnej listy.
- **Wypisanie etykiet** – funkcja wypisująca etykiety lub komunikat, że etykiet nie było w danym datasecie.
- **Wypisanie danych datasetu** – funkcja wypisuje kolejne wiersze datasetu. Bez podania parametrów wypisywany jest cały dataset, ale możliwe też podanie 2 parametrów, które określają przedział, który ma zostać wyświetlony,
- **Podział datasetu na zbiór treningowy, testowy i walidacyjny**. Funkcja przyjmuje 3 parametry określające ile elementów trafia do poszczególnych zbiorów.
- **Wypisz liczbę klas decyzyjnych** – wypisanie ile jest unikalnych klas decyzyjnych (ostatnia kolumna).
- **Wypisz dane dla podanej wartości klasy decyzyjnej** – wypisuje wiersze z zadaną wartością klasy decyzyjnej.
- **Zapisanie danych do pliku csv** – jako parametr przyjmowana jest dowolna lista, która może być podzbiorem datasetu, zmienną przechowującą dane treningowe, itp. Dodatkowo podawana jest nazwa pliku, do którego dane zostaną zapisane.