COMARCH SZKOLENIA

# CENTRUM SZKOLENIOWE COMARCH

**Szkolenie:** Programowanie Python na poziomie podstawowym. Akademia IT

Trener: Tomasz Kaniecki

COMARCH

# Geneza

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.

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.

# Cechy

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.

# Cechy

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 społecznoścs Pythona jest bardzo duża co przekłada się na łatwość uzyskania odpowiedzi na pytania, sporą ilość dobrej dokumentacji oraz rozbudowaną listę bibliotek i dodatków.

# Typy danych

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.

# Operatory

```
# 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"
# jak to zadziała ?
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))
```

# Typy danych

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

# Typy danych

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.W przypadku liczb rzeczywistych można również określić precyzję z jaką zostaną wyświetlone.

```
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
```

# Typy danych

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.

```
imie = "Tomasz"
nazwisko = "Kowalski"
# 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 Anakondą?!".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))
```

COMARCH

# Typy danych

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.

```
imie = "Tomasz"
nazwisko = "Kowalski"
# 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 Anakondą?!".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))
```

COMARCH

# Typy danych

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.

```
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]

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.

# Typy danych

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.

```
# 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())
```

# Typy danych

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))
```

COMARCH

# Typy danych

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.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.

```
# 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
```

# INSTRUKCJE WARUNKOWE

Język Python posiada tylko jedną wbudowaną instrukcję warunkową i jest nią instrukcja if/elif/else. Nie znajdziemy tutaj konstrukcji case/switch (od wersji 3.10 już jest)

```
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")

zbior_dopuszczalny = [1, 3, 5, 7, 9]
if liczba not in zbior_dopuszczalny:
     print("Podana liczba nie znajduje się w zbiorze")
```

# INSTRUKCJE WARUNKOWE

Python posiada typ None, który odpowiada Null znanemu z innych języków oraz baz danych.

```
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")
```

# INSTRUKCJE WARUNKOWE

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

# INSTRUKCJE WARUNKOWE

Od Python 3.10 możemy użyć instrukcji match:

```
http_code = "418"match http_code:
    case "200":
        print("OK")
        do_something_good()
    case "404":
        print("Not Found")
        do_something_bad()
    case "418":
        print("I'm a teapot")
        make_coffee()
    case _:
        print("Code not found")
```

# PĘTLE

W Pythonie mamy do dyspozycji dwie pętle: for oraz while przy czym ta pierwsza jest zdecydowanie bardziej „popularna” wśród programistów Pythona.

```
# 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))

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])
```

# PĘTLE

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.

# Python comprehensions

Mechanizm ten polega na generowaniu kolekcji (lista, słownik, zbiór) na podstawie jednowierszowego zapisu określającego warunki dla zmiennych, które zostaną w danej kolekcji umieszczone.

```
# 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 nawias klamrowy jak dla słowników
lista = [1, 2, 2, 3, 4, 4, 4, 5]
zbior = {i for i in lista}
print(zbior)
```

# ORGANIZACJA KODU WEDŁUG PEP8

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/.

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)
```

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)
```

# ORGANIZACJA KODU WEDŁUG PEP8

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.

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"."""
```

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

# IMPORT

Poszczególne instrukcje importu powinny być rozdzielone na oddzielne linie.
Przykład:

```
import os
import sys

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.

# 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.

```
# 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))
```

# 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
```

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.

# 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.

```
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.

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"))
```

# DEFINIOWANIE FUNKCJI

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())
```

COMARCH

# Lambda

Jej najprostsza postać wygląda tak:
lambda <parametry> : <wyrażenie>

Praktyczne zastosowanie funkcji Lambda

- **Filter, map, reduce.**

Wszystkie 3 funkcje, wykonują operacje na zbiorze danych. Funkcja Map, zamienia elementy zbioru, funkcja filter filtruje zbiór danych, natomiast funkcja reduce, go redukuje. W jaki sposób? W taki, jaki wskażemy naszą funkcją lambda.
Zobaczmy:

```
lista = [1,3,5,7]
from functools import reduce
print(f"Nasza lista: {lista}\n")
print(f"Przykład zastosowania map: {list( map(lambda _: _*2, lista) )}")
print(f"Przykład zastosowania filter: {list( filter(lambda _: _>3, lista) )}")
print(f"Przykład zastosowania reduce: { reduce(lambda x,y: x+y, lista) }")
```

# Lambda

```
wiek = lambda x: "dziecko" if x < 10 else ("Nastolatek" if x < 18 else "Dorosły")
wiek(18)
```

Lambda nie wnosi niczego, czego nie byli byśmy wstanie osiągnąć za pomocą zwykłej funkcji. Jej celem jest jedynie skrócenie zapisu. Trzeba jednak przyznać, że jest bardzo wygodna i jak tylko się ją opanuje, to jej użycie sprawia sporo radości.

# OBIEKTOWY PYTHON

```
"""docstring dla modułu"""

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

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.

```
"""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())
```

COMARCH

# OBIEKTOWY PYTHON

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.

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

```
"""docstring dla modułu"""
class Pojazd:

sygnal = "Piiib piiib"
""" 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

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)
```

COMARCH

# Obsługa plików

```
uchwyt = open("plik.txt")
uchwyt = open(r"C:\Users\python\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 „\”.

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

# Pliki CSV

Jak zwykle, mamy gotowy moduł w standardowej bibliotece służący do obsługi tego typu formatu. Standardowo zaczynamy od jego importu.

```
import csv
```

## Odczyt

Skoro jest to plik tekstowy, to otwieramy go identycznie jak standardowy plik tekstowy

```
# konstrukcja *with* pozwala na otworzenie pliku i korzystanie z niego wewnątrz niej
# po jej opuszczeniu automatycznie zamknie strumień odczytu
with open('plik.csv', 'r') as csvfile:
    # deklarujemy nasz *czytacz*
    # parametr *delimiter* jest opcjonalny i wskazuje jaki został w pliku użyty separator
    csvreader = csv.reader(csvfile, delimiter=',')
```

Otwierając plik, chcemy wskazać jego kodowanie (nie musi to być w każdym przypadku `utf-8`, ktoś mógł użyc innego kodowania do zapisu tego pliku, ale jeżeli są *krzaki* to najłatwiej jest spróbować otworzyć z `utf-8`).

```
# pojawia się nowy paramet encoding ustawiony na utf-8
with open('plik.csv', 'r', encoding='utf-8') as csvfile:
    csvreader = csv.reader(csvfile, delimiter=',')
```

# Pliki CSV

**Zapis**

```
# zwroc uwage na zmiane symbolu trybu otwarcia pliku 'r' na 'w'
# 'r' to read czyli odczyt, 'w' oznacza write czyli zapis

with open('plik.csv', 'w', encoding='utf-8') as csvfile:
    # inicjujemy *zapisywacz*
    csvwriter = csv.writer(csvfile)
    # wpisujemy pierwsza linie naszego pliku CSV (nazwy kolumn)
    csvwriter.writerow('kolumna 1', 'kolumna 2', 'kolumna 3')
    # czas na kolejne linie z wartosciami
    csvwriter.writerow('wartosc1_kol1', 'wartosc1_kol2', 'wartosc1_kol3')
    csvwriter.writerow('wartosc2_kol1', 'wartosc2_kol2', 'wartosc2_kol3')
    csvwriter.writerow('wartosc3_kol1', 'wartosc3_kol2', 'wartosc3_kol3')
```

# PyDoc

Dokumentacje generujemy za pomocą """ """

```
"
This example module shows various types of documentation available for use
with pydoc.  To generate HTML documentation for this module issue the
command:

    pydoc -w foo

"""

class Foo(object):
    """
    Foo encapsulates a name and an age.
    """
    def __init__(self, name, age):
        """
        Construct a new 'Foo' object.

        :param name: The name of foo
        :param age: The ageof foo
        :return: returns nothing
        """
        self.name = name
        self.age = age

def bar(baz):
    """
    Prints baz to the display.
    """
    print baz

if __name__ == '__main__':
    f = Foo('John Doe', 42)
    bar("hello world")
```

Generowanie dokumentu: pydoc myModule.thefilename

COMARCH

# XML w Python

Obsługa XMLa w Pythonie jest pełna i zapewniana przez wiele modułów.

```
<znajomi>
	<osoba>
		<imie foo="zzz">Zygmunt</imie>
		<email>1@1.pl</email>
	</osoba>
	<osoba>
		<imie foo="aaaa">Janina</imie>
		<email>2@2.pl</email>
	</osoba>
</znajomi>
```

A oto pierwszy kod Pythona:

```
from xml.dom import minidom

#otwieramy plik w parserze
DOMTree = minidom.parse('plik.xml')

print DOMTree.toxml()
```

COMARCH

# XML w Python

Kod na poprzedniej stronie otwiera plik z kodem XML i zwraca obiekt minidom pod zmienną "**DOMTree**". Zastosowana na końcu metoda **.toxml()** służy do wyświetlania kodu XML - powyższy kod wyświetli całą strukturę XMLa z naszego przykładowego pliku. **.toxml()** jest dość przydatne, gdyż przeglądając dzieci danego taga XML możemy szybko sprawdzić jak wygląda sytuacja (na jakim poziomie jesteśmy i jakie dane mamy dostępne). Oto kolejny przykład:

```
from xml.dom import minidom

#Open XML document using minidom parser
DOMTree = minidom.parse('plik.xml')

#pobieramy elementy struktury dokumentu XML
cNodes = DOMTree.childNodes
# struktura pierwszego dziecka "osoba"
print cNodes[0].getElementsByTagName("osoba")[0].toxml()
```

Metoda **DOMTree.childNodes** zwraca obiekt "DOM Element" w tupli, dzięki któremu możemy uzyskać dostęp do elementów dokumentu XML. Kolejna metoda **getElementsByTagName("NAZWA")** umożliwia dostęp do taga-dziecka o podanej nazwie (też przez tuplę).

COMARCH SZKOLENIA

# Dziękujemy za uwagę

**Centrum Szkoleniowo-Konferencyjne Comarch**

ul. prof. M. Życzkowskiego 23, 31-864 Kraków