

CENTRUM SZKOLENIOWE COMARCH

Szkolenie: Python dla zaawansowanych

Trener: Radosław Janiak

Środowisko pracy, Interpreter języka Python

Podstawy języka Python

Programowanie funkcyjne

Programowanie obiektowe

Moduły

Obsługa plików i źródeł danych

Współbieżność i równoległość

Wydajność i optymalizacja

Zaawansowane aspekty programowania w języku Python

Programowanie sieciowe

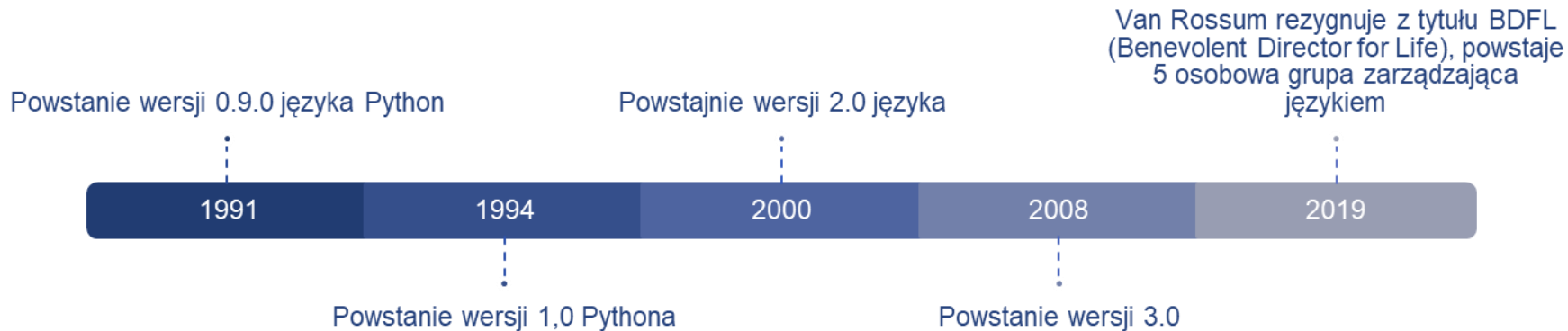
Wybrane aspekty zastosowań bibliotek Pythona



Historia języka Python:

- Python jest aktualnie jednym z najbardziej popularnych języków programowania wysokiego poziomu o bardzo różnorodnym zakresie funkcjonalnym. Język ten został zaprojektowany przez Holendra Guido van Rossum w 1990 roku a następnie dokumentacja została przygotowana przez instytucję Python Software Foundation (Wiskunde & Informatica (CWI) Amsterdam) . Idea to połączenie minimalizmu w kodzie z indywidualizmem rozwiązań oraz implemetacji.
- Guido jest fanem serialu telewizyjnego BBC „Latający cyrk Monty Pythona”. Nazwa była inspirowana tym właśnie serialem.
- Prekursorem języka Python był język ABC, który stał się API dla systemu Amoeba
- Język Python w swojej konstrukcji posiada najlepsze cechy języka ABC oraz języka Modula-3
- Python na swojej drodze rozwoju czerpał z wielu inspiracji łącznie z takimi językami jak C, Perl czy Java.

Kalendarium



Co dalej z językiem Python

Która wersja Pythona będzie następną? Czy zmiana wersji z 3 na 4?
Czy kolejna subwersja czyli 3.x

Python
3.12



Python 4

Guido Van Rossum (twórca języka Python) nie zostawia żadnych wątpliwości, że na zmianę głównej wersji należy jeszcze poczekać. Prawdopodobnie niedługo pojawi się wersja 3.10, która ma wprowadzić kilka nowości.

Wersjonowanie języka Python

Python R.E.X np. 3.9.6

R – wersja główna (pierwsza cyfra wersji), aktualnie nr 3. Każda nowa wersja wprowadza istotne i kluczowe zmiany w składni języka, zmienia sposób interpretacji programu

E – wersja drugorzędna (druga wartość po kropce), aktualnie nr 9. Każda nowa wersja wprowadza ważne zmiany w składni języka, jednak nie zmienia sposobu interpretacji programu

X – wersja trzeciorzędna (trzecia wartość po kropce), aktualnie nr 6. Każda nowa wersja wprowadza drobne zmiany w składni języka, usprawnia interpreter i optymalizuje rozwiązania

Python 3.8.4

Python 2.7.1

Python 3.6.5



Przykładowe wersje języka Python

Idea języka:

Idea nadania ludziom pierwszeństwa nad maszynami jest podstawą filozofii stojącej za Pythonem. Z pewnością fakt, że jest to język interpretowany, w przeciwieństwie do języka skompilowanego, oznacza, że programista otrzymuje natychmiastową informację zwrotną na temat kodu, który pisze, bez konieczności poświęcania czasu na jego rekompilację po dokonaniu każdej zmiany.

„Istnieje cała masa typowych zadań programistycznych, które są łatwe w Pythonie”

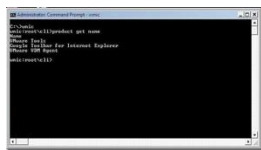
/Guido van Rossum/

„W Pythonie każdy symbol, który wpisujesz, jest niezbędny”

/Guido van Rossum/

Zastosowania języka Python

Aplikacja konsoli



Aplikacja rocket science



Aplikacje Web



Język Python

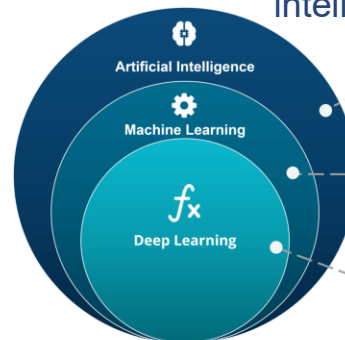


Zastosowania w aplikacjach znanych firm



Google
NETFLIX

Aplikacje AI, ML, DL – sztuczna inteligencja



ARTIFICIAL INTELLIGENCE
A technique which enables machines to mimic human behaviour

MACHINE LEARNING
Subset of AI technique which use statistical methods to enable machines to improve with experience

DEEP LEARNING
Subset of ML which make the computation of multi-layer neural network feasible



Analiza danych, bazy

Zastosowania języka Python

Język Python ma bardzo wiele zastosowań, wymienimy najważniejsze:

1. Aplikacje standardowe (konsola) – wykorzystanie podstawowych elementów języka
2. Aplikacje bazodanowe, analiza danych, analiza struktur danych, Big Data, Data Science
3. Aplikacje Web – platformy: Django, Flask, Bottle, Jade - wysoki poziom bezpieczeństwa
4. Aplikacje AI(Sztuczna Inteligencja) - w tym algorytmy Machine Learning i Deep Learning, jest wiele bibliotek pozwalających na programowanie tego typu algorytmów takich jak sieci neuronowe, genetyka itd. (scikit-learn, keras)
5. Rocket Science - programowanie eksperymentalnych algorytmów oraz struktur danych z użyciem wyników nowoczesnych badań naukowych.

Firmy takie jak: Netflix, Google, Facebook, Spotify i wiele innych korzystają w swoich rozwiązaniach z języka Python. Prowadzą badania naukowe oraz sam tworzą nowe biblioteki dla Pythona udostępniając je społeczności programistycznej.

Specyfikacja PEP

- Specyfikacja PEP 8 (ang. *Python Enhancement Proposal #8*) to styl określający sposób formatowania kodu utworzonego w Pythonie.
- Oczywiście można tworzyć kod Pythona w dowolny sposób, pod warunkiem że jego składnia jest prawidłowa.
- Użycie spójnego stylu powoduje, że kod staje się przystępniejszy i łatwiejszy w odczycie.

Znaki odstępu

W Pythonie znaki odstępu mają duże znaczenie dla składni. Dlatego też programiści Pythona są szczególnie wyczuleni na wpływ, jaki znaki odstępu mają na czytelność kodu. Stosuj się do następujących reguł dotyczących znaków odstępu:

- We wcięciach stosuj spacje zamiast tabulatorów.
- Każdy poziom wcięcia ważnego z punktu widzenia składni powinien składać się z czterech spacji.
- Wiersz powinien mieć długość maksymalnie 79 znaków.
- Jeżeli długie wyrażenie jest kontynuowane w kolejnych wierszach, wówczas należy je wciąć o cztery dodatkowe spacje względem standardowego poziomu wcięcia.
- W pliku funkcje i klasy powinny być rozdzielane dwoma pustymi wierszami.
- W klasie metody powinny być rozdzielone pustym wierszem.

- W słowniku nie należy umieszczać spacji między nazwą klucza i dwukropkiem. Należy umieścić jedną spację przed wartością klucza, o ile mieści się ona w tym samym wierszu.
- Przed operatorem przypisania (=) i po nim należy umieszczać tylko i wyłącznie jedną spację.
- W przypadku adnotacji typów nie można używać spacji między nazwą zmiennej i dwukropkiem oraz trzeba użyć jednej spacji przed informacjami o typie.

Konwencje nazw

- Specyfikacja PEP 8 sugeruje użycie unikatowych stylów nadawania nazw w różnych częściach języka. Dzięki temu podczas odczytu kodu źródłowego można łatwo ustalić, do czego odnoszą się poszczególne nazwy. Stosuj się do następujących reguł dotyczących konwencji nazw:
- Funkcje, zmienne i atrybuty powinny być zapisywane w formacie małe_litery_ podkreślenie.

- Chronione atrybuty egzemplarza powinny być zapisywane w formacie `_podkreślenie_nazwa`.
- Prywatne atrybuty egzemplarza powinny być zapisywane w formacie `__podwójne_podkreślenie_nazwa`.
- Klasy i wyjątki powinny być zapisywane w formacie `NazwaKlasyLubWyjątku`.
- stałe na poziomie modułu powinny być zapisywane `WIELKIMI_LITERAMI`.
- Metody egzemplarza w klasie powinny używać `self` jako nazwy pierwszego parametru (który odwołuje się do obiektu).
- Metody klasy w klasie powinny używać `cls` jako nazwy pierwszego parametru (który odwołuje się do klasy).

Polecenia i wyrażenia

- Według *filozofii Pythona* „powinien być jeden — i najlepiej tylko jeden — oczywisty sposób na wykonanie danego zadania”. Specyfikacja PEP 8 próbuje to skodyfikować w wytycznych dotyczących wyrażień i poleceń.
- Używaj negacji typu inline (`if a is not b`) zamiast negacji wyrażień pozytywnych (`if not a is b`).
- Nie sprawdzaj pod kątem pustych wartości kontenerów lub sekwencji (na przykład `[]` lub `''`) za pomocą operacji porównania z zerem wielkości danego elementu (`if len(lista) == 0`). Zamiast tego użyj `if not lista` i przyjmij założenie, że puste wartości są uznawane za `False`.

- To samo dotyczy niepustych wartości kontenerów lub sekwencji (na przykład [1] lub 'witaj'). Polecenie if lista przyjmuje wartość True dla niepustych wartości.
- Unikaj jednowierszowych poleceń if oraz pętli for i while, z wyjątkiem poleceń złożonych.
- W celu zapewnienia większej czytelności wymienione polecenia powinny być umieszczane w wielu wierszach.
- Jeżeli wyrażenie nie mieści się w jednym wierszu, ujmij je w nawiasy oraz dodaj znaki nowego wiersza i wcięcia, aby ułatwić jego odczyt.
- Preferowane jest ujmowanie w nawiasy wyrażień wielowierszowych zamiast stosowania znaku kontynuowania wiersza (\).

Polecenia import

Specyfikacja PEP 8 sugeruje przestrzeganie pewnych reguł dotyczących sposobu stosowania poleceń importujących moduły w kodzie:

- Polecenia `import` (a także `from x import y`) zawsze umieszczaj na początku pliku.
- Podczas importu modułów zawsze stosuj bezwzględne nazwy plików, a nie nazwy względne dla ścieżki dostępu dla bieżącego modułu. Na przykład w celu importu modułu `foo` z pakietu `bar` powinieneś użyć polecenia `from bar import foo` zamiast po prostu `import foo`.
- Jeżeli koniecznie musisz użyć względnej ścieżki podczas importu, to zastosuj składnię `from . import foo`.
- Polecenia importujące powinny się znajdować w sekcjach zachowujących następującą kolejność: moduły biblioteki standardowej, moduły firm trzecich, własne moduły. W poszczególnych sekcjach polecenia importujące należy umieścić w kolejności alfabetycznej.

Przykład PEP8

Nazwa klasy z dużej litery

Odległość pustego wiersza pomiędzy elementami klasy

Wcięcie o rozmiarze 4 spacje

Nazwa funkcji małymi literami

```
class Student(Employee, Sport, ExtraData):
```

```
    def __init__(self, name, age, student_id, grad_year, firm="", position="", kind="", best_score="", years=""):
```

```
        Employee.__init__(self, name, age, firm, position)
```

```
        Sport.__init__(self, kind, best_score, years)
```

```
        self.student_id = student_id
```

```
        self.grad_year = grad_year
```

```
    def print_student(self):
```

```
        print(f"Informacja o studencie: id: {self.student_id}, rok ukończenia studiów: {self.grad_year}")
```

```
    def czyPracownik(self):
```

```
        if self.firm == "":
```

```
            return False
```

```
        else:
```

```
            return True
```

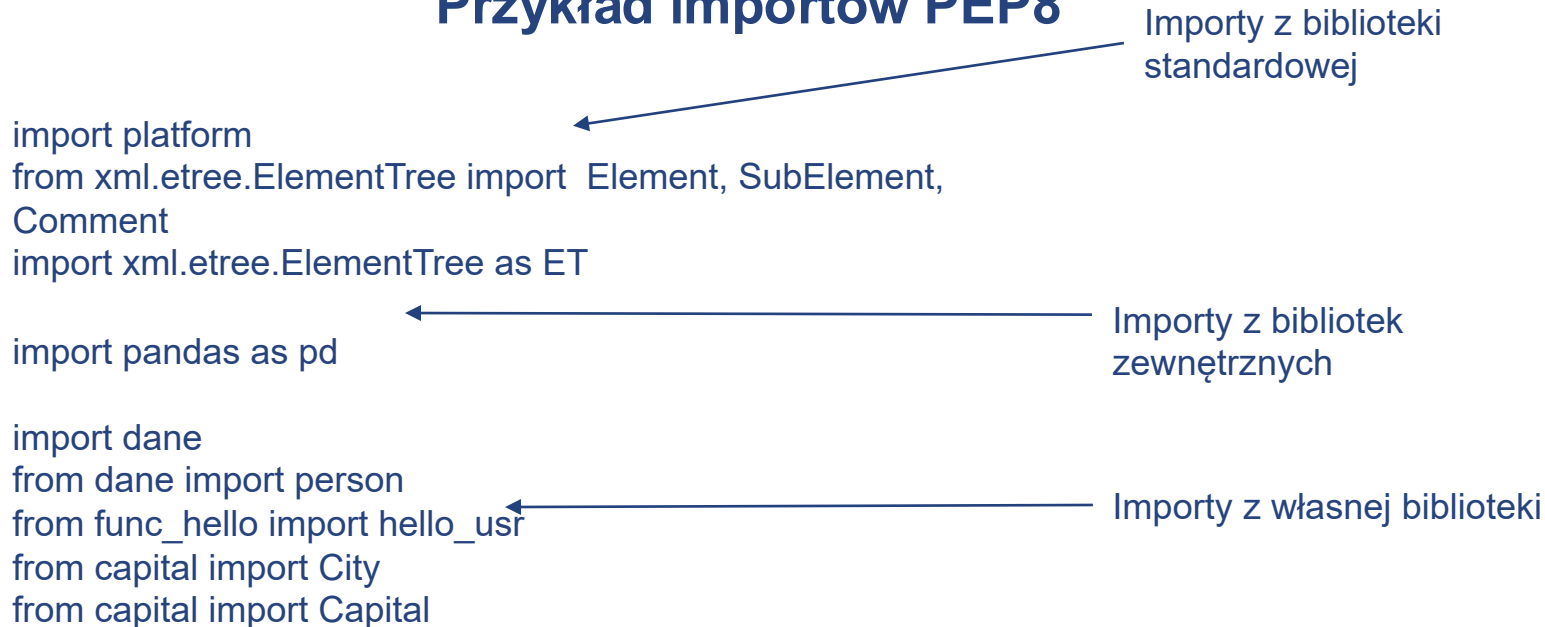
Metoda egzemplarza definiowana przez self

Nie stawiamy spacji przed dwukropkiem

Przykład importów PEP8

```
import platform
from xml.etree.ElementTree import Element, SubElement,
Comment
import xml.etree.ElementTree as ET
```

Importy z biblioteki
standardowej



```
graph RL
    A[Importy z biblioteki standardowej] --> B[import platform  
from xml.etree.ElementTree import Element, SubElement,  
Comment  
import xml.etree.ElementTree as ET]
    C[Importy z bibliotek zewnętrznych] --> D[import pandas as pd]
    E[Importy z własnej biblioteki] --> F[from dane import person  
from func_hello import hello_usr  
from capital import City  
from capital import Capital]
```

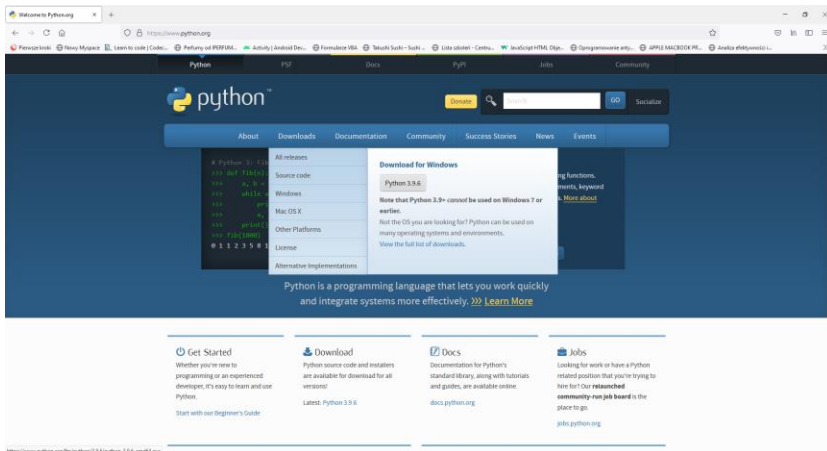
```
import pandas as pd
```

Importy z bibliotek
zewnętrznych

```
import dane
from dane import person
from func_hello import hello_usr
from capital import City
from capital import Capital
```

Importy z własnej biblioteki

Instalator interpretera Python



Interpreter języka Python, jest dostępny na stronie: <https://www.python.org/>

Można pobrać najnowszą lub starsze wersje interpretera.

Pobranie i instalacja interpretera języka Python to pierwszy krok, który należy zrobić żeby programować w tym języku

Środowisko programistyczne

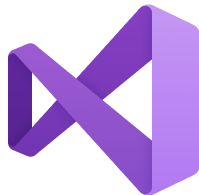
Środowisko programistyczne, jest to kompletny zestaw narzędzi służących do tworzenia programów w wybranym języku. Środowisko takie pozwala tworzyć rozwiązania i projekty, projektować graficznie aplikację, nadzorować pracę nad testami, analizować kod źródłowy oraz właściwości elementów i obiektów.

Nie ma jednego dedykowanego środowiska programistycznego dla języka Python. Jakże można wybrać najlepsze środowiska języka Python

JetBrains PyCharm



Microsoft Visual Studio 2019



Microsoft Visual Studio Code



JetBrains PyCharm

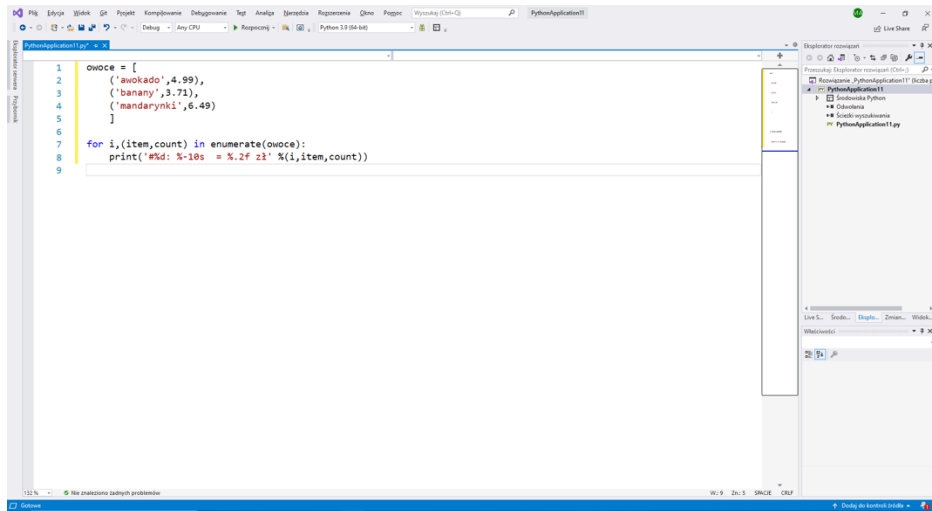


Jest to wygodne i kompletne środowisko programistyczne dla języka Python. Występuje w dwóch wersjach: bezpłatnej - Community oraz płatnej – Professional.

Wersja Community obsługuje tylko projekty konsolowe, natomiast wersja Professional wszystkie możliwe projekty i rozwiązania dostępne dla tego języka.

Link do pobrania pliku instalacyjnego(wersja Community i Professional):
<https://www.jetbrains.com/pycharm/download/#section=windows>

Microsoft Visual Studio 2019



Jest to kompletne multi - środowisko programistyczne firmy Microsoft dla wielu języków, w tym języka Python. Występuje w dwóch wersjach: bezpłatnej - Community oraz płatnych: Professional i Enterprise. Wersja Community obsługuje pełne spektrum projektów, łącznie z okienkowymi i webowymi/

Link do pobrania środowiska:
<https://visualstudio.microsoft.com/pl/vs/>

Typy i zmienne

Zmienna (ang. variable) to miejsce w pamięci operacyjnej reprezentowane przez adres komórki pamięci, które przechowuje pojedynczą wartość określonego typu. Każda zmienna ma nazwę, dzięki której można się do niej odwoływać.

Typ określa rodzaj informacji, jakie można przechowywać w naszej zmiennej. W Pythonie typem może być: ciąg znaków, liczba oraz tak zwane typy złożone czyli zbiory elementów

Wprowadzenie do składni języka Python

```
print(„aktualnie jest lato!”)
```

Funkcja print() wyświetla ciąg string(ciąg znaków) na ekranie

```
a = „Później będzie  
jesień.”
```

Zmienna a w sposób dynamiczny reprezentuje kolejny ciąg znaków

```
print (a)
```

Funkcja print() wyświetla zawartość zmiennej a

Zmienne

```
x = "tekst"  
x = 5  
x = 1.33  
x = True
```

Jeżeli stosujemy jedną wspólną nazwę dla wielu zmiennych to taką nazwę określamy wspólnym interfejsem zmiennej i każda kolejna zdana wartość przesłania poprzednio zdefiniowaną.

Taki sposób deklarowania danych ma charakter dynamiczny tzn. język Python sam przydziela typ wartości w zależności od rodzaju zadanej wartości

```
g:int = 6  
print(g)
```

W języku Python istnieje też możliwość statycznego typowania, czyli samodzielnego definiowania typu który przyjmuje zmienna – robimy to pisząc po nazwie zmiennej dwukropek, a po nim opisując jawnie typ danych np. int (liczba całkowita)

```
a = "Lato"  
b = "następna pora:  
jesień"  
print (a,b)  
print (a+ " " + b)
```

Zmienne możemy wyświetlać na różne sposoby:
Używając symbolu „+” czyli łączenie ciągów znakowych
lub
Pisząc zmienne po przecinku – czyli wyświetlanie jako obiektów kolekcji


```
▶ 0.0s
1 s = "Nie! "
2 print (5*s)
3 x = "7"
4 print (x*3)
5
```

```
Nie! Nie! Nie! Nie! Nie!
777
```

obliczenie
wartosci

```
▶ 0.0s
1 x = eval(x)
2 print (x*3)
3
```

21

Jeśli stosujemy operator mnożenia w stosunku do ciągów string otrzymamy wielokrotność tego ciągu

Komentarz – czyli linię nieinterpretowaną piszemy po znaku #

Funkcja eval() pobiera dane numeryczne z ciągu string i pozwala wykonać właściwe obliczenie na danych numerycznych np. mnożenie arytmetyczne

Podział operatorów

Arytmetyczne

Relacyjne

Logiczne

Bitowe

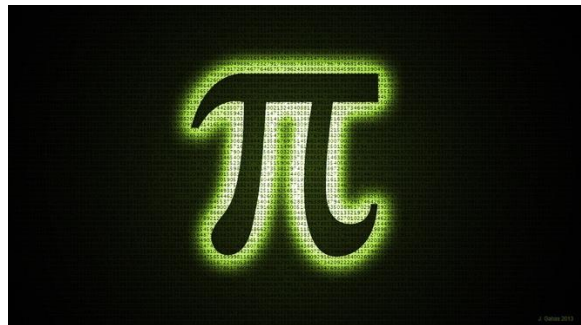
Przynależności

Tożsamości

Typy danych

Python obsługuje następujące typy danych:

- numeryczne (liczbowe): int, float, complex
- tekstowe: str
- sekwencji: list, tuple
- odwzorowania (mapping type): dict
- zestawów (set types): set, frozenset
- logiczne: bool
- binarne: bytes, bytearray



Co to jest zmienna?

Zmienna to konstrukcja programistyczna która posiada trzy podstawowe atrybuty:

- symboliczną nazwę,
- adres pamięci, czyli miejsce przechowywania
- wartość, która pozwala w kodzie źródłowym odwoływać się przy jej pomocy do wartości lub miejsca przechowywania.

Deklaracja zmiennej w Pythonie:

zmienna = 299 (zmiennej o nazwie „zmienna” przypisana została liczba 299)

a = 88.99 (zmiennej o nazwie „a” przypisana została liczba 88.99)

info = True (zmiennej o nazwie „info” przypisana została wartość logiczna True)

Typy liczbowe

Python jest językiem obiekowym i każda zmienna w Pythonie jest obiektem.

Python obsługuje trzy typy liczbowe – liczby całkowite (ang. integers w skrócie int), rzeczywiste (float) i liczby zespolone.

Tworzenie liczb całkowitych:

`k=8` (przypisanie do zmiennej "k" liczby 8)

`z=899` (przypisanie do zmiennej "z" liczby 899)

Tworzenie liczb rzeczywistych - stosujemy kropkę do oddzielenia części dziesiętnej :

`k=0.0976` (przypisanie do zmiennej "k" liczby 0.0976)

`z=455.55434343254` (przypisanie do zmiennej "z" liczby 455.55434343254)

Liczby zmiennoprzecinkowe

Liczby rzeczywiste zazwyczaj zapisuje się w pamięci komputera za pomocą tzw. techniki zmiennego przecinka (z ang. floating point), stąd często stosowana nazwa – liczba zmiennoprzecinkowa. Praktyczną konsekwencją jest to, że liczba jest zapisywana z określoną dokładnością.

Jak ustawić precyzję, z jaką wyświetlaną jest liczba rzeczywista?

```
x = 4.3
```

```
print(x)
```

```
print('%.20f % x)
```

 - % oznacza miejsce wstawienia dowolnej wartości zewnętrznej. Zapis .20f oznacza zaokrąglenie do 2 miejsc po przecinku a f liczbę typu float (rzeczywista)

Napisy

Napis (string) jest inicjowany za pomocą pojedynczego lub podwójnego cudzysłowu.

```
napis1 = 'witaj'  
napis2 = "witaj"  
napis 1 == napis2 # zwróci "True"
```

Typ bool

Typ logiczny bool to dwie stałe: True (w starszych wersjach Pythona równe 1) i False (w starszych wersjach Pythona równe 0).

Python jako fałsz logiczny traktuje:

- liczbę zero (0, 0.0, 0e0, 0j, itp.),
- False,
- None (null),
- puste kolekcje (pusta lista, tuple, słownik, itp.),
- puste napisy,
- obiekty posiadające metodę `__nonzero__()`, jeśli zwraca ona False lub 0.

Wszystko inne jest prawdą logiczną.

Kolekcje

Python posiada cztery podstawowe kolekcje:

1. Listy (list)
2. Krotki (tuple)
3. Zbiory (set)
4. Słowniki (dictionary)

Kolekcja to taki zestaw danych, który łączy w sobie inne dane lub zestawy danych. Kolekcja jest swoistym obiektem a inne dane traktuje również jako obiekty.

Lista (list)

Lista to najbardziej elastyczny typem obiektu uporządkowanej struktury danych.

Lista (list) w Pythonie cechuje się trzema podstawowymi charakterystykami:

- jest tzw. container sequence, czyli może przechowywać elementy różnego typu:

Elementy listy mogą reprezentować różne typy danych:

- Liczby
- Łańcuchy znaków
- Wartości bool
- Wielokrotnie zagnieżdżone listy.

Lista

- jest strukturą mutowalną, czyli możemy do niej elementy dodawać, odejmować, zmieniać ich wartość lub kolejność itp. bez tworzenia nowego obiektu,
- gwarantuje nam zachowanie kolejności elementów, w jakiej zostały one dodane do listy (chyba że sami jawnie zmienimy ich kolejność, np. w wyniku sortowania czy odwrócenia listy).

Lista nie przechowuje wartości swoich elementów fizycznie w obrębie swojej przestrzeni w pamięci (w przeciwieństwie, na przykład, do sekwencji bytes czy array), a jedynie posiada referencję do każdego z obiektów, będących jej elementami.

Inicjalizacja listy

Listę tworzymy używając tzw. literału listy, czyli po prostu nawiasów kwadratowych: [], w środku podając kolejne elementy po przecinku:

```
samochody = ["bmw", "audi", "mercedes", "toyota", "nissan"]
```

```
Liczby = ["34", "122", "111", "1.988", "0"]
```

Podstawowe operacje na listach

Operacje na listach wykonujemy z pomocą odpowiednich funkcji:

`lista.append(„nazwa”)` – dodawanie elementu do listy na pierwszej wolnej pozycji

`lista.sort()` - sortowanie listy od A do Z

`lista.reverse()` lub `lista.sort(reverse=True)` – sortowanie listy od Z do A

`liczby.remove(0)` - usunięcie elementu o wartości 0 z listy o nazwie liczby

Łączenie ze sobą tablic:

```
▶ 0.0s
1 miasto = ["Kraków", "warszawa", "Gdańsk"]
2 city = ["Rzym", "Londyn", "Oslo"]
3 m = miasto + city
4 print(m)
5
```

```
['Kraków', 'warszawa', 'Gdańsk', 'Rzym', 'Londyn', 'Oslo']
```

Dostęp do elementów

Do elementów listy odwołujemy się za pomocą ich indeksów. W Pythonie, tak jak w większości prawidłowych języków programowania, elementy indeksowane są od zera.

Pierwszy element listy to `lista[0]`

Trzydziesty element listy to `lista[29]`

Ostatni element listy to `lista[-1]`

Wyświetlanie cykliczne elementów z listy

```
▶ 0.0s
1 owoce = ["jabłko", "kiwi", "banan", "cytryna", "mango"]
2
3 for x in owoce:
4 |     print(x)
5
```

```
jabłko
kiwi
banan
cytryna
mango
```

Najszybciej elementy listy wyświetlimy za pomocą pętli typu for, gdzie wartość x jest pojedynczym elementem listy a owoce jest zadeklarowaną listą.

Instrukcja jest wykonywana aż do ostatniego istniejącego elementu.

KROTKA(TUPLE)

Krotka jest to kolekcja która odwzorowuje pewien podzbiór iloczynu kartezjańskiego zbiorów, co w praktyce oznacza że jest to rekord tabeli bazy danych lub po prostu wiersz w tabeli. Krotka jest kolekcją uporządkowaną.

Krotki przechowują stałe wartości o różnych typach danych – nie można zmodyfikować żadnego elementu.

Krotka w ogólnym ujęciu jest listą, której nie da się zmodyfikować – czyli listą niemutowalną.

Odczyt elementu podobnie jak w przypadku listy wymaga podania jego indeksu liczbowego.

Tworzenie krotki (tuple)

Krotkę tworzymy używając tzw. literału krotki, czyli po prostu nawiasów zwykłych: (), w środku podając kolejne elementy po przecinku:

```
samochody = ("bmw", "audi", "mercedes", "toyota", "nissan")
```

```
Liczby = ("34", "122", "111", "1.988", "0")
```

Operacje na krotkach

Niewiele jest dozwolonych operacji na krotkach w porównaniu z listami.

Możemy wyświetlać elementy podając indeks elementu w krotce np. `animal[5]`

Możemy łączyć krotki ze sobą tak jak listy poprzez użycie operatora `+` :

```
an = animal + anim2
```

Możemy konwertować listę na krotkę lub krotkę na listę:

`mojalista` jest to zmienna przechowująca listę zbudowaną za pomocą funkcji `list()` wykonanej na krotce `mojakrotka`

```
mojalista = list(mojakrotka)
```

Drugakrotka jest krotką, która powstała przez przetworzenie listy `mojalista` za pomocą funkcji `tuple()`

```
drugakrotka = tuple(mojalista)
```

Możemy wyświetlać za pomocą pętli elementy krotki (po indeksach) – w identyczny sposób tak jak w przypadku listy:

```
for a in animal:
```

```
    print(a)
```

Gdzie `a` jest pojedynczym elementem listy `animal`

Przykład operacji na krotkach

▶ 0.0s

```
1 animal = ("pies", "kot", "papuga", "królik", "mysz")
2 anim2 = ("ryba", "pająk")
3 an = animal + anim2
4 print(an)
5 mojakrotka = tuple(("ob34", 67, 3.33, True))
6 mojalista = list(mojakrotka)
7 drugakrotka = tuple(mojalista)
8 print(drugakrotka)
9
.0 for a in an:
.1 |     print(a)
.2
```

('pies', 'kot', 'papuga', 'królik', 'mysz', 'ryba', 'pająk')

('ob34', 67, 3.33, True)

pies

kot

papuga

królik

mysz

ryba

pająk

Sprawdzanie zawartości krotki

W przypadku krotek istnieje możliwość sprawdzenia logicznego czy element znajduje się w krotce czy też nie:

```
if "papuga" in animal:  
    print("Tak, papuga to zwierzę!")
```

W drugim przypadku można dodać blok else, które będzie reagował również w czasie sytuacji gdy elementu nie ma w krotce.

```
if "ferrari" in animal:  
    print("Nie to błąd!")  
else:  
    print("nie mam ferrari!")
```

Krotka - przykład

```
▶ 0.0s
1 animal = ("pies","kot","papuga","królik","mysz")
2 if "papuga" in animal:
3 |     print("Tak, papuga to zwierzę!")
4 if "ferrari" in animal:
5 |     print("Nie to błąd!")
6 else:
7 |     print("nie mam ferrari!")
8
```

Tak, papuga to zwierzę!
nie mam ferrari!

Słownik (dictionary)

Słowniki są to struktury asocjacyjne czyli takie gdzie każda wartość opiswana jest parą (klucz, wartość) . Klucz opisuje indeks pola wartości czyli definiuje jego charakterystyczną nazwę po której można odszukać konkretną wartość. Z tego wynika, że klucz jest wartością niepowtarzalną. Słowniki definiujemy je za pomocą nawiasów: {}

Każdy z elementów słownika podzielony jest na dwie składowe:

- key – klucz elementu
- value – wartość elementu

Klucz i wartość rozdzielane są znakiem dwukropka :, elementy natomiast standardowo przecinkiem. Jak można się domyślać, klucze w słownikach są unikalne.

Indeksowanie elementów słownika

Bardzo ciekawą cechą słowników w języku Python jest to, że tak jak w przypadku list i krotek elementy słownika posiadają swoje identyfikatory, czyli swoiste indeksy nazywane kluczami. Indeks dla listy i krotki zależy od kolejności wpisania elementów. Pozycja elementu odpowiada za wartość indeksu, które są numerowane automatycznie od wartości 0. W przypadku słownika kolejność nie jest ważna. Ważna jest wartość klucza, który opisuje i indeksuje elementy.

W łatwy sposób dzięki obecności klucza możemy dodawać i podmieniać wartości. Klucz może posiadać wartość liczbową jak również tekstową.

Przykład słownika

```
auto = {  
    "marka": "Ford",  
    "model": "Mustang",  
    "rocznik": 1976  
}
```

W powyższym przykładzie auto jest nazwą słownika. Słownik deklarujemy poprzez zadanie par klucz:wartość. Kluczami są w tym przypadku: marka, model i rocznik, a wartościami:

Ford, Mustang, 1976

Każda z par wymieniana jest po przecinku, a całość zamknięta nawiasami klamrowymi

Wyświetlanie elementów realizuje się za pomocą klucza:

```
m = auto["model"]
```

Wyświetlanie struktur słownika

`print(auto.items())` – w ten sposób wyświetlamy pary (klucz, wartość) w postaci krotek

`print(auto.keys())` - w ten sposób wyświetlamy klucze w formie listy

`print(auto.values())` - w ten sposób wyświetlamy wartości w formie listy

```
for x,y in auto.items():  
    print(x,":",y)
```

Przykład na wyświetlanie danych ze słownika

```
▶ 0.0s
1 auto = {
2     "marka":"Ford",
3     "model":"Mustang",
4     "rocznik":1976
5 }
6 m = auto["model"]
7 print(auto.items())
8 print(auto.keys())
9 print(auto.values())
10 for x,y in auto.items():
11     print(x,":",y)
12
```

```
dict_items([('marka', 'Ford'), ('model', 'Mustang'), ('rocznik', 1976)])
dict_keys(['marka', 'model', 'rocznik'])
dict_values(['Ford', 'Mustang', 1976])
marka : Ford
model : Mustang
rocznik : 1976
```

Zbiór (set)

Warto podkreślić kilka cech zbiorów:

Zbiór jest kolekcją nieuporządkowaną, czyli nie gwarantuje zachowania żadnej kolejności elementów. Nie posiada indeksów odpowiedzialnych za porządkowanie elementów

Zbiór gwarantuje brak duplikacji, czyli w zbiorze nigdy nie pojawią się dwa elementy o takiej samej wartości. Jeśli dopiszemy element, który istnieje to zostanie on odrzucony przez interpreter języka Python

Zbiory posiadają wiele metod odpowiedzialnych za modyfikację, dodawanie elementów zbioru oraz usuwanie.

Elementów zbioru nie można sortować gdyż nie są indeksowane.
Frozenset - zbiór niemutowalny - nie można do niego dodawać lub z niego usuwać elementów.

Co to zbiory (sets) w języku Python?

Zbiór jest rodzajem kolekcji, która trochę przypomina słownik, ale nie posiada kluczy tylko wartości. Przez ten fakt elementy zbioru są nieuporządkowane. Zbiór służy do przechowywania nieuporządkowanych, ale również unikatowych wartości.

Każdy zbiór podlega modyfikacjom wykonywanym na zbiorze za pomocą funkcji. Funkcje te pozwalają np. dodawać elementy lub usuwać elementy.

Niektóre działania są niedostępne dla zbiorów np. sortowanie.

Operacje na zbiorach

```
drzewa = {"buk", "dąb", "jesion", "baobab", "jabłoń"}  
print(drzewa)
```

Elementy w zbiorze z założenia mają być unikatowe i są nieindeksowane. Wyświetlanie elementów odbywa się w sposób losowy.

Elementy zbioru można wyświetlić za pomocą pętli, ale bez gwarancji kolejności:

```
for d in drzewa:  
    print(d)
```

Operacje na zbiorach

`print("osika" in drzewa)` – sprawdzenie czy element zawiera się w zbiorze

`drzewa.add("osika")` - dodawanie elementu do zbioru

`drzewa.update(["topola", "wierzba", "klon"])` – dodawanie do zbioru elementów zawartych w liście

`drzewa.remove("osika")` – usuwanie elementu ze zbioru, w sytuacji gdy nie istnieje element interpreter wywołuje `runtime error`
`drzewa.discard("jojoba")` - – usuwanie elementu ze zbioru, w sytuacji gdy nie istnieje element interpreter nie reaguje na to

Zbiory można łączyć za pomocą metody `union`

Przykład operacje na zbiorach

▶ 0.0s

```
1 drzewa = {"buk", "dąb", "jesion", "baobab", "jabłłoń"}
2 print(drzewa)
3 for d in drzewa:
4     print(d)
5 print("osika" in drzewa)
6 drzewa.add("osika")
7 drzewa.update(["topoła", "wierzba", "klon"])
8 drzewa.remove("osika")
9 drzewa.discard("jojoba")
```

```
{'jesion', 'dąb', 'buk', 'jabłłoń', 'baobab'}
jesion
dąb
buk
jabłłoń
baobab
False
```

Programowanie funkcyjne paradygmaty

Programowanie funkcyjne jest oparte na ideach matematyki i wywodzi się z kilku uniwersalnych rachunków związanych z analizą matematyczną oraz algebrą. Pierwszą definicją programowania funkcyjnego był rachunek lambda. Programowanie funkcyjne wprowadziło bardzo ważne zmiany w paradygmacie programowania proceduralnego zmieniając podejście do stanu wszelkich parametrów programu

Programowanie wykorzystuje struktury zwane funkcjami, które dla zasady nie opisują stanu globalnego i nie wpływają na niego. Funkcje posiadają parametry i to one opisują stan w zakresie działania funkcji. Każda funkcja powinna zwracać wynik w dowolne miejsce (program, plik py czy też inna funkcja). Generalnie stan globalny realizowany jest przez programowanie obiektowe, a stan lokalny przez funkcyjne.

Programowanie funkcyjne

Programowanie funkcyjne zostało tak zaprojektowane żeby nie posiadać jakiejkolwiek zależności od danych nie reprezentowanych przez samą funkcję. Funkcje nie przeprowadzają żadnych ,modyfikacji na danych spoza swoich parametrów.

Tak jak wspomnieliśmy, programowanie funkcyjne nie wpływa na stan globalny i przez to nie generuje błędów związanych z rozwiązywaniem zmian związanych z elementami globalnymi. Nie zawsze programowanie funkcyjne jest najlepszym sposobem na rozwiązanie problemu. Czasami problem potrzebuje rozwiązania stanu globalnego gdyż grupa obiektów na których bazuje rozwiązanie jest połączona stanem globalnym. Przykładowo może to być API konkretnej usługi(biblioteki związane z usługą) lub GUI aplikacji (Graficzny Interfejs Użytkownika).

Funkcja

Funkcja to fragment kodu źródłowego opisanego przez instrukcję def (definition).
Funkcja posiada nazwę zadawaną zgodnie z poniższym wzorcem:

```
def nazwa(ewentualne parametry):  
    <ciało funkcji>
```

Funkcja może nie posiadać żadnych parametrów, albo może posiadać ich nieoznaczoną ilość. Jeśli funkcja nie posiada parametrów, to piszemy tylko pusty nawias po nazwie funkcji. Parametry funkcji są takimi wartościami od których zależy wynik zwracany przez funkcję. Powinny one być z jednej strony wystarczające do opisanie wyniku, a z drugiej strony nie powinno być żadnej nadmiarowości.

Funkcja

```
def nazwa(a,b,c = 0):  
    return a+b+c
```

```
def info():  
    print("najładniejsze miasto: ", miasto3)
```

```
def zamki(id, *zamki):  
    print("największy zamek:", zamki[2])  
    print(id)  
  
zamki(1, "Malbork", "Ogrodzieniec", "Janowiec")
```

```
x = lambda a:a+10;  
print(x(5))  
print(x(11))
```

Instrukcje - instrukcja warunkowa

Python 3.10

```
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")
```

Python 3.x

```
http_code = "418"
if http_code == „200”:
    print("OK")
    do_something_good()
elif http_code == "404":
    print("Not Found")
    do_something_bad()
elif http_code == "418":
    print("I'm a teapot")
    make_coffee()
else:
    print("Code not found")
```

Programowanie obiektowe

Programowanie obiektowe – paradygmat programowania, w którym programy definiuje się za pomocą obiektów – elementów łączących stan i zachowanie.

Obiektowy program komputerowy wyrażony jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.

Paradygmaty programowania obiektowego



Paradygmaty programowania obiektowego

Abstrakcja: każdy obiekt w systemie służy jako model abstrakcyjnego "wykonawcy", który może wykonywać pracę, opisywać i zmieniać swój stan, oraz komunikować się z innymi obiektami w systemie, bez ujawniania, w jaki sposób zaimplementowano dane cechy.

Hermetyzacja(separacja): oddzielenie „co” od „jak”. Enkapsulacja zapewnia, że obiekt nie może zmieniać stanu innych obiektów w nieokreślony sposób. Każdy typ obiektu dostarcza interfejsu, który określa sposób współpracy z innymi obiektami. Jedynie za pomocą określonych metod mamy możliwość zmienić stan obiektu, bezpośredni dostęp do zmiennych jest zabroniony.

Paradygmaty programowania obiektowego

Dziedziczenie(kompozycja): umożliwia stworzenie hierarchii obiektów w programie. Polega na przejęciu właściwości i funkcjonalności obiektów innej klasy i ewentualnej modyfikacji tych właściwości i funkcjonalności w taki sposób, by były one bardziej wyspecjalizowane

Polimorfizm(wielopostaciowość): referencje i wskaźniki obiektów mogą dotyczyć obiektów różnego typu, a wywołanie metody dla referencji spowoduje zachowanie odpowiednie dla pełnego typu obiektu wywoływanego. Zazwyczaj można wyróżnić dwa rodzaje polimorfizmu: dynamiczne- wykonywane podczas działania programu, a także statyczne- na etapie kompilacji

Tworzenie klas

```
class KlasaPierwsza:  
  
    def __init__(self, a, b):  
        self.a=a  
        self.b=b  
  
    def print_ab(self):  
        print("a = {}, b = {}".format(self.a, self.b))
```

dziedziczenie

```
class KlasaDruga(KlasaPierwsza):  
  
    def __init__(self, a, b, c):  
        super().__init__(a, b)  
        self.c = c  
  
    def print_abc(self):  
        print("a = {}, b = {}, c = {}".format(self.a, self.b, self.c))  
  
    def policz_sume(self):  
        print("suma a+b+c wynosi {}".format(self.a + self.b + self.c))
```

wielodziedziczenie

```
class KlasaTrzecia(KlasaDruga,Ekstra):  
  
    def __init__(self, a, b, c, g, d):  
        KlasaDruga.__init__(self,a, b, c)  
        Ekstra.__init__(self, g)  
        self.d = d  
  
    def print_abcd(self):  
        print("a = {}, b = {}, c = {}, d = {} , ekstra wartość g = {}".  
              format(self.a, self.b, self.c, self.d, self.g))  
  
    def policz_sume(self):  
        print("suma a+b+c+d+g wynosi {}".  
              format(self.a + self.b + self.c + self.d + self.g))
```

Klasa abstrakcyjna

```
from abc import ABC, abstractmethod

class Pojazd(ABC):

    @abstractmethod
    def pokaz_naped(self):
        pass

    @abstractmethod
    def predkosc_max(self):
        pass
```

```
class Rower(Pojazd):

    def pokaz_naped(self):
        return "korba"

    def predkosc_max(self):
        return 60
```

Obsługa błędu - wyjątki

```
def dziel(x,y):  
    try:  
        wynik = x/y  
    except ZeroDivisionError:  
        print("dzielenie przez 0!")  
    except NameError:  
        print("brak rozwiązania")  
    else:  
        print("wynik dzielenia: {}".format(wynik))  
    finally:  
        print("jeśli chcesz dzielić nadal, podaj kolejne wartości.")  
  
try:  
    dziel(1,1)  
except:  
    print("nieoczekiwany błąd!")
```

Dostęp do plików i OS

```
f = open("message.txt", "a")  
f.write("to jest ważna informacja \n")  
f.close()
```

```
f = open("message.txt", "r")  
print(f.read())  
f.close()
```

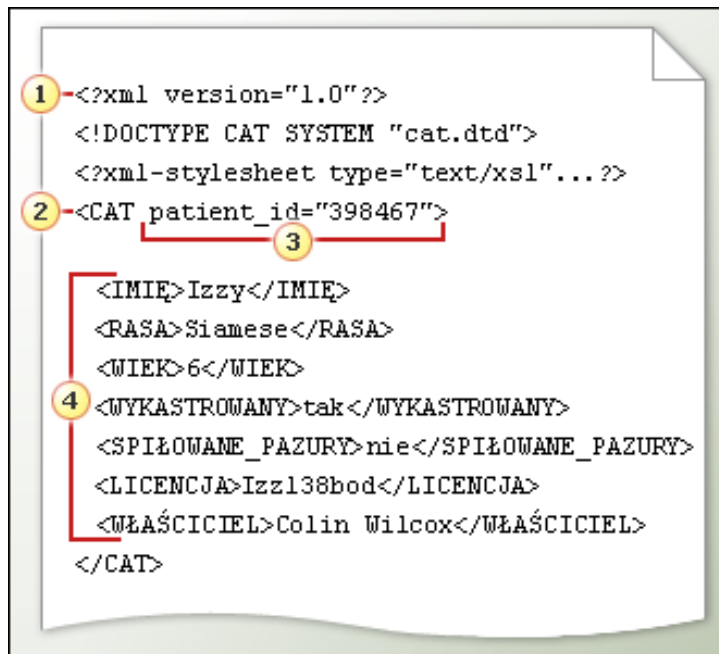
```
if os.path.exists("message.txt"):  
    os.remove("message.txt")  
    print("Plik został usunięty")  
else:  
    print("Plik nie istnieje")
```


POWODY STWORZENIA JĘZYKA XML:

[ZŁE CECHY JĘZYKA HTML]

- Dokument nie zawierający typowych składników (nagłówków, akapitów, list tabel itp.). [Trochę zmieniło się to w HYML5]. HTML-owi brakuje na przykład elementów potrzebnych do oznaczenia notacji muzycznej lub równań matematycznych.
- Baza danych np. katalog książek. Strony HTML możesz używać do przechowywania i wyświetlania informacji ze statycznej bazy danych (na przykład z listy z opisami książek). Jeśli jednak chciałbyś posortować, przefiltrować, odnaleźć lub opracować informacje inaczej, to każda informacja musiałaby zostać oznaczona.
- Dokument, który chcesz uporządkować w hierarchiczną strukturę danych drzewa – powiedzmy, że piszemy książkę – chcemy pooznaczać jej strukturę: rozdziały, podrozdziały. HTML nie daje takich możliwości – nie wprowadza fizycznej struktury.

Budowa dokumentu XML



1. Deklaracje - to znajdujące się w pliku danych informacje identyfikujące sam plik i inne powiązane z nim pliki.
2. Element główny - tag zawierający wszystkie inne tagi i dane w każdym pojedynczym rekordzie.
3. Atrybut - zawierają informacje o tagach i znajdujących się w nich danych. Typy informacji w atrybutach są zdefiniowane przez system XML dla każdego rodzaju tagu.
4. Tagi i dane – podstawowy składni pliku XML

Podstawowymi składnikami dokumentu XML są:

- elementy,
- atrybuty, które są umieszczane w elementach, jako dodatkowe informacje.

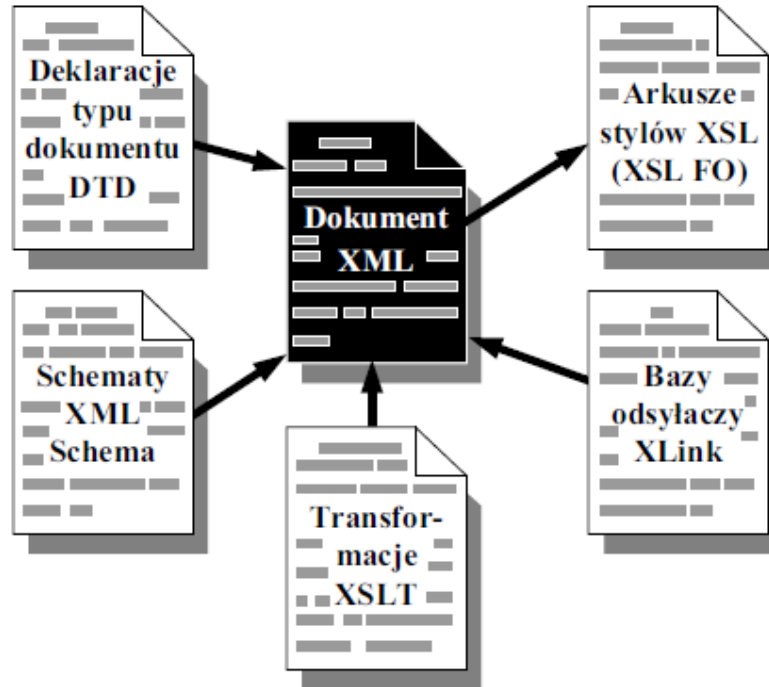
Elementy mogą być przy tym:

- nie puste — posiadające treść,
- puste — bez treści.

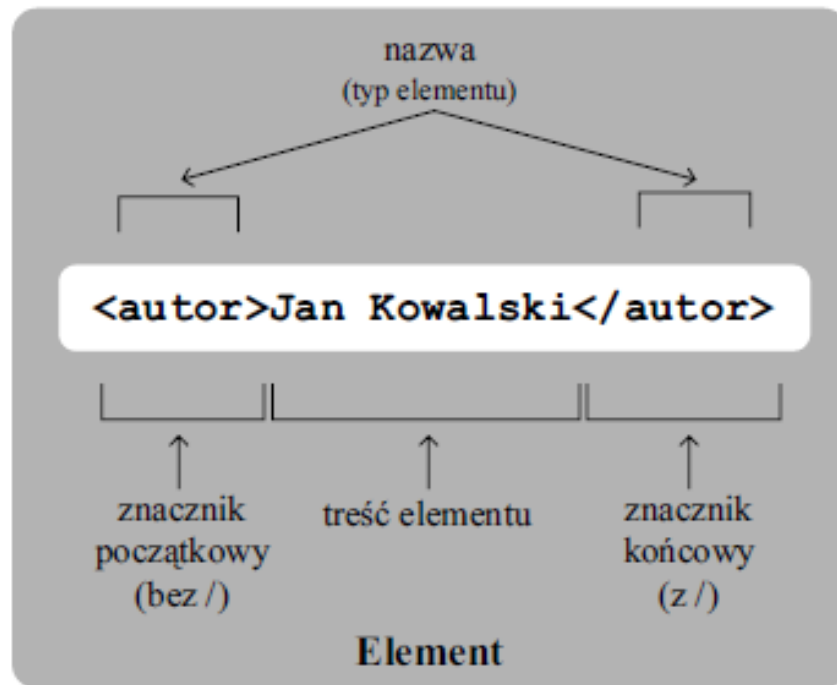
Oprócz tego w dokumencie XML można umieszczać:

- deklaracje,
- instrukcje przetwarzania,
- jednostki.

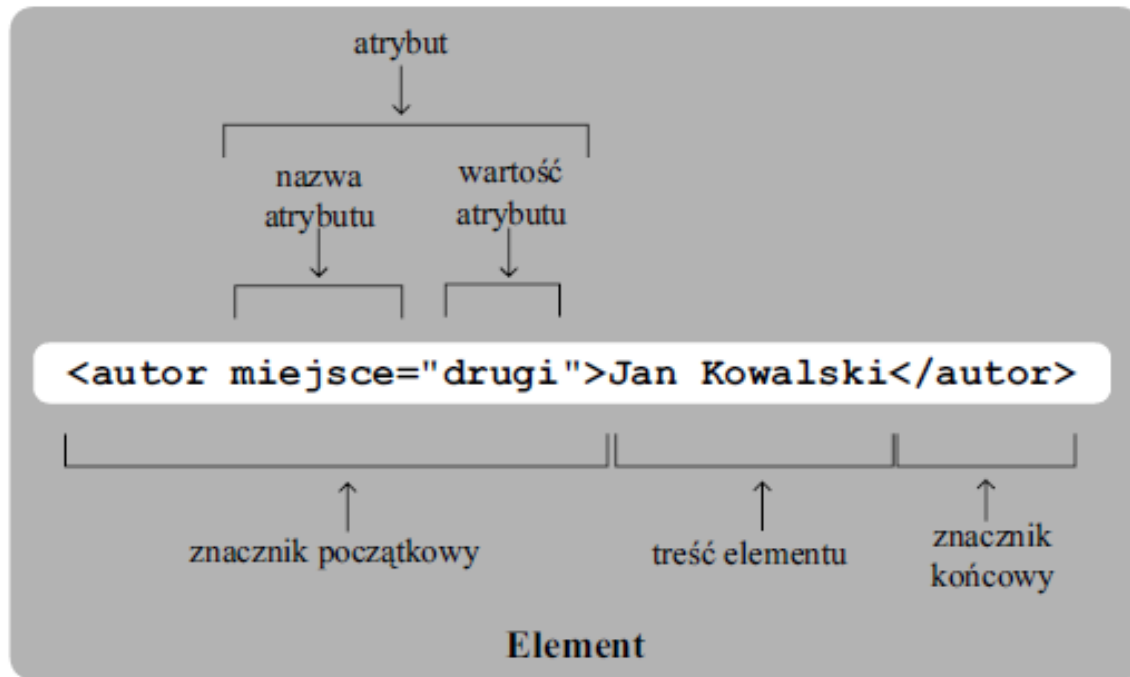
Dokument XML



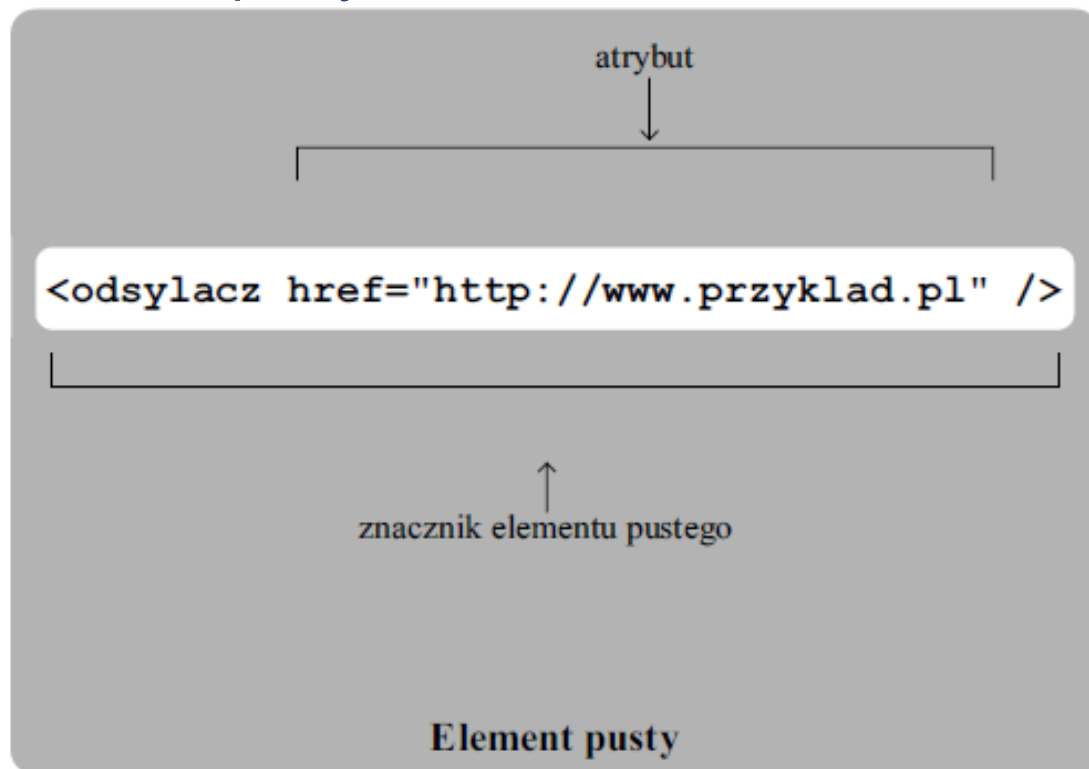
Składowe elementu



Atrybut w elemencie



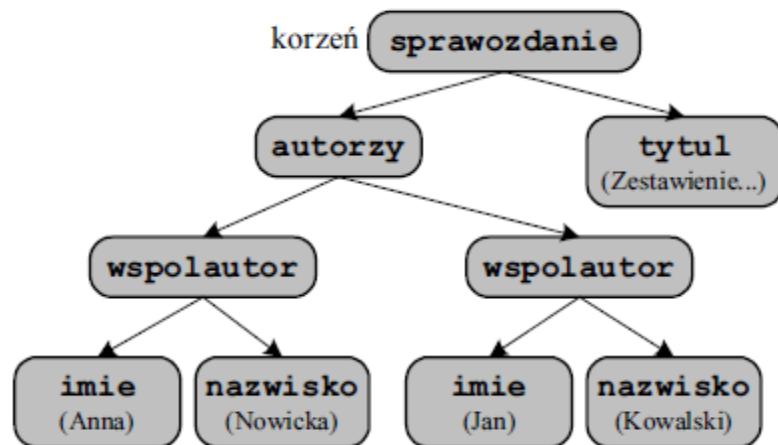
Element pusty



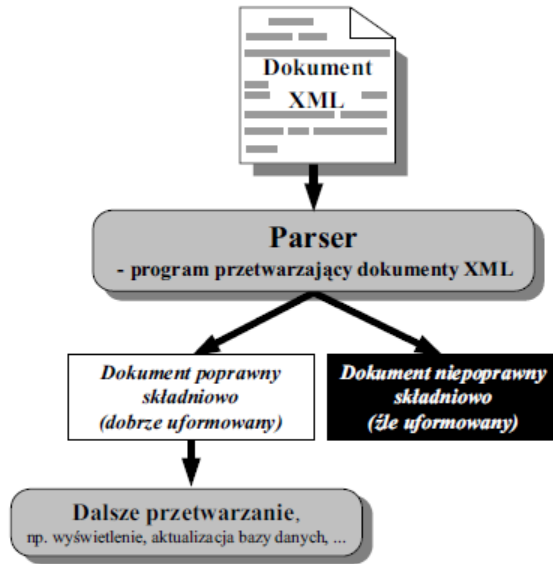
Hierarchia elementów w dokumencie XML

```
<sprawozdanie>
  <autorzy liczba_autorow="2">
    <wspolautor>
      <imie> Anna </imie>
      <nazwisko> Nowicka </nazwisko>
    </wspolautor>
    <wspolautor>
      <imie> Jan </imie>
      <nazwisko> Kowalski </nazwisko>
    </wspolautor>
  </autorzy>

  <tytul> Zestawienie zysków / obrotów </tytul>
</sprawozdanie>
```



Przetwarzanie dokumentów XML



Dokumenty XML mogą być czytane przez użytkowników w zwykłych edytorach tekstowych, jednak najczęściej są one przetwarzane przez programy, zwane parserami.

Programy te powinny w pierwszej kolejności sprawdzić poprawność składniowa dokumentu, tzn. stwierdzić czy jest on dobrze uformowany

Moduły pythona dla obsługi xml

1. `xml.etree.ElementTree`: the ElementTree API, prosty i lekki procesor XML
2. `xml.dom`: definicja DOM API
3. `xml.dom.minidom`: minimalna implementacja DOM
4. `xml.dom.pulldom`: wsparcie dla budowy częściowego drzewa DOM
5. `xml.sax`: podstawowe klasy i funkcje SAX2
6. `xml.parsers.expat`: parser Expat