WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Programowanie w języku Python
Wykład w ramach przedmiotu „Informatyka” (EE-DI)

Dawid Warchoł

Literatura

Do przygotowania tego wykładu zostały wykorzystane głównie poniższe źródła:

1. Dokumentacja języka Python – *https://docs.python.org/pl/3/tutorial*
2. Kurs W3Schools – *https://www.w3schools.com/python*

Pomocne okazały się również trzy pozycje literaturowe:

1. Rob Miles, „Python – Zacznij Programować”, Helion, 2019;
2. Zofia Matusiewicz, „Zacznij od Pythona – Pierwsze Kroki w Programowaniu”, Helion, 2020;
3. Urszula Wiejak, Adrian Wojciechowski, „Python w Zadaniach – Programowanie Dla Młodzieży”, Helion, 2020.

Książki te są polecane przez prowadzącego jako materiały do nauki języka Python, szczególnie pozycja nr. 1.

Program, aplikacja komputerowa, programowanie, język programowania

- Program jest zbiorem instrukcji wykonywanych przez komputer.
- Aplikacja jest programem komputerowym korzystającym z zasobów (np. obrazy, dźwięki) zapewniając użytkownikowi wrażenie pracy w kompletnym środowisku.
- Programowanie – tworzenie programu komputerowego, najczęściej poprzez pisanie kodu w wybranym języku programowania.
- Język programowania – zbiór zasad opisujący jak wykonać określone obliczenia za pomocą wpisywania ciągu znaków.

Rodzaje języków programowania – niskiego i wysokiego poziomu

- Języki programowania niskiego poziomu:
  - niski poziom abstrakcji kodu;
  - mało zrozumiałe dla człowieka;
  - jedna instrukcja najczęściej odpowiada jednemu rozkazowi procesora;
  - zastosowania: programowanie sterowników urządzeń i niskopoziomowych systemów wbudowanych, hakowanie;
  - przykłady: języki asemblera, IL.

- Języki programowania wysokiego poziomu:
  - wysoki poziom abstrakcji kodu;
  - zrozumiałe dla człowieka;
  - jedna instrukcja odpowiada najczęściej wielu rozkazom procesora;
  - zastosowania: programowanie aplikacji, tworzenie zaawansowanych programów obliczeniowych;
  - przykłady: Pascal, Delphi, C, C++, C#, Java, JavaScript, PHP, **Python**, Ruby, Matlab, Rust.

Rodzaje języków programowania – kompilowane i interpretowane

- Języki programowania kompilowane:
  - kod programu jest kompilowany (tłumaczony) na kod maszynowy lub język pośredni;
  - przykłady: C, C++, C#, Rust.

- Języki programowania interpretowane:
  - programy są wykonywane bezpośrednio na podstawie kodu przy użyciu tzw. interpretera;
  - przykłady: JavaScript, PHP, **Python**, Ruby, Matlab.

Podział związany z paradygmatami programowania: języki programowania: strukturalne, obiektowe, funkcyjne, logiczne

- Języki programowania strukturalne:
  - polecenia wykonywane są sekwencyjnie i zmieniają krok po kroku zawartość pamięci, aż do uzyskania wyniku (paradygmat programowania imperatywnego);
  - stosuje się instrukcje warunkowe if-else oraz pętle typu while i for; unika się instrukcji skoku (paradygmat programowania strukturalnego);
  - przykłady: Pascal, C.

- Języki programowania obiektowe:
  - program stanowi zbiór obiektów, które wykazują interakcję miedzy sobą;
  - przykłady: C++, C#, Java, JavaScript, PHP, Ruby, **Python.**

Źródło: *https://wazniak.mimuw.edu.pl/index.php?title=Paradygmaty_programowania/Wykład_15:_Inne_paradygmaty_warte_wspomnienia*

Podział związany z paradygmatami programowania: języki programowania strukturalne, obiektowe, funkcyjne, logiczne

- Języki programowania funkcyjne (funkcjonalne):
  - program rozumiemy jako złożoną funkcję matematyczną, która oblicza pewien wynik na podstawie danych wejściowych;
  - rekurencja zamiast pętli;
  - zastosowania: obliczenia matematyczne, systemy o wysokiej niezawodności (np. finansowe, telekomunikacyjne);
  - przykłady: Scheme, SML, Haskell, Clojure, F#.

- Języki programowania logiczne:
  - program składa się z przesłanek i celu;
  - program próbuje udowodnić cel na podstawie przesłanek;
  - zastosowania: systemy ekspertowe, analiza kodu, edukacja (nauka logiki);
  - przykłady: Prolog, ASP, Datalog.

Źródło: *https://wazniak.mimuw.edu.pl/index.php?title=Paradygmaty_programowania/Wykład_15:_Inne_paradygmaty_warte_wspomnienia*

Jakim językiem programowania jest Python?

- Python jest językiem:
  - wysokiego poziomu;
  - interpretowanym;
  - obiektowym - jednak umożliwia również programowanie strukturalne, którym będziemy się zajmować w przeważającej części tego przedmiotu.

Dlaczego uczymy się akurat Pythona na tym przedmiocie?

- Jest prosty pod względem składni, ale bardzo rozbudowany pod względem możliwości.
- Jest jednym z najbardziej popularnych i najczęściej używanych języków programowania (w ostatnich latach wygrywa w rankingach popularności lub zajmuje drugie miejsce, zaraz za językiem JavaScript).
- Posiada wiele zaawansowanych modułów (bibliotek) umożliwiających pisanie programów związanych ze sztuczną inteligencją, multimedialnych, matematycznych, statystycznych, finansowych i innych.
- Umożliwia tworzenie aplikacji desktopowych oraz webowych.

Środowisko programistyczne

Jest to aplikacja, przy użyciu której tworzymy i uruchamiamy własne programy.
W skład środowiska do Pythona wchodzą przede wszystkim:

- edytor – piszemy w nim kod;
- interpreter – interpretuje i uruchamia pisane przez nas programy;
- interaktywna konsola – programy są w niej uruchamiane; możemy w niej też wykonywać pojedyncze instrukcje bez pisania całego programu;
- debugger – narzędzie ułatwiające znajdywanie błędów (bugów) w programie.

Popularne środowiska do programowania w języku Python (Windows, Linux, macOS):

- **Pycharm** (uniwersalne i zalecane dla studentów tego przedmiotu);
- **Spyder** (polecane dla naukowców i inżynierów danych);
- **Visual Studio Code** (edytor kodu dla wielu języków, który można rozbudować, żeby pełnił funkcję środowiska programistycznego Pythona).

OnlineGDB – proste środowisko internetowe, nie wymaga instalowania: *https://www.onlinegdb.com/online_python_interpreter*.

Środowiska programistyczne do Pythona dla urządzeń mobilnych

Jeśli ktoś chciałby testować programy z wykładów na smartfonie/tablecie z systemem Android lub IOS, to najlepiej skorzystać ze środowisk programistycznych w formie aplikacji mobilnych:

- **Pydroid 3** (Android – dostępne za darmo w sklepie Google Play),
- **Pythonista 3** (iOS – dostępne odpłatnie w App Store),
- **Python Coding Editor & IDE App** (iOS – dostępne za darmo w App Store).

Instalacja środowiska Pycharm

- Aby móc korzystać ze środowiska Pycharm należy go pobrać i zainstalować: *https://www.jetbrains.com/pycharm/download*

- Jest to darmowe oprogramowanie. Można jednak wykupić subskrypcję Pro, która zwiększa możliwości Pycharma, głównie w zakresie programowania webowego oraz obsługi baz danych. Istnieje 30-dniowa wersja próbna – trial.

- Oprócz tego należy pobrać i zainstalować interpreter języka Python (najlepiej najnowszą wersję): *https://www.python.org/downloads*
  - Interpreter można też pobrać automatycznie podczas tworzenia projektu w Pycharm (Python version: download and install), jednak mamy tam do wyboru tylko kilka wersji Pythona; może nie być najnowszej wersji lub takiej, którą chcemy.

- Korzystanie ze środowiska Pycharm zostanie omówione po części na wykładzie, ale również na zajęciach laboratoryjnych.

Pierwszy program - sprawdzenie wersji Pythona

- Na tym przedmiocie omawiamy język Python w wersji 3.10 lub wyższej.
- Poniższy program wyświetla informacje o tym, z jakiej wersji aktualnie korzystamy.
- Jeśli mamy wcześniejszą wersję, to powinniśmy zainstalować nowszy interpreter lub zmienić środowisko (w środowiskach internetowych zazwyczaj nie ma możliwości zmiany wersji).

```
import sys
print(sys.version)
```

Drugi program – funkcja do wypisywania tekstu na konsoli

- Funkcja `print` służy do wypisywania tekstu i wartości liczbowych/logicznych na konsoli.

```
print('Drugi program')

print("Drugi program")

print('2')

print('2+3')

print('2' + '3')

print(2+3)

print('dwa plus trzy jest równe:', 2+3)

print(f'dwa plus trzy jest równe: {2+3}.')
```

Komentarze

- Komentarze służą do wstawiania opisów kodu lub do tymczasowego wyłączania linijek kodu z wykonywania (kod „zakomentowany”).

```
print('Drugi program') #teksty (łańcuchy znakowe) muszą być zapisane w apostrofie

print("Drugi program") #albo w cudzysłowie

print('2') #to też jest tekst (składający się z cyfry)

print('2+3') #to również jest tekst - dlatego wyświetli się dosłownie

print('2' + '3') #dodawanie dwóch cyfr w formie tekstu spowoduje ich złączenie

print(2+3) #tutaj zadziała dodawanie liczb, bo zapisaliśmy je bez cudzysłowu

print('dwa plus trzy jest równe:', 2+3)

#tej instrukcji już nie potrzebujemy, ale na razie nie usuwamy jej całkowicie
#print(f'dwa plus trzy jest równe: {2+3}. ')
```

Komentarze wielolinijkowe

```
#print("Drugi program")

print("Drugi program") #tylko ta instrukcja się wykona

'''
print('2')

print('2+3')

print('2' + '3')

print(2+3)

print('dwa plus trzy jest równe:', 2+3)

print(f'dwa plus trzy jest równe: {2+3}.')
'''
```

Operatory arytmetyczne

```
2 + 4   #dodawanie

3 - 1   #odejmowanie

3 * 4   #mnożenie

6 / 2   #dzielenie

7 % 3   #reszta z dzielenia (7 przez 3)

2 ** 3 #potęgowanie (2 do potęgi 3)
```

Operator reszty z dzielenia

- Może się przydać do określenia parzystości liczby lub podzielności jednej liczby przez drugą.

- Co wyświetli program (jaki będzie wynik poniższych operacji)?

```
print(4 % 2)

print(5 % 2)

print(8 % 3)
```

Pierwiastkowanie przy użyciu operatora potęgowania

- W jaki sposób przy pomocy operatora potęgowania obliczyć pierwiastek kwadratowy z 9?

```
9 ** (1/2) #sposób I
9 ** 0.5   #sposób II (lepszy, bo program nie musi wykonywać dzielenia)
9 ** 1/2   #To nie zadziała poprawnie. Dlaczego?
```

- W jaki sposób przy pomocy operatora potęgowania obliczyć pierwiastek sześcienny (3-go stopnia) z 27?

```
27 ** (1/3) #Nawiasy zmieniają domyślną kolejność operacji (jak w matematyce).
```

- Uwaga: Ułamki dziesiętne w języku Python zapisujemy z kropką, nie z przecinkiem! Np. `3.6`, a nie `3,6`.

Czy operatory arytmetyczne działają z tekstami?

- Co wyświetli program?

```
print('Ala' + 'ma' + 'kota')

print('Ala ' + 'ma ' + 'kota')

print('(^_^)' * 10)
```

- Te instrukcje są błędne. Nie wykonają się. Przeczytać opisy błędów i zinterpretować je.

```
print('A' - 'B')

print('A' / 'B')

print('A' + 10)
```

Podstawowe typy danych

```
#str (string) - tekst, napis, łańcuch znakowy
'ABC'
'123'
"tekst"

#int (integer) - liczba całkowita
123
0
-10

#float (floating point) - liczba zmiennoprzecinkowa/zmiennopozycyjna
#Ten typ odnosi się do liczb rzeczywistych.
4.25 #liczby zmiennoprzecinkowe zapisujemy z kropką
7.0
0.0
-5.2
```

- Wartości danego typu wpisane w kodzie programu nazywają się literałami lub stałymi dosłownymi.

Jak możemy zapamiętać wartości w języku programowania?

- Służą do tego zmienne i operator przypisania =.

```
liczba_calkowita = 20
liczba_zmiennoprzecinkowa = 5.2
tekst = 'ABC'

X1 = 1
X2 = X1 + 2 #wyrażenie, którego wartość jest równa 3
print(X1)
print(X2)
```

- Zawsze przypisujemy wartość wyrażenia z prawej strony operatora = do zmiennej z lewej strony. Nigdy na odwrót!
- Zmienne przechowują swoje wartości podczas działania programu w pamięci operacyjnej (RAM).

Jak długo zmienna przechowuje swoją wartość?

- Zazwyczaj zmienne przechowują swoje wartości dopóki trwa program lub funkcja, w której zostały po raz pierwszy użyte.
- Wartość zmiennej może się zmienić, jeśli zostanie nadpisana przez kolejną operację.

```
A = 1
print(A)
A = A + 1
print(A)
```

- Możemy usunąć utworzoną wcześniej zmienną wpisując jej nazwę po słowie `del`.

```
a = 2
del a
print(a) #błąd - zmienna a nie istnieje
```

Pierwsze użycie zmiennej

- Pierwsze użycie zmiennej musi zawsze wiązać się z wpisaniem do niej wartości. Nie można po raz pierwszy użyć zmiennej odczytując z niej wartość (bo taka zmienna wtedy nie istnieje).

```
A = 1
A = B * 2 #błąd (przeczytać jego opis)
print(B)   #błąd
```

```
B = 4
A = B * 2
print(A)
```

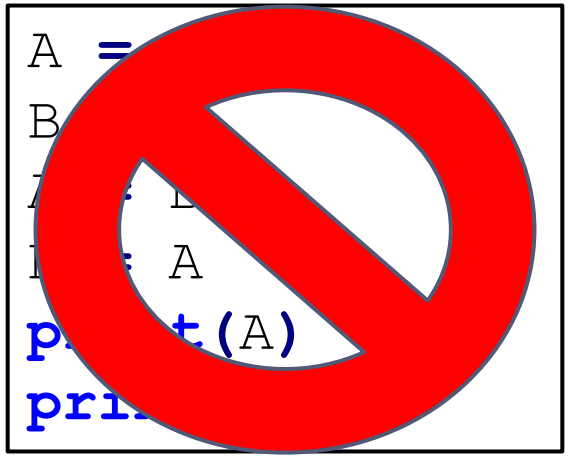
Zadanie 1

- Napisać program, który wpisuje do dwóch zmiennych przykładowe wartości. Następnie program powinien zamienić ze sobą wartości obu zmiennych. Na końcu trzeba je wypisać (dla sprawdzenia).

```
A = 1
B = 2
A = B
B = A
print(A)
print(B)
```

Zadanie 1

- Napisać program, który wpisuje do dwóch zmiennych przykładowe wartości. Następnie program powinien zamienić ze sobą wartości obu zmiennych. Na końcu trzeba je wypisać (dla sprawdzenia).



Zadanie 1

- Napisać program, który wpisuje do dwóch zmiennych przykładowe wartości. Następnie program powinien zamienić ze sobą wartości obu zmiennych. Na końcu trzeba je wypisać (dla sprawdzenia).

```
A = 1
B = 2
pomocnicza = A
A = B
B = pomocnicza
print(A)
print(B)
```

Operatory arytmetyczne połączone z operatorami przypisania

```
a = 0

a += 2   #równoznaczne z: a = a + 2
a -= 1   #równoznaczne z: a = a - 1
a *= 6   #równoznaczne z: a = a * 6
a /= 3   #równoznaczne z: a = a / 3
a %= 3   #równoznaczne z: a = a % 3
a **= 4 #równoznaczne z: a = a ** 4

print(a) #co zostanie wypisane?
```

- **Uwaga dla programistów innych języków:** Python nie posiada operatorów inkrementacji i dekrementacji (++ i --) znanych z języka C i języków wywodzących się z C. A szkoda. ☹

Zasady nazywania zmiennych

- Nazwy zmiennych mogą składać się z jednego lub więcej znaków spośród następujących grup:
  - małe i duże litery angielskiego alfabetu
    - znaki polskie i charakterystyczne dla innych języków są dopuszczalne w najnowszych wersjach Pythona, ale są one niezalecane;
    - najlepiej więc nazwać zmienną `zolw` zamiast `żółw`.
  - cyfry (cyfra nie może być pierwszym znakiem występującym w nazwie)
  - znak podkreślenia _
- Zmienne powinny nazywać się tak, żeby programista czytający kod mógł się domyślić, do czego służą.
- Dobre przykłady: `wysokosc_trojkata`, `szerokoscOkna`, `nazwisko`, `temperatura`.
- Złe przykłady: `zmienna1`, `liczba`, `ajwwgafjkh`.

Wczytywanie danych z konsoli

- Funkcja `input` służy do wczytywania tekstu (np. napisanego klawiaturą) poprzez konsolę.
- Domyślnie `input` wczytuje dane w formie tekstu, ale można je rzutować (przekonwertować) na liczbę całkowitą lub zmiennoprzecinkową funkcjami `int` i `float`.

```
print('Podaj nazwę produktu: ')
nazwa = input()

print('Podaj liczbę sztuk: ')
ile_sztuk = int(input())

print('Podaj cenę: ')
cena = float(input())
```

Wczytywanie danych z konsoli

- Podczas wywołania `input` można w nawiasie wpisać tekst, który wyświetli się na konsoli w momencie, gdy funkcja oczekuje na wpisanie danych przez użytkownika.
- Nie trzeba w takim wypadku wywoływać wcześniej funkcji `print`.

```
nazwa = input('Podaj nazwę produktu: ')

ile_sztuk = int(input('Podaj liczbę sztuk: '))

cena = float(input('Podaj cenę: '))
```

Dwa podstawowe rodzaje błędów w programowaniu

1. Błędy składniowe (syntaktyczne). Są to błędy, które wynikają z tego, że kod naszego programu jest niezrozumiały dla interpretera. Przykłady:

```
a = a+*3
3 = a
print(a
```

2. Błędy znaczeniowe (semantyczne). Program jest zrozumiały dla interpretera, ale nie robi tego, co od niego oczekujemy. Przykłady:
   - Chcąc przypisać wartość 0 do zmiennej x programista napisał: `x==0`
   - Chcąc wyświetlić wartość zmiennej `x` programista napisał `print('x')`

Jeśli mamy w kodzie błąd składniowy, interpreter na pewno nas o tym poinformuje (czerwonym komunikatem). Błędy znaczeniowe zazwyczaj musimy sami odnaleźć. Może nam w tym jednak pomóc tzw. debugger.

Funkcje do rzutowania typów

- Funkcje `int` i `float` możemy użyć nie tylko przy wywołaniu funkcji `input` ale również w dowolnym miejscu programu, aby rzutować typ zmiennej lub literału.

```
A = '10'
B = A * 2
B_int = int(A) * 2
print(B)
print(B_int)
```

```
A = '2.5'
B = A * 2
B_float = float(A) * 2
print(B)
print(B_float)
```

Funkcje do rzutowania typów

- Można również rzutować na typ str (tekst). Służy do tego funkcja `str`.

```
liczba = 123
tekst = str(liczba) * 2
liczba = liczba * 2

print(tekst)
print(liczba)
```

Funkcja do sprawdzania typu danych

- Aby sprawdzić typ zmiennej możemy użyć funkcję `type`. Funkcja jedynie zwraca typ danych. Aby go wyświetlić należy dodatkowo użyć `print`.

```
A = '123'
B = "123"
C = 123
D = 123.0

print(type(A))
print(type(B))
print(type(C))
print(type(D))
```

Typ operacji dzielenia i operator dzielenia całkowitoliczbowego

- Wynikiem operacji dzielenia przy użyciu operatora / jest zawsze wartość typu float (nawet jeśli dzielenie jest bez reszty).
- Jeśli chcemy wykonać dzielenie całkowitoliczbowe, czyli takie, którego wynikiem jest wartość typu int (z obcięciem ewentualnej części ułamkowej liczby), to należy skorzystać z operatora //.
- Uwaga: Inaczej było w wersjach Pythona wcześniejszych niż 3.

```
print(5/2)
print(5//2)

print(type(6/3))
print(type(6//3))
```

Funkcja do zaokrąglania liczb (`round`)

- Funkcja `round` służy do zaokrąglania liczb.
- Zaokrąglanie, to nie jest to samo, co obcięcie części ułamkowej (znane z dzielenia całkowitoliczbowego).
- Pierwszy parametr funkcji `round`, to liczba, którą chcemy zaokrąglić, a drugi, to docelowa liczba miejsc po przecinku.
- Jeśli nie wpiszemy drugiego parametru, to liczba zaokrągli się do najbliższej liczby całkowitej (i będzie typu int).

```
print(round(2.2))
print(round(2.9))
print(round(2.117, 2))
```

Wartość `None`

- Wartość `None` przypisana do zmiennej oznacza, że zmienna ta nie ma żadnej konkretnej wartości.

- Zmienna z wartością `None` nie ma również konkretnego typu (typ `NoneType`).

- Wartość `None` można stosować np. w następujących przypadkach:
  - gdy nie chcemy całkowicie usuwać zmiennej, ale chcemy wymazać jej wartość oraz typ;
  - do określenia sytuacji, w której użytkownik nie zdecydował się podać żadnej wartości dla zmiennej (0 lub pusty łańcuch znakowy byłyby konkretną wartością konkretnego typu).

```
x = None
print(x)
print(type(x))
```

Znaki specjalne tekstu

- `\n` – znak przejścia do nowej linii
- `\t` – znak tabulacji
- `\'` – znak apostrofu
- `\"` – znak cudzysłowu

```
print('pies\tkot\nwróbel\tgołąb')
print('\'pies\'')
print('c:\\dokumenty')
```

- Należy pamiętać, że funkcja `print` automatycznie wstawia znak przejścia do nowej linii na końcu wyświetlanego tekstu. Nie musimy więc go wpisywać.

Operatory porównania

- Operatory porównania służą do sprawdzania relacji między dwiema wartościami.

```
1 >  2 #1 większe od 2
1 >= 2 #1 większe lub równe 2
1 <  2 #1 mniejsze od 2
1 <= 2 #1 mniejsze lub równe 2
1 == 2 #1 równe 2
1 != 2 #1 różne od 2
```

- Wynikiem operacji porównania jest wartość logiczna typu bool: `True` lub `False` (pisana zawsze z dużej litery). Można to sprawdzić w następujący sposób:

```
print(1>2)
print(type(1>2))
```

Operatory porównania

- Możemy je stosować dla wartości typu int, float, bool, a nawet str.
- Dla typu str porównywane są kody w standardzie Unikod (Unicode, https://www.ssec.wisc.edu/~tomw/java/unicode.html) z priorytetem od pierwszego znaku, do ostatniego.

```
print('Marian' == 'Mariusz')
print('Marian' < 'Mariusz')
```

- Można dzięki temu sprawdzić, czy dwa napisy są identyczne (==) lub nieidentyczne (! =).
- Można też sprawdzić, który napis jest „bliżej” lub „dalej” w kolejności alfabetycznej za pomocą operatorów >, <, >=, <= . Trzeba jednak pamiętać, że:
  - duże litery [A-Z] mają mniejsze kody niż małe litery [a-z];
  - polskie znaki (np. „Ą”, „Ć”) mają kody większe niż znaki z podstawowego zestawu [A-Z], [a-z].

Funkcja rzutowania bool

- Podobnie jak funkcje `int`, `float`, `str`, funkcja `bool` służy do rzutowania (konwertowania) wartości na typ logiczny bool zgodnie z następującymi zasadami:
  - int – liczba 0 jest zamieniana na wartość `False`; każda inna liczba – na `True`.
  - float – liczba 0.0 jest zamieniana na wartość `False`; każda inna liczba – na `True`.
  - str – pusty tekst '' jest zamieniany na `False`; każdy inny tekst – na `True`.

```
print(bool(0))
print(bool(-4))
print(bool(0.0))
print(bool(6.5))
print(bool(''))
print(bool('abc'))
```

Operatory logiczne

- Operatory logiczne umożliwiają wykonanie wielu porównań wartości zgodnie z zasadami logiki:
  - `and` (iloczyn logiczny, koniunkcja) – jeśli obie wartości są równe `True`, wynikiem operacji jest `True`; w przeciwnym wypadku wynikiem jest `False`.
  - `or` (suma logiczna, alternatywa) – jeśli przynajmniej jedna wartość jest równa `True`, wynikiem operacji jest `True`; w przeciwnym wypadku wynikiem jest `False`.
  - `not` (negacja) – zamienia wartość `False` na `True` lub wartość `True` na `False`.

```
print(3 > 2 and 1 == 1) #True
print(3 > 2 and 1 == 0) #False
print(3 > 2 or  1 == 1) #True
print(3 > 2 or  1 == 0) #True
print(3 < 2 or  1 == 0) #False
print(10 != 0 or 3 > 2 and 1 == 0) #Jaki będzie wynik?
#Podpowiedź: operator and ma wyższy priorytet niż or.
print(not 1 == 0) #równoważne z 1 != 0
```

Priorytet operatorów

- Określa kolejność działań z nimi związanych.
- Im niżej występuje operator w tabeli, tym wyższy ma priorytet (wcześniej wykona się jego działanie).
- Zwrócić uwagę na priorytet operatorów *, /, %, +, – oraz `and`, `or`.
- Jeśli operatory mają ten sam priorytet, to operacje wykonują się od lewej do prawej. Wyjątkiem jest operator potęgowania. W jego przypadku operacje wykonują się od prawej do lewej.

  Np. 2**3**2 jest równoważne $2^{(3^2)}$, a nie $(2^3)^2$.

```
print(2**3**2) #sprawdzić wynik
```

Operator	Opis
`lambda`	Wyrażenie lambda
`or`	Logiczne OR (lub)
`and`	Logiczne AND (i)
`not` *x*	Logiczne NOT (nie)
`in`, `not in`	Testy przynależności
`is`, `is not`	Testy tożsamości
`<`, `<=`, `>`, `>=`, `!=`, `==`	Porównania
`|`	Bitowe OR (lub)
`^`	Bitowe XOR (różnica symetryczna)
`&`	Bitowe AND (i)
`<<`, `>>`	Przesunięcia
`+`, `-`	Dodawanie i odejmowanie
`*`, `/`, `%`	Mnożenie, dzielenie, reszta z dzielenia
+*x*, -*x*	Identyczność, negacja
~*x*	Bitowe NOT (nie)
`**`	Potęgowanie
x`.`*atrybut*	Odwołanie do atrybutu
x`[`*indeks*`]`	Odwołanie do indeksu
x`[`*indeks*`:`*indeks*`]`	Wykrojenie
f`(`*argumenty*`...)`	Wywołanie funkcji
`(`*wyrażenia*`...)`	Powiązanie lub drukowalna forma krotki
`[`*wyrażenia*`...]`	Drukowalna forma listy
`{`*klucz*`:`*dana*`...}`	Drukowalna forma słownika
`` ``*wyrażenia*`...` ``	Konwersja napisowa

Jaki będzie wynik poniższej operacji?

- Co się wyświetli? Sprawdź i wyjaśnij na podstawie tabeli priorytetów operatorów.

```
print(-3**2)
```

Odpowiedź:

- W pierwszym przypadku wynikiem będzie -8 (bo operator − ma niższy priorytet od operatora **). Jeśli chcemy, żeby zadziałało to zgodnie z zasadami matematyki, to musimy napisać:

```
print((-3)**2)
```

Operator	Opis
`lambda`	Wyrażenie lambda
`or`	Logiczne OR (lub)
`and`	Logiczne AND (i)
`not` *x*	Logiczne NOT (nie)
`in`, `not in`	Testy przynależności
`is`, `is not`	Testy tożsamości
`<`, `<=`, `>`, `>=`, `!=`, `==`	Porównania
`|`	Bitowe OR (lub)
`^`	Bitowe XOR (różnica symetryczna)
`&`	Bitowe AND (i)
`<<`, `>>`	Przesunięcia
`+`, `-`	Dodawanie i odejmowanie
`*`, `/`, `%`	Mnożenie, dzielenie, reszta z dzielenia
+*x*, -*x*	Identyczność, negacja
~*x*	Bitowe NOT (nie)
`**`	Potęgowanie
x`.`*atrybut*	Odwołanie do atrybutu
x`[`*indeks*`]`	Odwołanie do indeksu
x`[`*indeks*`:`*indeks*`]`	Wykrojenie
f`(`*argumenty*`...)`	Wywołanie funkcji
`(`*wyrażenia*`...)`	Powiązanie lub drukowalna forma krotki
`[`*wyrażenia*`...]`	Drukowalna forma listy
`{`*klucz*`:`*dana*`...}`	Drukowalna forma słownika
`` ``*wyrażenia*`...`` ``	Konwersja napisowa

Operator（）do zmiany kolejności wykonywania operacji

- Operator（）(nawias okrągły) może zmienić kolejność wykonywania operacji.
- Operacje zawarte w nawiasie wykonują się w pierwszej kolejności (podobnie jak w matematyce).

```
print(1+2*3)    #wynik: 7
print((1+2)*3)  #wynik: 9
```

Instrukcja warunkowa `if`

- Umożliwia wykonanie operacji tylko wtedy, gdy spełniony jest warunek, tzn. gdy wartość logiczna warunku jest równa `True`.
- Po słowie `if` należy wpisać warunek, a potem dwukropek.
- Instrukcje, które mają się wykonać warunkowo wpisujemy w nowej linii zaczynając od wcięcia.
- Wcięciem jest najczęściej znak tabulacji, choć dopuszczalna jest spacja.

```
a = int(input('Podaj liczbę: '))
if a > 0:
    a = a * 2
    print(a)
```

Uwaga dla programistów innych języków - wcięcia w języku Python

- W większości języków programowania wcięcia wstawiane w instrukcjach pisanych pod warunkami (oraz w innych blokach kodu, np. w pętli, funkcji, klasie) są bardzo zalecane ze względu na zwiększenie czytelności kodu, ale nie są obowiązkowe.

- W języku Python **wcięcia są obowiązkowe!** Bez nich program nie zadziała.

Instrukcja warunkowa `if`

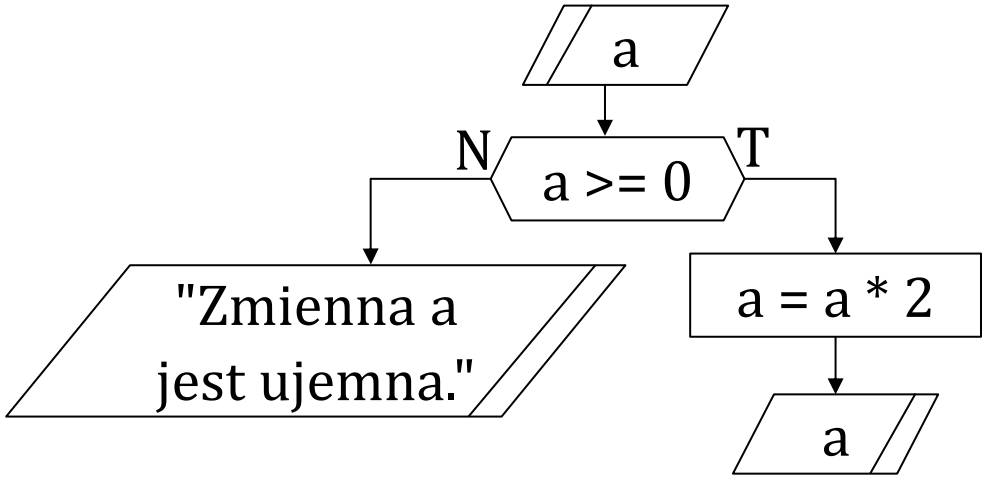
- Jeśli chcemy wykonać warunkowo tylko jedną instrukcję, dopuszcza się wpisanie jej w tej samej linii, co warunek.

```
a = int(input('Podaj liczbę: '))
if a > 10: print(a)
```

Instrukcja warunkowa `if-else`

- Instrukcje po słowie `else` wykonują się tylko wtedy, gdy warunek `if` nie jest spełniony.
- Słowo `else` zapisujemy po warunku `if` i następujących po nim instrukcjach.
- Słowo `else` zapisujemy bez wcięcia, tzn. musi być wyrównane do `if`, a nie do instrukcji następujących po nim.

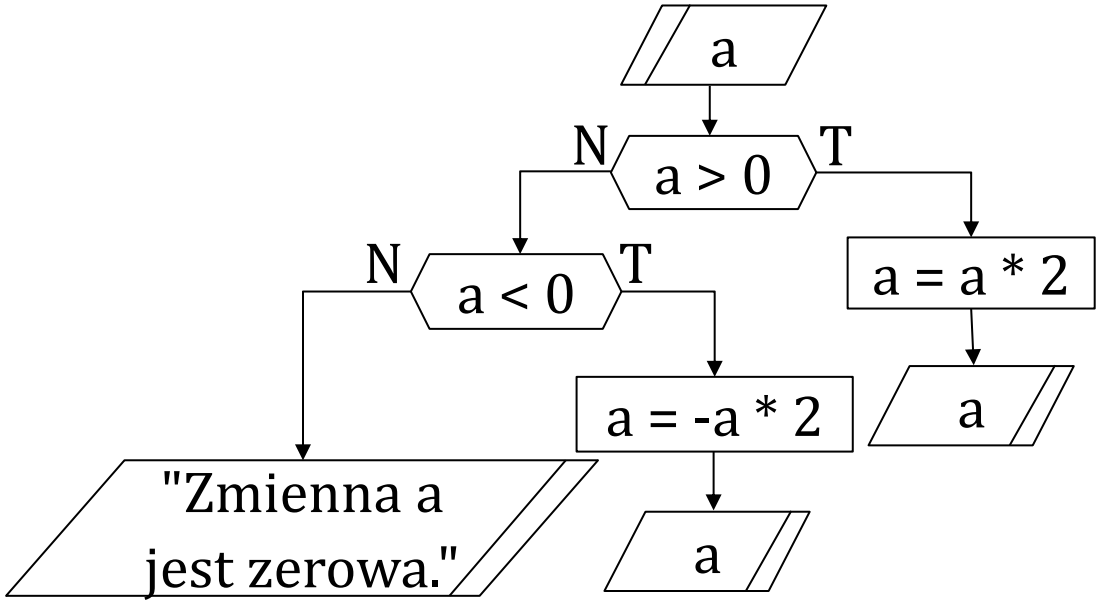
```
a = int(input('Podaj liczbę: '))
if a >= 0:
    a = a * 2
    print(a)
else:
    print('Zmienna a jest ujemna.')
```



Instrukcja warunkowa `if-elif-else`

- Jeśli w przypadku niespełnienia warunku podanego w `if` chcemy sprawdzić kolejny, należy go wpisać po słowie `elif`.

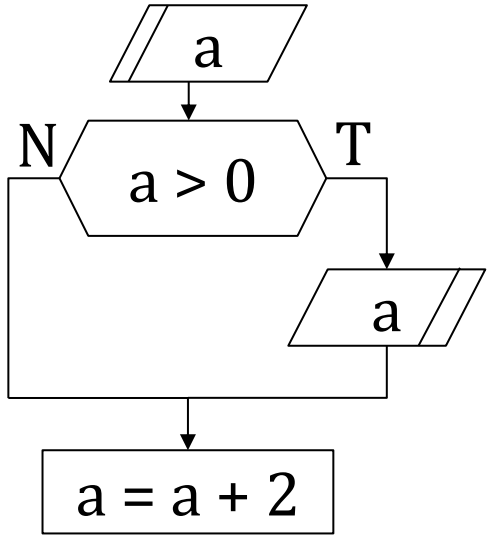
```
a = int(input('Podaj liczbę: '))
if a > 0:
    a = a * 2
    print(a)
elif a < 0:
    a = -a * 2
    print(a)
else:
    print('Zmienna a jest zerowa.')
```

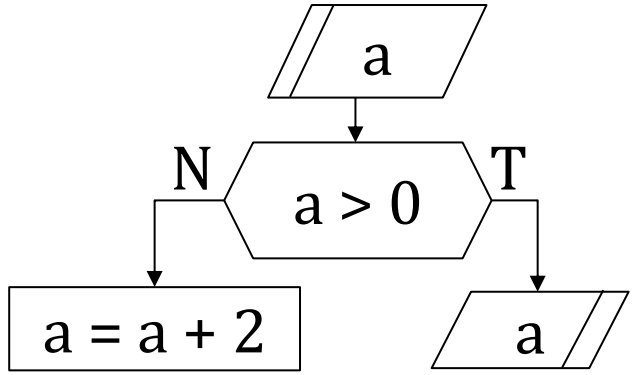


Czym różnią się te dwa programy

```
a = int(input('Podaj liczbę: '))
if a > 0:
    print(a)
a = a + 2
```

```
a = int(input('Podaj liczbę: '))
if a > 0:
    print(a)
else:
    a = a + 2
```





Zadanie 2

- Napisać program wczytujący dwie liczby całkowite, sprawdzający i wypisujący informację, czy pierwsza jest podzielna przez drugą lub druga przez pierwszą.

- Program powinien wypisać „TAK” lub „NIE” w zależności od tego, czy warunek jest spełniony.

Funkcja `exit`

- Powoduje natychmiastowe zakończenie działania programu, bez wykonywania pozostałych instrukcji.

```
a = int(input('Podaj liczbę nieujemną: '))
if a < 0:
    print('Podałeś niepoprawną liczbę.')
    exit()

#te instrukcje się nie wykonają, jeśli podaliśmy liczbę ujemną
pierwiastek = a**0.5
print(f'Pierwiastek kwadratowy z liczby {a} = {pierwiastek}.')
```

Funkcja `exit`

- Funkcja `exit` nie jest jednak zalecana. Można ją stosować tymczasowo, ale w końcowej wersji programu lepiej użyć `if-else`:

```
a = int(input('Podaj liczbę nieujemną: '))
if a < 0:
    print('Podałeś niepoprawną liczbę.')
else:
    #te instrukcje się nie wykonają, jeśli podaliśmy liczbę ujemną
    pierwiastek = a**0.5
    print(f'Pierwiastek kwadratowy z liczby {a} = {pierwiastek}.')
```

Pierwiastek z liczby ujemnej

- Co by się stało, gdybyśmy jednak próbowali obliczyć pierwiastek kwadratowy z liczby ujemnej? Czy program zakończyłby się z błędem?

```
a = -4
b = a**0.5
print(b)
```

- Program nie zakończy się z błędem i obliczy wartość zmiennej `b` w dziedzinie liczb zespolonych.

Typ complex – liczby zespolone

- Typ complex pozwala przechowywać i przetwarzać liczby zespolone.
- Część urojona liczby zespolonej oznaczana jest przez literę **j**. Jest to typowy zapis inżynierski, stosowany np. w elektrotechnice do reprezentacji prądu przemiennego w formie wektorów (fazorów).
- W typowym zapisie matematycznym część urojoną oznacza się literą **i**.

```
a = 3 + 2j
b = 4 + 3j
print(a+b)
print(type(a))
```

Separator instrukcji - średnik

- W Pythonie istnieje możliwość wpisywania wielu instrukcji w jednej linijce.

- Musimy jednak pamiętać, aby pomiędzy instrukcjami wstawić średnik (separator).

```
a=1 b=2 print(a+b)
```

błąd

```
a=1; b=2; print(a+b)
```

OK

- Nie należy jednak nadużywać pisania wielu instrukcji w jednej linijce, bo zmniejsza to czytelność kodu.

Zadanie 3

- Napisać program wyznaczający miejsca zerowe (pierwiastki rzeczywiste) funkcji kwadratowej $f(x) = ax^2 + bx + c$. Wartości współczynników a, b i c powinny być podane przez użytkownika.

Podpowiedzi:

- Na początku sprawdzić, czy podane współczynniki opisują funkcję kwadratową. Kiedy nie opisują?
- Przypomnieć sobie wzory na deltę oraz miejsca zerowe.

Pętle

- Służą do automatycznego powtarzania instrukcji.
- W Pythonie istnieją dwa rodzaje pętli:
  - `while` – powtarza się dopóki jest spełniony warunek kończący;
  - `for` – powtarza się dla każdej wartości danego obiektu iterowalnego.

- Czym jest obiekt iterowalny w języku Python?
  O tym później…

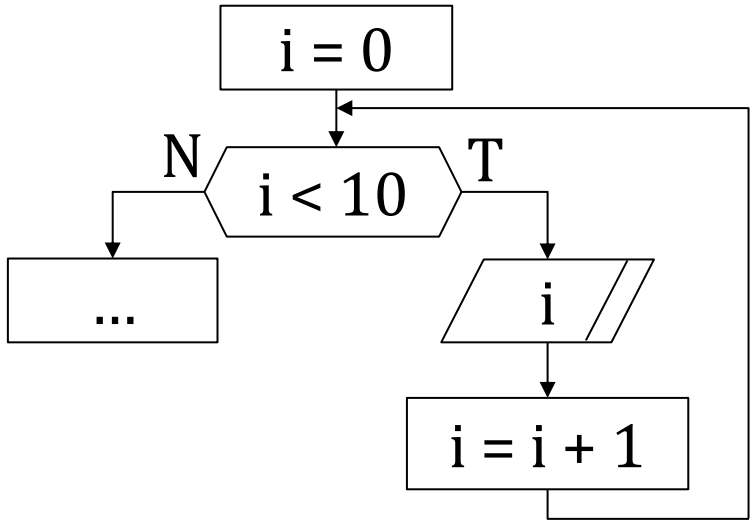
- **Uwaga dla programistów innych języków:** Python nie posiada pętli `do while`, czyli rodzaju pętli, w której warunek jest na końcu, a więc wykonuje się przynajmniej raz. A szkoda. ☹

Pętla `while`

- Pętla `while` powtarza się dopóki jest spełniony warunek kończący.

- W poniższym przykładzie warunkiem kończącym jest `i < 10`.

```
i = 0
while i < 10:
    print(i)
    i += 1
#...
```



Zadanie 4

- Napisać program, który wyświetli piramidę gwiazdek, taką jak na poniższym rysunku. Wysokość piramidy powinna być podana przez użytkownika.

```
*
**
***
****
*****
******
*******
********
*********
**********
```

Zadanie 5

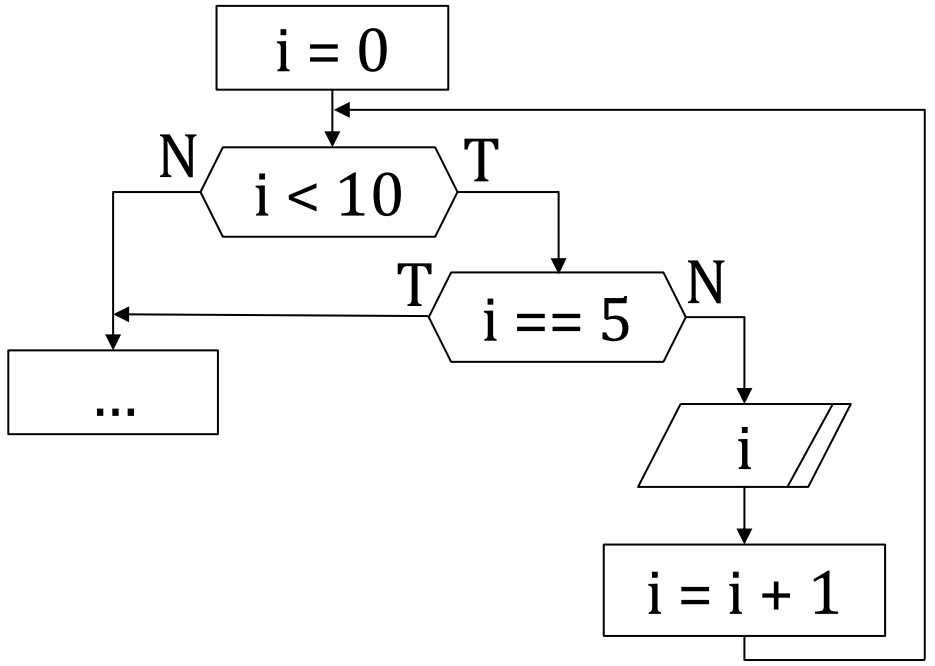
- Napisać program wczytujący *n*-elementowy ciąg liczb zmiennoprzecinkowych (*n* podajemy na wejściu) i obliczający sumę liczb dodatnich i (osobno) sumę liczb ujemnych.

Podpowiedź: skorzystać ze schematu blokowego z zadania 8, które było (lub będzie) omawiane na wykładzie z algorytmiki.

Instrukcja `break`

- Instrukcja `break` powoduje natychmiastowe przerwanie pętli.

```
i = 0
while i < 10:
    if i == 5:
        break
    print(i)
    i = i + 1
```



Przykład praktycznego zastosowania instrukcji `break`

- Program prosi użytkownika o podanie pięciu liczb dodatnich i wypisuje je.
- Jeśli użytkownik pomyłkowo poda liczbę niedodatnią, to pętla zostanie natychmiastowo przerwana.

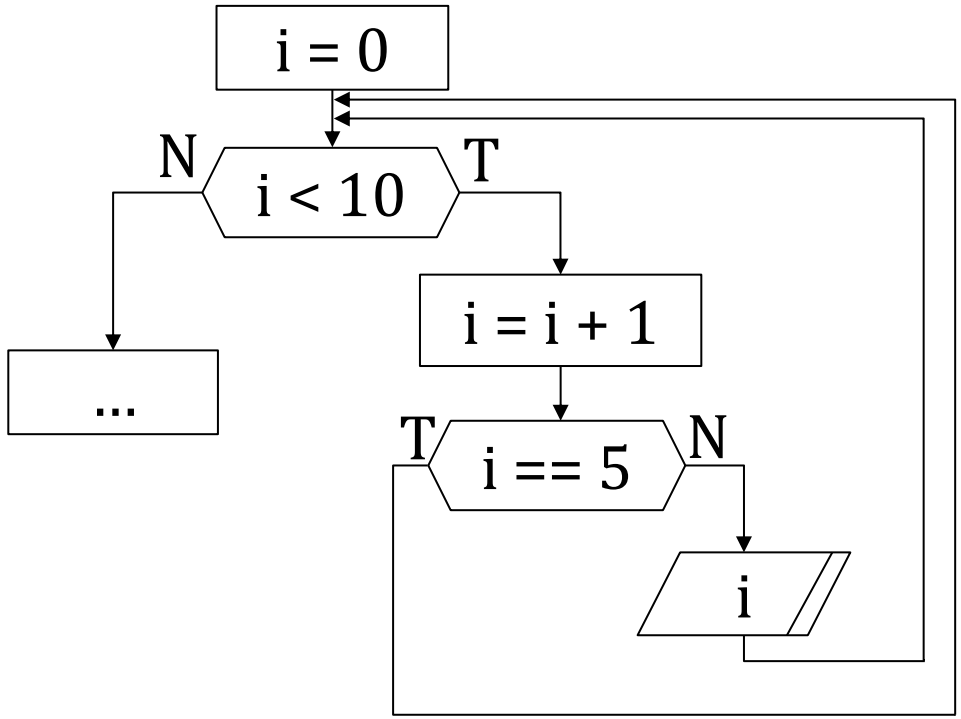
```
print('Podaj 5 liczb dodatnich.')
i = 0
while i < 5:
    a = int(input('Podaj liczbę dodatnią: '))
    if a <= 0:
        print('To nie jest liczba dodatnia! Koniec pętli.')
        break
    print('Podałeś liczbę dodatnią: ', a)
    i = i + 1
```

Instrukcja `continue`

- Instrukcja `continue` powoduje natychmiastowe przerwanie aktualnego obiegu pętli i przejście do następnego.

Przykład I

```
i = 0
while i < 10:
    i = i + 1
    if i == 5:
        continue
    print(i)
```



Instrukcja `continue`

- Te programy mogą nie działać tak jak oczekujemy.

- Jeden z nich zawiesi się po wypisaniu liczby 4, a drugi będzie wypisywał liczbę 5 w nieskończoność. Dlaczego?

Przykład II

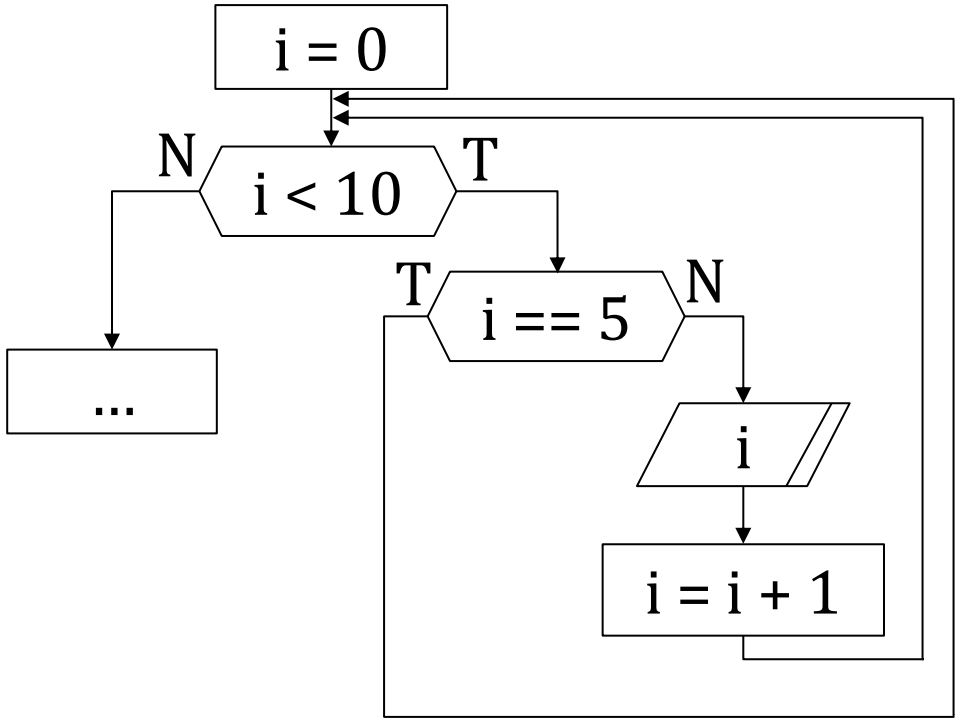
```
i = 0
while i < 10:
    if i == 5:
        continue
    print(i)
    i = i + 1
```

Przykład III

```
i = 0
while i < 10:
    print(i)
    if i == 5:
        continue
    i = i + 1
```

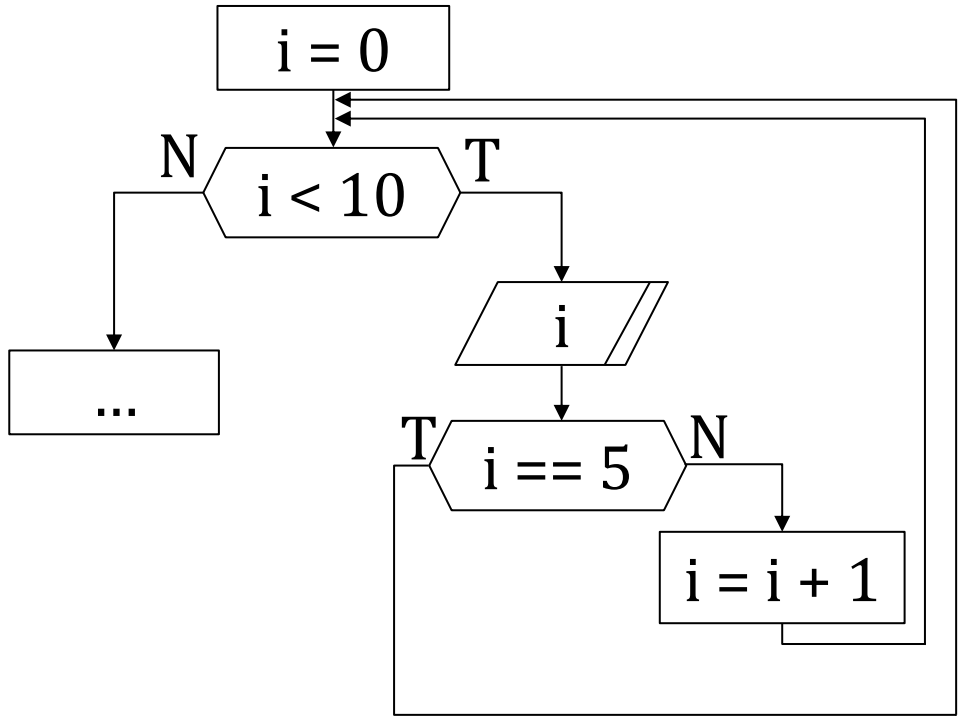
Instrukcja `continue` – przykład II

```
i = 0
while i < 10:
    if i == 5:
        continue
    print(i)
    i = i + 1
```



Instrukcja `continue` – przykład III

```
i = 0
while i < 10:
    print(i)
    if i == 5:
        continue
    i = i + 1
```



Przykład praktycznego zastosowania instrukcji `continue`

- Program prosi użytkownika o podanie 5 liczb dodatnich i wypisuje je.
- Jeśli użytkownik pomyłkowo poda liczbę niedodatnią, to aktualny obieg pętli zostanie przerwany (bez inkrementacji licznika i).
- Użytkownik będzie miał możliwość powtórzenia liczby w kolejnym obiegu.

```
print('Podaj 5 liczb dodatnich.')
i = 0
while i < 5:
    print('Podaj liczbę dodatnią nr', i+1)
    a = int(input())
    if a <= 0:
        print('To nie jest liczba dodatnia! Spróbuj jeszcze raz!')
        continue
    print('Podałeś liczbę dodatnią: ', a)
    i = i + 1
```

Pętla `for`

- Pętla `for` powtarza się dla każdej wartości danego obiektu iterowalnego.
- Jednym z obiektów iterowalnych jest zakres (przedział liczbowy) `range`.

```
i = 0
while i < 10:
    print(i)
    i += 1
```

```
for i in range(0,10):
    print(i)
```

```
for i in range(10):
    print(i)
```

Typy danych iterowalne

- `range` – przedział liczbowy
- `str` – tekst
- Kolekcje:
  - `list` – lista
  - `dict` (dictionary) – słownik
  - `tuple` – krotka
  - `set` – zbiór

Zmienne typu iterowalnego nazywamy obiektami iterowalnymi.

Instrukcja `continue` wewnątrz pętli `for`

- Pętla `for` automatycznie zwiększa licznik na końcu obiegu.
- Zostanie on zwiększony nawet wówczas, gdy zostanie wykonana instrukcja `continue` (przed przejściem do kolejnego obiegu).
- Odpowiada to sytuacji, w której w pętli `while` tuż przed `continue` zwiększylibyśmy licznik. A więc poniższe dwa programy są równoważne:

```
for i in range(10):
    print('ABC')
    continue
    print('DEF')
```

```
i = 0
while i < 10:
    print('ABC')
    i += 1
    continue
    print('DEF')
```

Listy – obiekty typu `list`

- **Uwaga dla programistów innych języków:** Listy (ang. lists) języka Python odpowiadają tablicom znanym z innych języków programowania.
- Listy umożliwiają przechowywanie wielu wartości w jednej zmiennej.
- Każda wartość ma swój indeks (numer pozycji).
- Indeksy są kolejnymi liczbami całkowitymi, zaczynającymi się od 0.

Listy – obiekty typu `list`

```
#pusta lista nie zawiera elementów; możemy je później dodać
pusta_lista = []
liczby = [4, 7, 0, -3, 5]
```

indeks	0	1	2	3	4
wartość	4	7	0	-3	5

```
liczby[1] = 5
```

indeks	0	1	2	3	4
wartość	4	**5**	0	-3	5

```
#dołączenie wartości 8 na koniec
liczby.append(8)
```

indeks	0	1	2	3	4	**5**
wartość	4	5	0	-3	5	**8**

```
#usuniecie elementu o indeksie 2
liczby.pop(2)
```

indeks	0	1	~~**2**~~	2	3	4
wartość	4	5	~~**0**~~	-3	5	8

```
#wypisujemy całą listę
print(liczby)
```

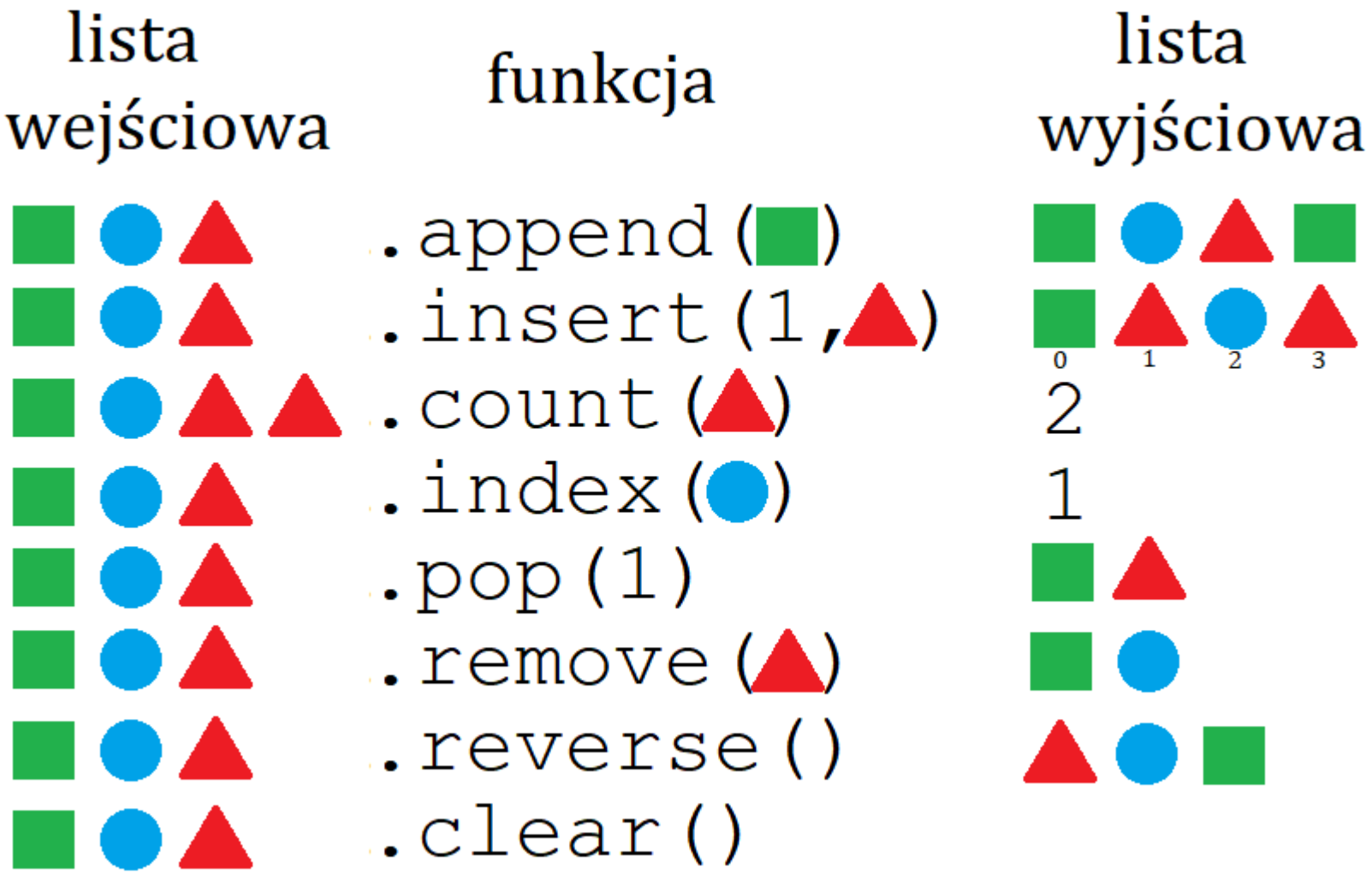
indeks	0	1	2	3	4
wartość	4	5	-3	5	8

Listy – przydatne triki i funkcje

```
lista_mieszana = [4, -6, 2.6, 'abc', False] #lista z wartościami o różnych typach
lista_mieszana[-1] = True #indeks -1 oznacza odwołanie się do ostatniego elementu

liczby = [6, 3, 8, 12, 1, 9]

print( max(liczby) ) #wyznaczenie największej wartości
print( min(liczby) ) #wyznaczenie najmniejszej wartości
print( len(liczby) ) #zwraca długość listy
print( sum(liczby) ) #oblicza sumę wszystkich elementów listy

liczby.insert(2, 100) #wstawienie do listy wartości 100 na pozycji 2
liczby.reverse() #odwrócenie kolejności elementów
liczby.sort() #sortowanie w porządku rosnącym
liczby.sort(reverse=True) #sortowanie w porządku malejącym
liczby.remove(12) #usunięcie elementu listy podając jego wartość (nie indeks)
#Uwaga: Jeśli w liście jest kilka elementów o takiej samej wartości, usunięty
#zostanie tylko pierwszy z nich.
```

Pytanie: Jak obliczyć średnią arytmetyczną wszystkich elementów listy korzystając z tych funkcji?

Listy – wizualne przedstawienie działania niektórych funkcji



Scalanie list operatorem +

```
liczby_dodatnie = [1, 2, 3, 4]
liczby_ujemne = [-4, -3, -2, -1]

#operator + użyty dla dwóch list powoduje ich scalenie (konkatenację)
liczby_mieszane = liczby_ujemne + liczby_dodatnie
print(liczby_mieszane)
```

indeks	0	1	2	3	4	5	6	7
wartość	-4	-3	-2	-1	1	2	3	4

Wczytywanie listy przy użyciu pętli `while` i `for`

```
liczby = []

i = 0
while i < 5:
    a = int(input('Podaj liczbę: '))
    liczby.append(a)
    i = i + 1
```

```
liczby = []

for i in range(5):
    a = int(input('Podaj liczbę: '))
    liczby.append(a)
```

Używanie wypełnionej listy przy użyciu pętli `while` i `for`

```
liczby = [1, 9, 2, 8, 3, 7, 4, 6, 5]

print('Pętla while:')
i = 0
while i < len(liczby):
    print(liczby[i])
    i = i + 1

print('Pętla for:')
for el in liczby:
    print(el)

print('Pętla for z indeksem:')
for i, el in enumerate(liczby):
    print(i, el)
```

Program wypisuje wszystkie elementy listy na trzy sposoby.

W trzecim przypadku wypisujemy nie tylko elementy listy, ale również ich indeksy.

Modyfikowanie wypełnionej listy przy użyciu pętli `while` i `for`

```
liczby = [2, 4, 6, 8]

i = 0
while i < len(liczby):
    liczby[i] = liczby[i] + 1
    i = i + 1
print(liczby)

for el in liczby:
    el = el + 1
print(liczby)

for i, el in enumerate(liczby):
    liczby[i] = liczby[i] + 1
print(liczby)
```

Program zwiększa wartość każdego elementu listy o 1 przy użyciu pętli `while` i `for`.

Uwaga: Elementów listy (i innych kolekcji) przetwarzanych w pętli `for` bez użycia indeksu i operatora `[]` nie można modyfikować. Można je wykorzystywać tylko do odczytu wartości.

Dlatego ten sposób nie zadziała. Lista nie zostanie zmodyfikowana.

Zadanie 6

- Napisać program wczytujący liczby całkowite do *n*-elementowej listy (*n* podajemy na wejściu). Następnie program powinien obliczyć sumę wszystkich elementów listy. Na koniec należy przemnożyć obliczoną sumę przez każdy element listy i zapisać wynik w liście.
- Program powinien wyświetlić obliczoną sumę i zaktualizowaną listę.

Używanie napisów (`str`) jak list

```
napis = 'NAPIS'

#można odczytywać poszczególne litery napisu tak jak elementy listy
print(napis[2]) #Jaka litera zostanie wypisana?

#nie możemy jednak w ten sposób modyfikować napisu
#ponieważ typ str jest niemutowalny (immutable)
napis[2] = 'W' #błąd!

#do odczytania napisu litera po literze można skorzystać z pętli for
for i in range(len(napis)):
    print(napis[i])

for litera in napis:
    print(litera)
```

Używanie napisów (`str`) jak list – edytowanie napisów

```
napis = 'NAPIS'

#jeśli chcemy mieć możliwość edytowania napisu (zmiany jego znaków),
#to musimy przekonwertować go na listę
napis = list(napis)
napis[2] = 'W'

#na koniec listę z powrotem konwertujemy na napis
napis = ''.join(napis)

print(napis) #jaki napis zostanie wyświetlony?
```

Wybieranie elementów listy lub fragmentów napisu z określonego przedziału

```
ptak = 'kuropatwa'

#Wybieramy znaki o indeksach z przedziału [2-6), czyli 2, 3, 4, 5
#i przypisujemy je do nowej zmiennej.
#Należy pamiętać, że początek przedziału jest zawsze zamknięty,
#a koniec otwarty (podobnie jak w funkcji range).
surowiec = ptak[2:6]
print(surowiec) #co zostanie wypisane?

#tak samo możemy postąpić z listą
liczby = [-3, -2, -1, 0, 1, 2, 3]
binarne = liczby[3:5]
dodatnie = liczby[4:] #pominięcie drugiej liczby oznacza „do końca”
ujemne = liczby[:3] #pominięcie pierwszej liczby oznacza „od początku”

print(binarne, dodatnie, ujemne)
```

Wybieranie elementów listy z określonego przedziału z krokiem innym niż 1

```
#Oprócz przedziału listy, możemy również określić jej
#krok po drugim dwukropku. Domyślnie krok wynosi 1.
liczby = [1,2,3,4,5,6,7,8,9,10]
nieparzyste = liczby[::2]
parzyste = liczby[1::2]

print(nieparzyste, parzyste)
```

Zadanie 7

- Napisać program, który wczytuje tekst z konsoli, a następnie zlicza małe i duże litery.
- Program powinien działać dla podstawowego alfabetu angielskiego (nie musi uwzględniać polskich znaków).

Odwzorowanie list (listy składane)

- Istnieje pewien szybszy sposób na utworzenie i wypełnienie listy przy użyciu pętli for.
- Jest to tzw. odwzorowanie list (ang. list comprehension).

```
lista = []
for i in range(10):
    lista.append(i*2)

print(lista)
```

```
lista = [i*2 for i in range(10)]

print(lista)
```

lista utworzona i wypełniona standardowo

lista odwzorowana

Odwzorowanie list (listy składane)

- Odwzorowanie list może się przydać również w innych sytuacjach, np. jeśli chcemy zmodyfikować elementy listy.
- Poniższy przykład zmniejsza wartość każdego elementu listy o 1.

```
lista = [4, 3, 2, 1]
for i, el in enumerate(lista):
    lista[i] -= 1

print(lista)
```

lista zmodyfikowana w standardowy sposób

```
lista = [4, 3, 2, 1]
lista = [el-1 for el in lista]

print(lista)
```

lista zmodyfikowana za pomocą odwzorowania

Odwzorowanie list – podsumowanie

- Odwzorowanie list może skrócić kod i przyspieszyć jego pisanie.
- Oprócz tego, kod wykonuje się szybciej niż w przypadku standardowych pętli. Zaleca się więc używać odwzorowania szczególnie wtedy, gdy przetwarzamy bardzo duże listy (lub inne kolekcje).
- Trzeba się jednak z tą techniką „oswoić”.
- Początkujący programiści mogą używać standardowych sposobów. Najważniejsze, żeby program, który piszemy był dla nas czytelny i zrozumiały.

Listy zagnieżdżone (dwuwymiarowe)

- Listy zagnieżdżone (dwuwymiarowe), to listy, których elementy są również listami.
- Do poszczególnych elementów listy 2D odnosimy się podając dwa indeksy (współrzędne elementu).

```
#tworzenie listy 2D - sposób I
wiersz_1 = [1, 3, 5, 7]
wiersz_2 = [2, 4, 6, 8]
wiersz_3 = [1, 2, 3, 4]
wiersz_4 = [9, 8, 7, 6]
lista2D = [wiersz_1, wiersz_2, wiersz_3, wiersz_4]
```

indeksy	**0**	**1**	**2**	**3**
0	1	3	5	7
1	2	4	6	8
2	1	2	3	4
3	9	8	7	6

```
#tworzenie listy 2D - sposób II
lista2D = [[1, 3, 5, 7], [2, 4, 6, 8], [1, 2, 3, 4], [9, 8, 7, 6]]
```

Listy zagnieżdżone (dwuwymiarowe)

```
lista2D = [[1, 3, 5, 7], [2, 4, 6, 8], [1, 2, 3, 4], [9, 8, 7, 6]]

print(lista2D[2][3]) #jaka liczba zostanie wypisana?

#pętle wypisujące wszystkie elementy listy
for wiersz in lista2D:
    for element in wiersz:
        print(element, end = ' ')
    print()

#pętle modyfikujące elementy listy
for i, wiersz in enumerate(lista2D):
    for j, element in enumerate(wiersz):
        lista2D[i][j] *= 2

print(lista2D)
```

Taki zapis powoduje, że na koniec wyświetlanego napisu zostaje wstawiona spacja (zamiast domyślnego znaku przejścia do nowej linii).

Listy zagnieżdżone - zastosowania

- Listy 2D stosujemy np. do przechowywania wartości macierzy (matematyka).

- Innym praktycznym zastosowaniem jest przechowywanie danych pikseli obrazów szarych (programy przetwarzające obrazy).

- Jeśli przetwarzamy obrazy kolorowe, np. 3-kanałowe RGB, to potrzebujemy list trójwymiarowych:
  - I wymiar – wiersz piksela,
  - II wymiar – kolumna piksela,
  - III wymiar – kanał (czerwony, zielony, niebieski).

Zadanie 8

- Napisać program wczytujący z konsoli macierz do dwuwymiarowej listy o wymiarach $n \times n$ (n podajemy na wejściu). Następnie program powinien podnieść do potęgi drugiej wszystkie elementy na przekątnej macierzy (od lewego-górnego do prawego-dolnego rogu), a pozostałe elementy wyzerować.

Słowniki – obiekty typu `dict`

- Uwaga dla programistów innych języków: Słowniki (ang. dictionaries) języka Python odpowiadają tablicom asocjacyjnym lub mapom znanym z innych języków programowania.
- Słowniki, podobnie jak listy, umożliwiają przechowywanie wielu wartości w jednej zmiennej.
- Każda wartość ma swój indeks.
- Indeksy słowników, w przeciwieństwie do list, nie muszą być liczbami całkowitymi. Mogą być wartościami dowolnego typu podstawowego, czyli np. `int`, `float`, `str`.
- Najczęściej stosuje się napisy (`str`) jako indeksy słowników.

Słowniki – obiekty typu `dict`

```
#pusty słownik nie zawiera elementów; możemy je później dodać
pusty_slownik = {}

#słownik przechowujący nazwy firm (indeksy) i ich lata założenia (wartości)
firmy = {'Microsoft':1975, 'Sony':1946, 'Nintendo':1889}

#dodanie nowej firmy (metoda append nie działa dla słowników)
firmy['Valve'] = 1996

#funkcja len zwraca liczbę elementów słownika (tak samo jak w przypadku list)
print('Liczba firm w bazie: ', len(firmy))
print('Trudno w to uwierzyć, ale firma Nintendo powstała w roku ', firmy['Nintendo'])
```

Indeks (nazwa firmy)	'Microsoft'	'Sony'	'Nintendo'	'Valve'
Wartość (rok założenia)	1975	1946	1889	1996

Indeksy (klucze) słowników

- Czasami indeksy słowników nazywa się kluczami. Mówi się wtedy, że słownik opisują pary klucz:wartość.

- Np. w poniższym słowniku kluczem pierwszego elementu jest napis „Microsoft”, natomiast wartością jest 1975.

```
firmy = {'Microsoft':1975, 'Sony':1946, 'Nintendo':1889}
```

Wczytywanie słownika przy użyciu pętli `while` i `for`

```
firmy = {}

i = 0
while i < 3:
    firma = input('Podaj nazwę firmy: ')
    rok = int(input('Podaj rok założenia: '))
    firmy[firma] = rok
    i += 1
```

```
firmy = {}

for i in range(3):
    firma = input('Podaj nazwę firmy: ')
    rok = int(input('Podaj rok założenia: '))
    firmy[firma] = rok
```

Przetwarzanie wypełnionego słownika przy użyciu pętli `for`

- Nie ma możliwości przetworzenia wypełnionego słownika korzystając z pętli `while`, ponieważ indeksy słowników nie są kolejnymi liczbami całkowitymi.

- Jeśli użyjemy pętli `for` w standardowy sposób (jak na poniższym przykładzie), to wypisane zostaną jedynie indeksy (nazwy firm), a nie wartości (lata).

```
firmy = {'Microsoft':1975, 'Sony':1946, 'Nintendo':1889, 'Valve':1996}

for firma in firmy:
    print(firma)
```

Elementy słownika są przetwarzane w pętli `for` w kolejności takiej, w jakiej zostały do niego dodane.

Przetwarzanie wypełnionego słownika przy użyciu pętli `for`

- Jak więc możemy wypisać wszystkie nazwy firm wraz z ich latami (indeksy i wartości) za pomocą pętli `for`?
- Istnieją dwa sposoby:

```
firmy = {'Microsoft':1975, 'Sony':1946, 'Nintendo':1889, 'Valve':1996}
```

Pierwszy sposób:

```
for firma, rok in firmy.items():
    print('Firma', firma, 'powstała w', rok, ,r.')
```

Drugi sposób:

```
for firma in firmy:
    print('Firma', firma, 'powstała w', firmy[firma], ,r.')
```

Zadanie 9

- Napisać program umożliwiający wczytanie do słownika *n* produktów (*n* podajemy na wejściu) sklepu spożywczego i ich ceny. Następnie program powinien obniżyć ceny wszystkich produktów o 5%. Na koniec program powinien wypisać produkty w dwóch osobnych grupach: „tanie produkty” (poniżej 10 zł) i „drogie produkty” (10 zł i więcej).

Krotki – obiekty typu `tuple`

- Listy są typem mutowalnym (ang. mutable), co oznacza, że można zmieniać ich elementy.

- Krotki są niemutowalnym (ang. immutable) odpowiednikiem list. Oznacza to, że mają większość właściwości list, ale nie można zmieniać elementów raz utworzonej krotki.

- Kiedy i po co używać krotek zamiast list?
  - Krotki zajmują mniej miejsca w pamięci, szybciej się przetwarzają i są bezpieczniejsze (kod mniej podatny na błędy).
  - Warto je używać, jeśli chcemy utworzyć listę, która nie będzie modyfikowana.

Krotki – obiekty typu `tuple`

```
#krotkę możemy utworzyć podając jej elementy w nawiasie okrągłym po przecinku
krotka_liczbowa_1 = (1, 5, 8, -4)

#możemy pominąć nawias i po prostu wymienić elementy po przecinku
krotka_liczbowa_2 = 0, -3, 12, 8

#krotka może mieć wartości dowolnego typu
krotka_mieszana = 2, 5.3, True, 'napis', (1, 5) #ostatni element też jest krotką

#wyświetlanie pojedynczego elementu krotki
print(krotka_mieszana[1])

krotka_mieszana[2] = False #błąd - nie można zmieniać elementów krotki

#pętle for również działają z krotkami
for element in krotka_mieszana:
    print(element)
```

Przydatny trik z użyciem krotek – zamiana wartości dwóch zmiennych

- Za pomocą krotek możemy w szybki, jednolinijkowy sposób zamienić wartości dwóch zmiennych.

standardowy sposób (bez krotek) używany we wszystkich językach programowania

```
A = 1
B = 2
pomocnicza = A
A = B
B = pomocnicza
print(A)
print(B)
```

zamiana przy użyciu krotek – sposób specyficzny dla języka Python

```
A = 1
B = 2

A,B = B,A

print(A)
print(B)
```

Zbiory – obiekty typu `set`

- Zbiory służą do przechowywania wielu wartości, podobnie jak listy, słowniki i krotki.
- W przeciwieństwie do list, słowników i krotek, wartości zbioru nie mają indeksów.
- W przeciwieństwie do list, słowników i krotek, wartości zbioru nie mogą się powtarzać.
- Zbiory w języku Python mają podobne właściwości jak zbiory w matematyce.

Zbiory – obiekty typu `set`

```
#Podstawowy sposób na utworzenie zbioru
zbior_1 = {'jeden', 'trzy', 'siedem'}

#Zbiór można też utworzyć konwertując listę lub krotkę funkcją set
zbior_2 = set([1, 8, -4])
zbior_3 = set(('abc', False, 45.6))

#wartości w zbiorze nie mogą się powtarzać
#jeśli powtórzymy wartość zbioru, zostanie ona pominięta
zbior_4 = {2, 4, 1, 4}
print(zbior_4)

print(zbior_1[2]) #błąd - nie można odnosić się do elementów zbioru poprzez indeksy

#pętle for również działają ze zbiorami
for element in zbior_1:
    print(element)
```

Kolejność przetwarzania elementów zbioru w pętli `for` jest nieprzewidywalna.

Operacje na zbiorach

```
zbior_1 = {'jeden', 'trzy', 'siedem'}
print(zbior_1)

#funkcja add dodaje element do zbioru
zbior_1.add('osiem')
print(zbior_1)

#funkcja discard usuwa element ze zbioru
zbior_1.discard('trzy')
print(zbior_1)

#funkcja clear czyści zbiór
zbior_1.clear()
#po wyczyszczeniu zbiór ma 0 elementów, a więc jest to zbiór pusty
print(zbior_1)
print(len(zbior_1))
```

Operacje na zbiorach

```
ssaki = {'lew', 'słoń', 'delfin'}
zwierzeta_wodne = {'rekin', 'delfin', 'ośmiornica'}

#suma zbiorów
suma = ssaki.union(zwierzeta_wodne)
#część wspólna (iloczyn, przekrój, przecięcie) zbiorów
czesc_wspolna = ssaki.intersection(zwierzeta_wodne)
#różnica zbiorów
roznica = ssaki.difference(zwierzeta_wodne)
#różnica symetryczna zbiorów
roznica_symetryczna = ssaki.symmetric_difference(zwierzeta_wodne)

print(suma)
print(czesc_wspolna)
print(roznica)
print(roznica_symetryczna)
```

Sprawdzanie, czy podany element występuje w liście, krotce lub zbiorze

- Możemy sprawdzić, czy podany element występuje w liście, krotce lub zbiorze za pomocą operatora `in`.

- Jeśli element będący lewym operandem występuje w liście/krotce/zbiorze będącym prawym operandem, zwrócona zostaje wartość `True`.
  W przeciwnym wypadku – wartość `False`.

```
liczby_pierwsze = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
print(19 in liczby_pierwsze)
print(9 in liczby_pierwsze)

owady = {'komar', 'modliszka', 'biedronka', 'osa'}
print('modliszka' in owady)
print('pająk' in owady)
```

Sprawdzanie, czy podany element występuje w słowniku

- Operator `in` można używać również na słownikach, ale jego działanie jest nieco inne, niż w przypadku list, krotek i zbiorów.
- Sprawdza on, czy w słowniku występuje element o podanym indeksie (nie o podanej wartości).

```
menu = {'ogórkowa':5.99, 'jarzynowa':4.99, 'barszcz':5.49}
print('jarzynowa' in menu) #jest
print(5.99 in menu) #nie ma (5.99 to nie indeks, tylko wartość)
```

- Jeśli jednak chcemy koniecznie sprawdzić, czy w podanym słowniku istnieje jakaś wartość (nie indeks), to możemy użyć funkcję `values`:

```
print(5.99 in menu.values()) #jest
```

Obiekty iterowalne – mutowalne i niemutowalne

mutowalne	**niemutowalne**
lista (`list`)	krotka (`tuple`)
zbiór (`set`)	tekst (`str`)
słownik (`dict`)	

Kolekcje - podsumowanie

- Ten slajd ma za zadanie pomóc w odróżnieniu poszczególnych typów kolekcji w Pythonie.

```
lista = [2, 4, 6, 8] #nawias kwadratowy, wartości oddzielone przecinkami
print( lista[1] )

#nawias klamrowy, pary indeks:wartość oddzielone przecinkami
slownik = {'a':2, 'b':4, 'c':6}
print( slownik['b'] )

krotka_1 = (2, 4, 6) #nawias okrągły, wartości oddzielone przecinkami
krotka_2 = 2, 4, 6 #brak nawiasu, wartości oddzielone przecinkami
print ( krotka_1[1] )

zbior = {2, 4, 6} #nawias klamrowy, wartości oddzielone przecinkami
#do pojedynczych elementów zbioru możemy odwoływać się tylko poprzez pętlę for
for element in zbior:
    print(element)
```

Definiowanie (tworzenie) własnych funkcji

- Funkcje są podprogramami, czyli wydzielonymi fragmentami programów, które można wywoływać (uruchamiać) w celu wykonania określonych operacji.

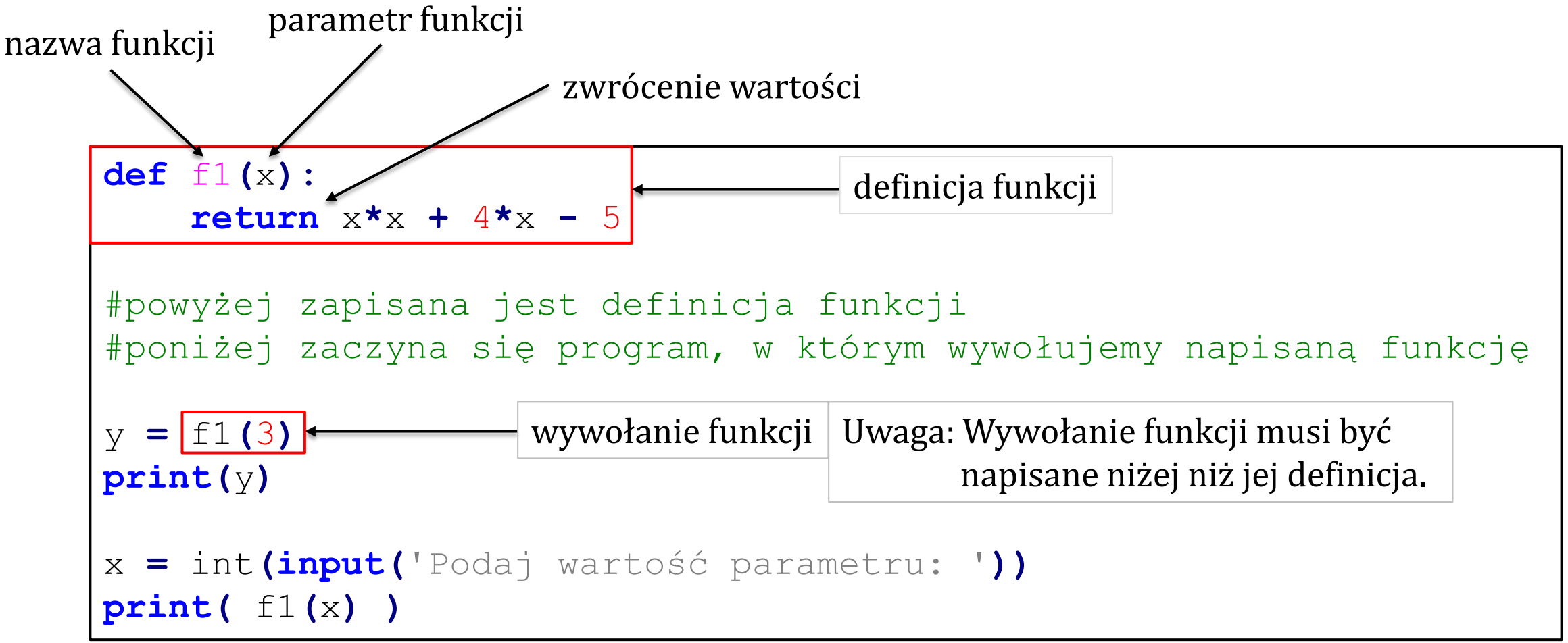
- Funkcje mogą przyjmować parametry (argumenty). Możliwe są również funkcje bezparametrowe.

- Funkcje mogą zwracać wartość po zakończeniu swoich operacji. Wartość ta może zostać wykorzystana przez program, który funkcję wywołał.

- Po co pisze się funkcje?
  1. Aby nie pisać tych samych lub podobnych grup instrukcji wielokrotnie w różnych miejscach programu. Funkcję piszemy raz, a potem można ją wielokrotnie wywołać (pojedynczą instrukcją).
  2. Aby znacznie zwiększyć czytelność kodu (do tego stopnia, że w poważnych programach pisanie kodu bez funkcji nie ma sensu).

Funkcje - składnia

- Poniższa funkcja oblicza wartość wyrażenia $x^2 + 4x - 5$.



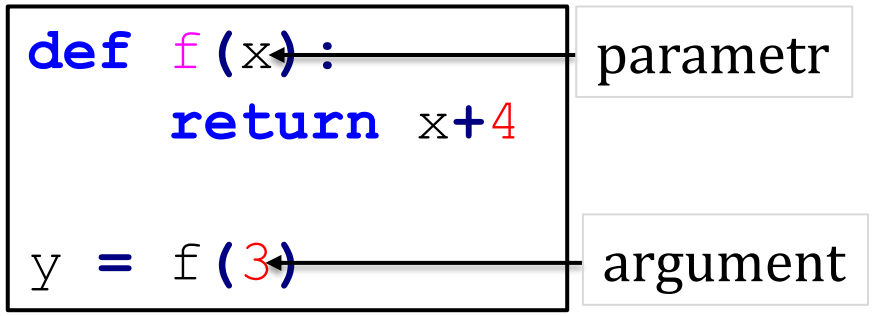
Funkcje - przykład

- Poniższa funkcja oblicza wartość wyrażenia $2x^2 - \sqrt{y}$.

```
#funkcje mogą przyjmować wiele parametrów
#poniższa funkcja ma dwa parametry: x, y
def f2(x,y):
    if y < 0:
        return 'Błąd! Tego nie można obliczyć w dziedzinie liczb rzeczywistych.'
    else:
        return 2*x*x - y**0.5

#tu zaczyna się program
wynik = f2(3,5)
print(wynik)

x1 = int(input('Podaj wartość pierwszego parametru: '))
x2 = int(input('Podaj wartość drugiego parametru: '))
#nazwy zmiennych podawanych przy wywołaniu nie muszą być takie same, jak nazwy
#parametrów w definicji funkcji
print( 'Wynik funkcji:', f2(x1,x2) )
```

Parametr i argument

- Pojęcia parametr i argument nie oznaczają dokładnie tego samego.
- Parametr wpisuje się przy definiowaniu funkcji.
- Argument jest wartością parametru podawaną podczas wywołania funkcji.



- Często jednak programiści używają tych pojęć zamiennie.

Funkcja niezwracająca wartości - przykład

```
#poniższa funkcja nie zwraca żadnej wartości
#jej zadaniem jest wypisanie tekstu w odpowiednim formacie
def przedstawSie(imie, nazwisko, miejsce_ur):
    print(f'Nazywam się {imie} {nazwisko}. Pochodzę z miejscowości {miejsce_ur}.')

#tu zaczyna się program
przedstawSie('Dawid', 'Warchoł', 'Rzeszów')
przedstawSie('Ignacy', 'Łukasiewicz', 'Zaduszniki')
```

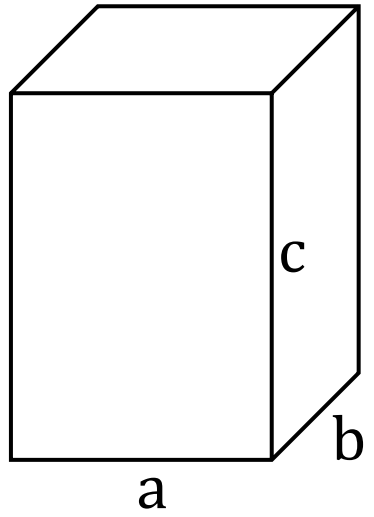
- Zasady dotyczące nazywania funkcji są identyczne, jak w przypadku nazw zmiennych.
- Nazwa funkcji powinna odzwierciedlać czynność, którą ta funkcja wykonuje.

Funkcja bezparametrowa - przykład

```
#poniższa funkcja nie zawiera parametrów
#jej zadaniem jest wczytanie z konsoli parametrów trapezu
def wczytajDaneTrapezu():
    a = float(input('Podaj długość pierwszej podstawy: '))
    b = float(input('Podaj długość drugiej podstawy: '))
    h = float(input('Podaj wysokość: '))
    return [a, b, h]

#tu zaczyna się program
trapez = wczytajDaneTrapezu()
pole_trapezu = (trapez[0]+trapez[1])*trapez[2]/2 #(a+b)*h/2
print('pole wczytanego trapezu:', pole_trapezu)
```

Wywołanie funkcji w innej funkcji

```
#program obliczający i wypisujący pole powierzchni
#całkowitej prostopadłościanu:
def polePowierzchniBocznej(a, b, c):
    return 2*a*c + 2*b*c

def polePowierzchniPodstaw(a, b):
    return 2*a*b

def polePowierzchniCalkowitej(a, b, c):
    pole = polePowierzchniBocznej(a, b, c) + polePowierzchniPodstaw(a, b)
    print('pole powierzchni całkowitej:', pole)

#tu zaczyna się program
polePowierzchniCalkowitej(4, 6, 2)
polePowierzchniCalkowitej(7, 3, 6)
```



Zadanie 10

- Napisz definicję funkcji `wczytajKsiazke`, która wczytuje z konsoli dane książki: tytuł, rok wydania i liczbę stron. Dane powinny zostać zapisane w słowniku.
- Następnie napisz funkcję `porownajKsiazki`, która jako parametry przyjmuje słowniki opisujące dwie książki i wypisuje informacje o tym, która z książek jest starsza oraz która jest dłuższa.
- Program powinien wywołać napisane funkcje w celu wczytania dwóch przykładowych książek i porównania ich.

Parametry funkcji z wartością domyślną

- Jeśli spodziewamy się, że jakiś parametr najczęściej będzie przyjmował jakąś jedną konkretną wartość, a inne rzadziej, to można mu przypisać wartość domyślną w definicji funkcji.

- Jeśli przy wywołaniu funkcji nie zostanie podana wartość tego parametru, to automatycznie wpisze się wartość domyślna.

Parametry funkcji z wartością domyślną - przykład

```
#skala jest parametrem o domyślnej wartości 'C'
def zamienNaFahrenheita(stopnie, skala='C'):
    if skala == 'C': #stopnie Celsjusza
        return 9/5*stopnie + 32
    elif skala == 'K': #kelwiny
        return 9/5*stopnie - 459.67
    else:
        return 'Błąd! Nieznana skala.'

print( zamienNaFahrenheita(20, 'C') )
#nie podano drugiego parametru, więc domyślnie wpisało się 'C'
print( zamienNaFahrenheita(20) )
print( zamienNaFahrenheita(290, 'K') )
print( zamienNaFahrenheita(15, 'X') )
```

Parametry funkcji z wartością domyślną - przykład

- Wartości domyślne można przypisać wielu parametrom, ale muszą to być ostatnie parametry funkcji.

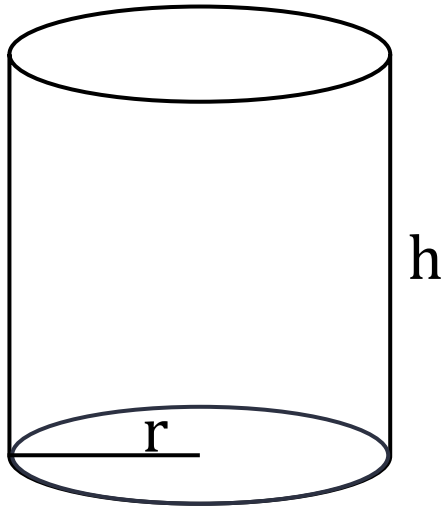
```
def funkcja(a, b=2, c=0): #OK

def funkcja(a='coś'): #OK

def funkcja(a='coś', b, c='nic'): #błąd!
```

Przekazywanie parametrów podczas wywoływania funkcji

- Istnieją dwa sposoby przekazywania parametrów do funkcji.

- Sposób I – parametry pozycyjne (używany w dotychczasowych przykładach)
  - Parametry podaje się w odpowiedniej kolejności
- Sposób II – parametry nazwane
  - Parametry podaje się w dowolnej kolejności podając ich nazwy

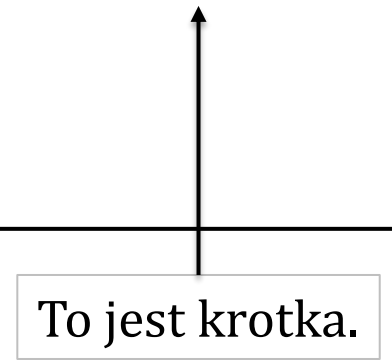
```
def objetoscWalca(r, h):
    return 3.14*r**2*h

d = objetoscWalca(3, 7) #sposób I
print(d)
d = objetoscWalca(h=7, r=3) #sposób II
print(d)
```



Zmienna liczba parametrów funkcji

- Istnieją dwa sposoby na napisanie funkcji, która może przyjmować dowolną liczbę parametrów.
- Sposób I: Parametr poprzedzony gwiazdką * powoduje, że wywołując funkcję możemy w jego miejsce wstawić dowolną liczbę parametrów pozycyjnych. Funkcja obsługuje te parametry w formie krotki.

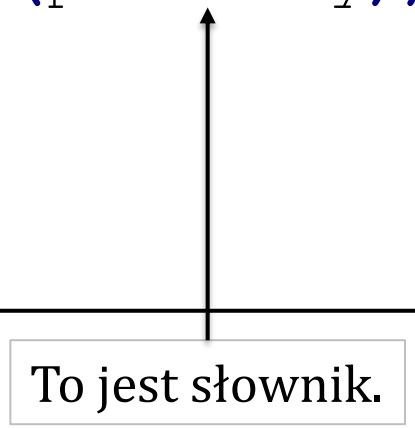
```
def funkcja(*parametry):
    print('Liczba parametrów:', len(parametry))
    print('Suma wszystkich parametrów:', sum(parametry))

funkcja(1, 2, 3)
funkcja(2, -2, 3, -3, 4, -4)
```



Zmienna liczba parametrów funkcji

- Sposób II: Parametr poprzedzony dwiema gwiazdkami ** powoduje, że wywołując funkcję możemy w jego miejsce wstawić dowolną liczbę parametrów nazwanych. Funkcja obsługuje te parametry w formie słownika.

```
def funkcja(**parametry):
    print('Liczba parametrów:', len(parametry))
    print(parametry)

funkcja(x1=1, x2=2)
funkcja(x1=1.0, x2=2.0, x3=3.0)
```



Zmienne globalne

- Funkcje mogą korzystać z trzech rodzajów zmiennych: parametry (przekazane do funkcji), zmienne lokalne (utworzone wewnątrz funkcji) oraz zmienne globalne (utworzone poza funkcjami).
- Funkcje z poniższego programu korzystają ze zmiennej `x`, która została utworzona poza funkcjami. Jest to zmienna globalna.

```
def funkcja_1():
  print("Zmienna globalna ma wartość:", x)

def funkcja_2():
  print([x, x**2, x**3])

x = 3
funkcja_1()
funkcja_2()
```

Zmienne globalne

- Należy jednak pamiętać, że w funkcji nie można korzystać ze zmiennej lokalnej, utworzonej wewnątrz innej funkcji. Poniższy kod nie zadziała.

```
def funkcja_1():
  x = 3

def funkcja_2():
  print(x)

funkcja_1()
funkcja_2()
```

Zmienne globalne

- Jeśli chcemy mieć możliwość zmiany wartości zmiennej globalnej (a nie tylko jej odczytania) wewnątrz funkcji, należy na początku funkcji określić, że jest ona globalna przy pomocy słowa `global`.

```
def funkcja_1():
    global x
    x *= 10

def funkcja_2():
    print(x)

x = 3
funkcja_1()
funkcja_2()
```

Jeśli usuniemy tę linijkę, to program nie zadziała. Nie uda się zmienić wartości zmiennej globalnej `x`.

Funkcje rekurencyjne

- Rekurencja (rekursja) jest to wywoływanie funkcji przez samą siebie.

- Funkcja rekurencyjna jest funkcją, w której definicji znajduje się przynajmniej jedno wywołanie samej siebie.

```
def fr(x):
    print(x)
    if x == 10:
        return 0
    else:
        return fr(x+1)

print(fr(5))
```

warunek kończący rekurencję

wywołanie rekurencyjne

Co wypisze funkcja?

Funkcje rekurencyjne - przykład

```
def fr(x):
    print(x)
    if x == 10:
        return 0
    else:
        return fr(x+1)

print(fr(5))
```

warunek kończący rekurencję

wywołanie rekurencyjne

Co wypisze funkcja?

Kolejne wywołania rekurencyjne:

`fr(5)→fr(6)→fr(7)→fr(8)→fr(9)→fr(10)`

Koniec rekurencji – przekazywanie wartości 0 od funkcji `f(10)` aż do `f(5)`:

`fr(10)→fr(9)→fr(8)→fr(7)→fr(6)→fr(5)`

W tym miejscu zostaje spełniony warunek kończący rekurencję i zaczynają się powroty. Oznacza to, że wartość 0 jest przekazywana do wywołania `f(5)` przechodząc po drodze przez wszystkie pośrednie wywołania. Na koniec wartość 0 zostaje wypisana.

Funkcje rekurencyjne – dodatkowe informacje

- W trakcie wykonywania się funkcji, jej parametry są odkładane w specjalnym obszarze pamięci zwanym stosem.
- Tworząc funkcję rekurencyjną należy poprawnie określić warunek kończący rekurencję. Nie może ona być nieskończona lub zbyt głęboka.
- W przeciwnym wypadku mogłoby dojść do przepełnienia stosu (ang. stack overflow).
- Język Python ma ustawioną maksymalną głębokość rekurencji na 1000, aby nie dopuścić do przepełnienia stosu. Jeśli przekroczymy tą głębokość, to program zostanie przerwany z błędem: „maximum recursion depth exceeded”.

```
def fr(x):
    print(x)
    return fr(x+1)

print(fr(5))
```

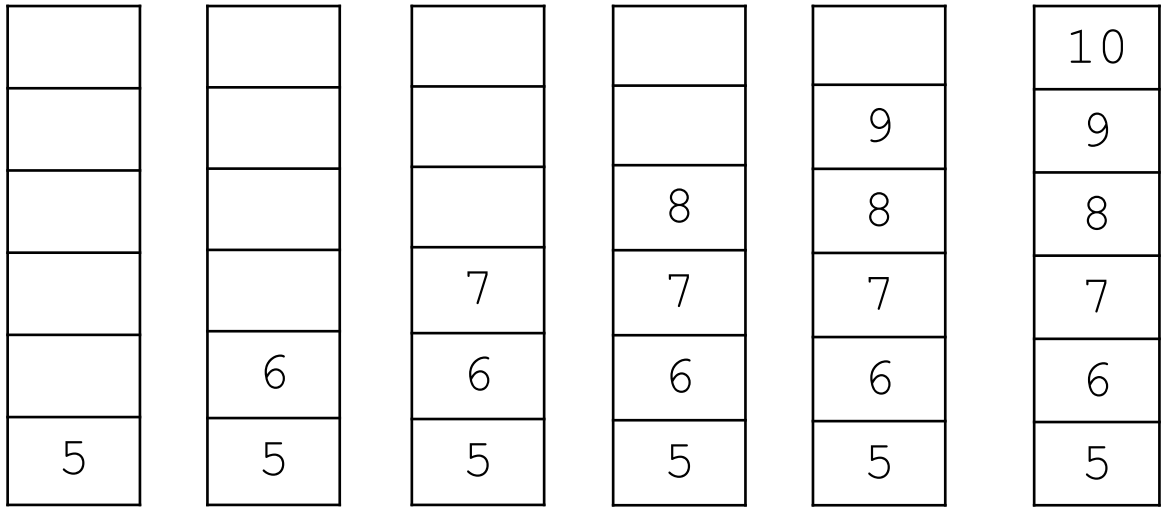
Brak warunku kończącego rekurencję. Błąd! Program się przerwie, aby nie dopuścić do przepełnienia stosu.

Funkcje rekurencyjne – stos

- W kolejnych krokach zilustrowano pamięć stosu zmieniającą się po kolejnych wywołaniach rekurencyjnych funkcji `fr` oraz po osiągnięciu końca rekurencji:

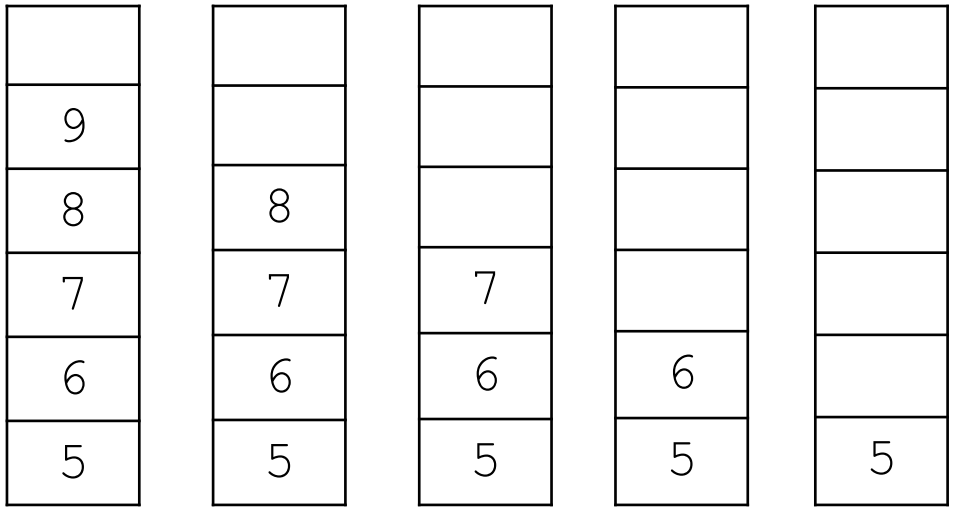
Kolejne wywołania rekurencyjne:

`fr(5)→fr(6)→fr(7)→fr(8)→fr(9)→fr(10)`



Koniec rekurencji – przekazywanie wartości 0:

`→fr(9)→fr(8)→fr(7)→fr(6)→fr(5)`



- Warto zwrócić uwagę, że pamięć stosu działa na zasadzie last-in-first-out, tzn. ostatni element wrzucony na stos jest pierwszym, który będzie z niego zdejmowany.

Funkcje rekurencyjne – dodatkowe informacje

- Funkcje rekurencyjne mają zastosowanie w językach programowania funkcyjnych (funkcjonalnych), w których zastępuje ona iteracje (pętle).

- Rekurencje można jednak stosować w językach strukturalnych i obiektowych. Często powoduje ona znaczne uproszczenie kodu, jednak może (ale nie musi) wiązać się z długim czasem obliczeń.

- Konkretny przykład zastosowania rekurencji stanowi zadanie z kolejnego slajdu.

Zadanie 11

- Napisz rekurencyjną definicję funkcji `fibonacci_rek` obliczającą *n*-ty wyraz ciągu Fibonacciego (*n* podajemy jako parametr).
- Należy skorzystać z rekurencyjnego wzoru:

$$Fn = \begin{cases} 0 & dla\ n = 0 \\ 1 & dla\ n = 1 \\ Fn - 1 + Fn - 2 & dla\ n > 1 \end{cases}$$

Przykłady praktycznego wykorzystania rekurencji

- Poszukiwanie pliku o danej nazwie na dysku (lub w folderze, w którym znajdują się inne foldery).
- Sortowanie liczb (algorytmy sortowania szybkiego i sortowania przez scalanie).
- Wyszukiwanie liczb w posortowanej liście (algorytm wyszukiwania binarnego).
- Grafika komputerowa:
  - proceduralne generowanie terenów, tekstur i obiektów (np. drzew);
  - generowanie fraktali (tzw. obiektów samopodobnych).

Źródło: *https://pl.wikipedia.org/wiki/Paproć_Barnsleya*

Programowanie obiektowe

- Paradygmat programowania, w którym program stanowi zbiór obiektów wykazujących interakcje między sobą.

- Czym jest obiekt w języku Python? Jest on po prostu zmienną. Gdy jednak mówimy o obiektach, najczęściej nie mamy na myśli zmiennych, które mają prosty typ, jak np. `int`, `float`. Są to zmienne bardziej rozbudowanych typów, np. lista, słownik albo typów, które tworzy sam programista.

- Własny typ obiektów możemy utworzyć (zdefiniować) za pomocą tzw. klas.

Programowanie obiektowe – przykładowa klasa w pseudokodzie

- Uwaga: To nie jest Python!

```
klasa Zwierzę
    @ masa_ciała
    @ kolor
    * poruszaj_się
    * daj_głos

klasa Ptak(Zwierzę)
    @ rozpiętość_skrzydeł
    * leć
```

- `@` oznacza pole (zmienną składową) klasy.
- `*` oznacza metodę (funkcję składową) klasy.

Pola i metody są często nazywane **atrybutami klasy.**

- Nawias oznacza dziedziczenie klas.
  - Jeśli klasa A dziedziczy po klasie B, to uzyskuje dostęp do wszystkich jej pól i metod.
  - Klasa dziedzicząca (A) nazywana jest podklasą lub klasą podrzędną (ang. subclass).
  - Klasa, po której dziedziczymy (B), nazywa się nadklasą, klasą nadrzędną (ang. superclass) lub klasą bazową (ang. base class).

Programowanie obiektowe – przykładowa klasa w Pythonie

- Przykład z poprzedniego slajdu napisany w Pythonie.
- Metody w tym przykładzie nie mają ciała; mają jedynie nagłówki.

```
class Zwierze:
    masa_ciala = 0 #[kg]
    kolor = ''
    def poruszaj_sie(self, odleglosc):
        #...
    def daj_glos(self):
        #...

class Ptak(Zwierze):
    rozpietosc_skrzydel = 0 #[m]
    def lec(self, wysokosc):
        #...
```

pola (zmienne klasy)

metody (funkcje klasy)

dziedziczenie klasy `Ptak` po klasie `Zwierze`

Programowanie obiektowe – tworzenie obiektów klas

```
class Zwierze:
    masa_ciala = 0 #[kg]
    kolor = ''
    def poruszaj_sie(self, odleglosc):
        #...
    def daj_glos(self):
        #...

class Ptak(Zwierze):
    rozpietosc_skrzydel = 0 #[m]
    def lec(self, wysokosc):
        #...

zebra = Zwierze()
zebra.masa_ciala = 400
zebra.kolor = 'biało-czarny'
zebra.poruszaj_sie(15)
```

```
kanarek = Ptak()
kanarek.kolor = 'żółty'
kanarek.masa_ciala = 0.02
kanarek.rozpietosc_skrzydel = 0.2
kanarek.lec(wysokosc = 3)

kruk = Ptak()
kruk.kolor = 'czarny'
```

tworzenie obiektu klasy

Uwaga: Należy pamiętać o kolejności. Definicja klasy `Ptak` powinna być poniżej definicji klasy `Zwierze`. Tworzenie obiektów powinno być poniżej definicji obu klas.

Przykładowa klasa `Wiadro`

- Klasa reprezentująca wiadro o określonej pojemności. Do wiadra można dolewać płyn. Pojemność i ilość płynu są polami klasy, natomiast dolewanie jest metodą klasy.

```
class Wiadro:
    pojemnosc = 0 #[litr]
    ilosc_plynu = 0 #[litr]

    def dolej(self, ile):
        self.ilosc_plynu += ile
        if self.ilosc_plynu > self.pojemnosc:
            self.ilosc_plynu = self.pojemnosc
            print('Nie można dolać aż tyle płynu.')

    def wypisz_stan_wiadra(self):
        print('W wiadrze znajduje się', self.ilosc_plynu, 'l płynu.')
```

Tworzenie i korzystanie z obiektów klasy `Wiadro`

```
male_wiaderko = Wiadro()

male_wiaderko.pojemnosc = 5
male_wiaderko.ilosc_plynu = 1
male_wiaderko.wypisz_stan_wiadra()

male_wiaderko.dolej(3)
male_wiaderko.wypisz_stan_wiadra()

male_wiaderko.dolej(2)
male_wiaderko.wypisz_stan_wiadra()

duze_wiadro = Wiadro()
duze_wiadro.pojemnosc = 10
#...
```

Tworzymy małe wiaderko.

W wiadrze jest 1 litr płynu.

W wiadrze jest 4 litry płynu.

W wiadrze jest 5 litrów płynu; ostatniego litra nie udało się dolać, bo przekroczył pojemność wiadra.

Tworzymy duże wiadro (osobny obiekt).

Konstruktor

- Konstruktor jest specjalną metodą, która wykonuje się automatycznie kiedy tworzony jest obiekt klasy.

- Jeśli napiszemy konstruktor, będziemy mogli podawać wstępne wartości pól klasy w momencie tworzenia obiektu. Nie będzie więc potrzeby podawania ich w osobnych linijkach.

- Konstruktor w języku Python ma zawsze tę samą nazwę: `__init__`

  Uwaga: W nazwie konstruktora są dwa podkreślniki z przodu i dwa z tyłu.

Klasa `Wiadro` z konstruktorem

```
class Wiadro:
    pojemnosc = 0 #[litr]
    ilosc_plynu = 0 #[litr]

    def __init__(self, pojemnosc, ilosc_plynu):
        self.pojemnosc = pojemnosc
        self.ilosc_plynu = ilosc_plynu

    def dolej(self, ile):
        self.ilosc_plynu = self.ilosc_plynu + ile
        if self.ilosc_plynu > self.pojemnosc:
            self.ilosc_plynu = self.pojemnosc
            print('Nie można dolać aż tyle wody.')

    def wypisz_stan_wiadra(self):
        print('W wiadrze znajduje się', self.ilosc_plynu, 'l płynu.')
```

Te instrukcje nie są konieczne, jeśli mamy konstruktor i można je usunąć. Można jednak je zachować, aby wyraźnie widzieć, jakie pola występują w klasie (szczególnie jeśli ktoś przyzwyczaił się do składni języków: C++, C#, Java).

wstawiono konstruktor

Tworzenie i korzystanie z obiektów klasy `Wiadro`

```
male_wiaderko = Wiadro(5,1)

male_wiaderko.pojemnosc = 5
male_wiaderko.ilosc_plynu = 1
male_wiaderko.wypisz_stan_wiadra()
```

Dzięki konstruktorowi możemy podać wstępne wartości pól w jednej linijce.

Nie potrzebujemy podawać ich drugi raz.

Gdybyśmy jednak to zrobili, wartości podane podczas tworzenia obiektu zostałyby zamienione na nowe.

`WiadroZPokrywa` – klasa dziedzicząca po klasie `Wiadro`

```
class WiadroZPokrywa(Wiadro):
    zamkniete = False

    def __init__(self, pojemnosc, ilosc_plynu, zamkniete):
        self.pojemnosc = pojemnosc
        self.ilosc_plynu = ilosc_plynu
        self.zamkniete = zamkniete

    def dolej(self, ile):
        if self.zamkniete:
            print('Nie można dolać, bo wiadro jest zamknięte.')
            return
        self.ilosc_plynu += ile
        if self.ilosc_plynu > self.pojemnosc:
            self.ilosc_plynu = self.pojemnosc
            print('Nie można dolać aż tyle wody.')
```

dziedziczenie po klasie Wiadro

Dodajemy pole `zamkniete`, które określa, czy wiadro jest zamknięte, czy otwarte.

Nowe pole trzeba uwzględnić w konstruktorze.

Jeśli wiadro jest zamknięte, to nie dolejemy płynu. Funkcja natychmiast zakończy się napotykając słowo `return`.

`WiadroZPokrywa` – alternatywna, krótsza wersja

```
class WiadroZPokrywa(Wiadro):
    zamkniete = False

    def __init__(self, pojemnosc, ilosc_plynu, zamkniete):
        self.zamkniete = zamkniete
        super().__init__(pojemnosc, ilosc_plynu)

    def dolej(self, ile):
        if self.zamkniete:
            print('Nie można dolać, bo wiadro jest zamknięte.')
            return
        super().dolej(ile)
```

Funkcja `super` pozwala wywołać metody z nadklasy (klasy, po której dziedziczymy).

Dzięki temu kod jest krótszy, bo lepiej wykorzystujemy odziedziczone składniki klasy `Wiadro`.

- Jeśli piszemy metodę, która ma tę samą nazwę, co metoda w nadklasie, to nazywa się to przesłonieniem lub nadpisywaniem metod (ang. method override).
- W klasie `WiadroZPokrywa` przesłoniliśmy dwie metody: `dolej` i konstruktor.
- Metoda `wypisz_stan_wiadra` nie została przesłoniona, bo nie musieliśmy jej zmieniać. Należy jednak pamiętać, że została ona odziedziczona i możemy z niej korzystać.

Tworzenie i korzystanie z obiektów klasy `WiadroZPokrywa`

Tym razem są aż trzy parametry, więc dla czytelności kodu lepiej użyć parametry nazwane zamiast pozycyjnych.

```
wiaderko = WiadroZPokrywa(pojemnosc=5, ilosc_plynu=1, zamkniete=False)

wiaderko.dolej(1)
wiaderko.wypisz_stan_wiadra()
wiaderko.zamkniete = True
wiaderko.dolej(2) #błąd - wiaderko jest zamknięte, więc nie można dolać
wiaderko.wypisz_stan_wiadra()
```

- Uwaga: Należy pamiętać, że jeśli piszemy klasę dziedziczącą po innej klasie, to kod nadklasy powinien znajdować się powyżej klasy dziedziczącej.

- Analogicznie jest z funkcjami. Kiedy funkcja A korzysta z funkcji B (wywołuje ją), to kod funkcji B powinien być powyżej kodu funkcji A.

Funkcja `hasattr`

- Funkcja `hasattr` służy do sprawdzenia, czy klasa o podanej nazwie ma atrybut (metodę lub pole) o podanej nazwie.
- Poniższy program sprawdza, czy klasa `Budynek` posiada pola `ulica` i `miasto`. Sprawdza również, czy w klasie istnieje metoda `wypisz_adres`.

```
class Budynek:
    ulica = 'Akacjowa'
    numer = 123
    kod_pocztowy = '35-999'

    def wypisz_adres(self):
        print(f'{self.ulica} {self.numer}, {self.kod_pocztowy}')

print( hasattr(Budynek, 'ulica') )
print( hasattr(Budynek, 'miasto') )
print( hasattr(Budynek, 'wypisz_adres') )
```

Funkcja `isinstance`

- Funkcja `isinstance` służy do sprawdzenia, czy obiekt należy do klasy o podanej nazwie (czy jest instancją tej klasy).

```
class Budynek:
    ulica = 'Akacjowa'
    numer = 123
    kod_pocztowy = '35-999'

    def wypisz_adres(self):
        print(f'{self.ulica} {self.numer}, {self.kod_pocztowy}')

biuro = Budynek()
ulice = ['Akacjowa', 'Jodłowa', 'Dębowa']

print( isinstance(biuro, Budynek) ) #biuro to obiekt klasy Budynek
print( isinstance(ulice, Budynek) ) #ulice to nie obiekt klasy Budynek
print( isinstance(ulice, list) ) #ulice to lista, czyli obiekt klasy list
```

Sprawdzanie, czy obie zmienne reprezentują ten sam obiekt

- Operator `is` służy do sprawdzenia, czy zmienne reprezentują ten sam obiekt.
- Poniższy przykład prezentuje różnicę w działaniu operatorów `is` i `==`.

```
x = ['jabłko', 'gruszka']
y = ['jabłko', 'gruszka']
z = x
print(x is z) #True. x i z są tymi samymi obiektami
print(x is y) #False. x i y nie są tymi samymi obiektami, pomimo że mają tę samą
              #zawartość
print(x == y) #True. operator == sprawdza, czy obiekty mają tę samą zawartość (nie
              #muszą być tymi samymi obiektami)
#jeśli zmodyfikujemy obiekt z, to również zmodyfikuje się obiekt x (ponieważ są one
#tymi samymi obiektami)
z[1] = 'śliwka'
print(x)
```

Metody magiczne

- W obiektach języka Python istnieje wiele metod, które zaczynają się i kończą podwójnymi znakami podkreślenia.
- Do tej pory omawiana była tylko jedna z nich: `__init__`.
- Są to tzw. metody magiczne (ang. magic methods).
- Metody magiczne mają specjalne zadania. Służą głównie do integracji obiektów z wbudowanymi funkcjami i operatorami.

Metoda magiczna `__str__`

- Jedną z bardziej znanych metod magicznych jest `__str__`.
- Wywołuje się ona automatycznie wtedy, gdy próbujemy wypisać obiekt funkcją `print`.
- Możemy ją zdefiniować w klasie, żeby określić w jaki sposób obiekt ma być wypisywany.

```
class Pracownik:
    def __init__(self, imie, nazwisko, stanowisko):
        self.imie = imie
        self.nazwisko = nazwisko
        self.stanowisko = stanowisko

    def __str__(self):
        return (self.imie + ' ' + self.nazwisko +
                ' pracuje na stanowisku: ' + self.stanowisko)

jan = Pracownik('Jan','Nowak','programista')
print(jan)
```

Metody magiczne – przykłady

- Istnieją inne, podobne do `__str__` metody, które są wywoływane wtedy, gdy obiekt przekażemy odpowiednich funkcji:
  - `__round__` – wywołuje się po przekazaniu obiektu do funkcji `round` (zaokrąglanie);
  - `__floor__` – wywołuje się po przekazaniu obiektu do funkcji `math.floor` (zaokrąglanie w dół);
  - `__ceil__` – wywołuje się po przekazaniu obiektu do funkcji `math.ceil` (zaokrąglanie w górę);
  - `__trunc__` – wywołuje się po przekazaniu obiektu do funkcji `math.trunc` (obcięcie części ułamkowej);
  - `__abs__` – wywołuje się po przekazaniu obiektu do funkcji `abs` (wartość bezwzględna).

Metoda magiczna `__add__`

- Metoda `__add__` wywołuje się wtedy, gdy używamy operator + na obiektach.

```
class Wektor2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f'x:{self.x}, y:{self.y}'

    def __add__(self, other):
        return Wektor2D(self.x + other.x, self.y + other.y)

v1 = Wektor2D(2, 3)
v2 = Wektor2D(1, 4)

print(v1 + v2)
```

Tutaj wywoła się metoda `__add__`, która zwróci nowy obiekt `Wektor2D`. Następnie wywoła się metoda `__str__` i ten nowy obiekt zostanie wypisany.

Metody magiczne – przykłady

- Istnieją inne, podobne do `__add__` metody, które są wywoływane wtedy, gdy używamy operatory (arytmetyczne, porównania) na obiektach. Przykłady takich metod:
  - `__sub__` – wywołuje się po użyciu operatora – na obiektach;
  - `__mul__` – wywołuje się po użyciu operatora `*` na obiektach;
  - `__truediv__` – wywołuje się po użyciu operatora `/` na obiektach;
  - `__mod__` – wywołuje się po użyciu operatora `%` na obiektach;
  - `__pow__` – wywołuje się po użyciu operatora `**` na obiektach;
  - `__eq__` – wywołuje się po użyciu operatora == na obiektach;
  - `__ne__` – wywołuje się po użyciu operatora `!=` na obiektach;
  - `__lt__`, `__le__` – wywołują się po użyciu operatorów < i <= na obiektach;
  - `__gt__`, `__ge__` – wywołują się po użyciu operatorów > i >= na obiektach.

Metody magiczne

- Pełna lista metod magicznych znajduje się pod adresem: *https://www.geeksforgeeks.org/dunder-magic-methods-python*

Przydatne metody do przetwarzania tekstów – metody `upper`, `lower` i `swapcase`

- Za pomocą metody `upper` możemy zamienić wszystkie małe litery w tekście na duże.
- Metoda `lower` zamienia duże litery w tekście na małe.
- Metoda `swapcase` zamienia małe litery na duże i duże na małe.

```
tekst = 'AbEcAdŁo 123'
print(tekst.upper())
print(tekst.lower())
print(tekst.swapcase())
```

Przykładowe zastosowanie funkcji `lower` – sprawdzanie tekstów niezależne od wielkości liter

- Poniższy przykład sprawdza, czy użytkownik podał nazwę stanowiska „profesor”.
- Sprawdzenie jest niezależne od wielkości liter, tzn. warunek zostanie spełniony, jeśli nazwą stanowiska będzie: „profesor”, „Profesor”, „PROFESOR” itp.

```
stanowisko = input('Podaj stanowisko: ')

if stanowisko.lower() == 'profesor':
    print('Jest to profesor. ')
```

Przydatne funkcje do przetwarzania tekstów – metoda `find` i inne

- Sprawdza, czy w tekście występuje fragment podany jako parametr.
- Jeśli fragment występuje, to metoda zwraca jego pozycję (numer znaku licząc od 0).
- Jeśli fragment nie występuje, to metoda zwraca wartość -1.

```
zdanie = 'Ala ma kota.'
slowo_kot = zdanie.find('kot')
slowo_waran = zdanie.find('waran')

print(slowo_kot)
print(slowo_waran)
```

- Inne metody do przetwarzania tekstów (obiektów typu str) są wymienione i opisane tutaj: *https://www.w3schools.com/python/python_ref_string.asp*

`match-case` – alternatywa dla `if-elif-else`

- W wersji 3.10 języka Python wprowadzono alternatywę dla instrukcji `if-elif-else`. Jest to `match-case`.

- Można rozważyć stosowanie `match-case` w przypadku, gdy mamy wiele warunków zawierających operator == (sprawdzenie, czy zmienna przyjmuje jakąś konkretną wartość).

- Trzeba jednak pamiętać, że w wersjach Pythona poniżej 3.10 instrukcja `match-case` nie jest dostępna.

`match-case` – przykład i porównanie z `if-elif-else`

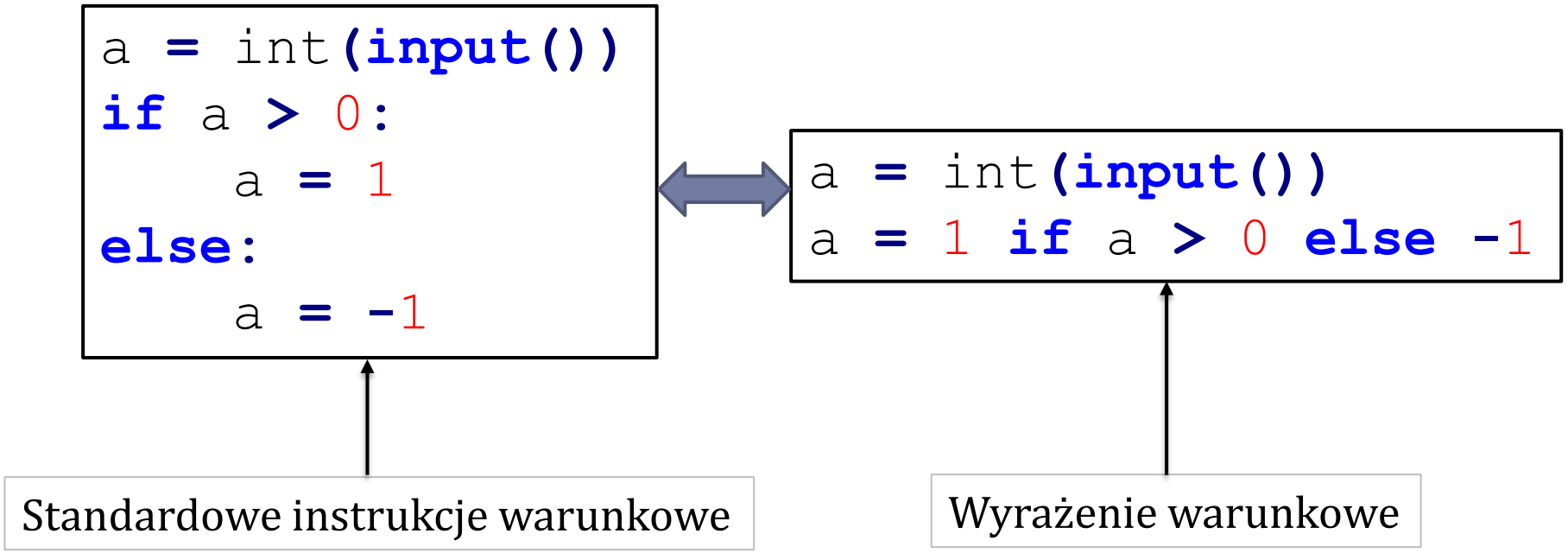
- Poniższe przykłady są równoważne.

```
liczba = int(input('Podaj liczbę z zakresu [1-5]:'))
if liczba == 1:
    print('I')
elif liczba == 2:
    print('II')
elif liczba == 3:
    print('III')
elif liczba == 4:
    print('IV')
elif liczba == 5:
    print('V')
else:
    print('nieznana')
```

```
liczba = int(input('Podaj liczbę z zakresu [1-5]:'))
match liczba:
    case 1:
        print('I')
    case 2:
        print('II')
    case 3:
        print('III')
    case 4:
        print('IV')
    case 5:
        print('V')
    case _:
        print('nieznana')
```

Wyrażenia warunkowe – alternatywna dla standardowych instrukcji warunkowych

- Jeśli warunki `if-else` decydują o tym jaką wartość przypisać do zmiennej, to możemy rozważyć użycie zapisu skrótowego w formie tzw. wyrażeń warunkowych:



Funkcje wyższego rzędu

- Funkcją wyższego rzędu jest funkcja, której co najmniej jeden parametr jest inną funkcją lub która zwraca inną funkcję jako wartość.

```
# funkcja wyższego rzędu
def funkcjaWR(paramFunkcja, liczba):
    return paramFunkcja(liczba)

# dwie zwykłe funkcje
def inkrementacja(x):
    return x + 1

def pierwiastek_kwadratowy(x):
    return x ** 0.5

# wywołanie funkcji wyższego rzędu
wynik1 = funkcjaWR(inkrementacja, 5)
wynik2 = funkcjaWR(pierwiastek_kwadratowy, 16)
print(wynik1, wynik2)
```

parametr będący funkcją

parametr będący zmienną

Funkcje lambda

- Lambda są anonimowymi (nie posiadającymi nazwy), jednolinijkowymi funkcjami.

- Funkcje lambda mają następującą składnię:

```
lambda <parametry> : <wartość_zwracana>
```

Funkcje lambda

- Zauważmy, że w przykładzie z dwóch slajdów wstecz funkcje `inkrementacja` i `pierwiastek_kwadratowy` zostały napisane tylko po to, aby przekazać je jako parametr funkcji wyższego rzędu. Prawdopodobnie nie zostaną więcej użyte w programie.
- Możemy więc zamienić funkcje `inkrementacja` i `pierwiastek_kwadratowy` na funkcje lambda, których definicje wstawimy bezpośrednio do wywołania funkcji wyższego rzędu. Taki zapis jest znacznie krótszy.

```
# funkcja wyższego rzędu
def funkcjaWR(paramFunkcja,liczba):
    return paramFunkcja(liczba)

# wywołanie funkcji wyższego rzędu
wynik1 = funkcjaWR(lambda x : x+1, 5)
wynik2 = funkcjaWR(lambda x : x**0.5, 16)
print(wynik1, wynik2)
```

funkcje lambda

Przykład praktycznego wykorzystania funkcji lambda

- Funkcja `sort`, która była omawiana na slajdzie „Listy – przydatne triki i funkcje” służy do sortowania listy.
- Załóżmy jednak, że elementami naszej listy nie są pojedyncze liczby, a pary (w formie list), których pierwszym elementem jest napis, a drugim liczba:

```
lista = [['c',3],['b',1],['a',2]]
```

- Jak posortować taką listę względem drugiego elementu par? Funkcja `sort` posiada parametr `key`, który określa kryterium sortowania.
- Kryterium sortowania jest funkcją (nie zmienną), a więc możemy podać własną tymczasową funkcję lambda zwracającą drugi element par listy (jako kryterium sortowania).

```
lista.sort(key = lambda el : el[1])
print(lista) #co zostanie wypisane?
```

Zadanie 12

- Zadanie: Jak zamienić poniższą funkcję na funkcję lambda?

```
def ograniczenieDoStu(x):
    if x > 100:
        return 100
    else:
        return x
```

Podpowiedź: skorzystać z wyrażenia warunkowego.

Operacje odczytu i zapisu plików – otwieranie pliku

- Aby wykonywać operacje na plikach należy skorzystać z funkcji `open`, która otwiera plik do odczytu lub zapisu (jeśli plik nie istnieje, funkcja go tworzy):

```
plik_wejsciowy = open('plik1.txt', 'r')
plik_wyjsciowy = open('plik2.txt', 'w')
```

- Pierwszy parametr jest nazwą pliku.
  - Jeśli plik mieści się w tym samym katalogu (folderze) co program, wystarczy podać samą nazwę (np. 'plik1.txt').
  - Jeśli plik mieści się w innym katalogu niż program, należy podać pełną ścieżkę do pliku (np. 'c:/pliki/plik1.txt') lub ścieżkę względną, prowadzącą od katalogu, w którym znajduje się program (np. 'pliki/plik1.txt').
- Za pomocą drugiego parametru określamy, czy plik będzie otwierany w celu odczytywania z niego danych (`'r'`), czy zapisywania do niego danych (`'w'`).
- Można również otworzyć plik z opcją `'a'` - dopisywanie danych na końcu pliku, bez kasowania jego poprzedniej zawartości.

Operacje odczytu i zapisu plików – zamykanie pliku

- Po wykonaniu wszystkich planowanych operacji należy zamknąć plik.
- Służy do tego metoda `close`.

```
plik_wejsciowy.close()
plik_wyjsciowy.close()
```

- *Zamknięcie pliku spowoduje natychmiastowe zatwierdzenie zmian i zwolni tzw. uchwyt pliku.*
- *Zwalnianie uchwytów jest ważne szczególnie wtedy, gdy program używa wielu plików lub jeden plik jest używany przez wiele programów.*

Czytanie z pliku

- Do odczytywania danych z pliku (i zapisywania ich do zmiennej) służą metody `read` i `readline`.

```
d = plik_wejsciowy.read() #odczytuje wszystkie pozostałe (niewczytane wcześniej) dane
d2 = plik_wejsciowy.read(5) #odczytuje 5 kolejnych znaków pliku tekstowego
d3 = plik_wejsciowy.readline() #odczytuje kolejną linię tekstu (aż do znaku przejścia
                               #do nowej linii, który również zostaje wczytany)
```

- Dane odczytywane są jako tekst. Jeśli chcemy wczytać liczbę lub wartość innego typu, należy ten tekst przekonwertować.

```
wczytany_tekst = plik_wejsciowy.readline()
wczytana_liczba = int(wczytany_tekst)
```

- Jeśli po osiągnięciu końca pliku wywołamy funkcję `read` lub `readline`, to zwróci ona pusty łańcuch znakowy `''`, ponieważ nie będzie więcej danych do odczytania.

Czytanie z pliku przy użyciu pętli `for`

- Istnieje sposób na odczytanie wszystkich linii tekstu za pomocą pętli `for`.

```
plik_wejsciowy = open('plik1.txt', 'r')

for linia in plik_wejsciowy:
    liczba = float(linia)
    print(liczba)

plik_wejsciowy.close()
```

Zmienna `linia` reprezentuje linie wczytane w kolejnych iteracjach pętli.

Zapis do pliku

- Do zapisywania danych do pliku służą funkcje `write` i `writelines`.

```
plik_wyjsciowy.write('tekst, który zostanie zapisany w pliku') #zapisuje tekst
                                                               #podany jako parametr
plik_wyjsciowy.writelines(['linia I', 'linia II', 'linia III']) #zapisuje listę
                                                                #linijek tekstu
```

- Zapisywane dane muszą być w formie tekstowej. Jeśli chcemy zapisać liczbę lub wartość innego typu, należy ją przekonwertować na tekst (typ str).

```
liczba = 13
plik_wyjsciowy.write(str(liczba))
```

Korzystanie z pliku przy użyciu wyrażenia `with`

- Wyrażenie `with` pozwala na korzystanie z pliku bez konieczności jego ręcznego zamykania. Otwarte w ten sposób pliki zamykają się automatycznie kiedy skończy się blok `with`, więc nie musimy wywoływać metody `close`.

```
with open('plik.txt', 'r') as plik_wejsciowy:
    pierwsza_linia = pik_wejsciowy.readline()
    pozostale_dane = plik_wejsciowy.read()
```

Inne przydatne metody związane z operacjami na plikach

- Metoda `tell` zwraca aktualną pozycję uchwytu do pliku.
- Metoda `seek` pozwala ustawić aktualną pozycję uchwytu do pliku.
- Czym jest pozycja uchwytu do pliku?

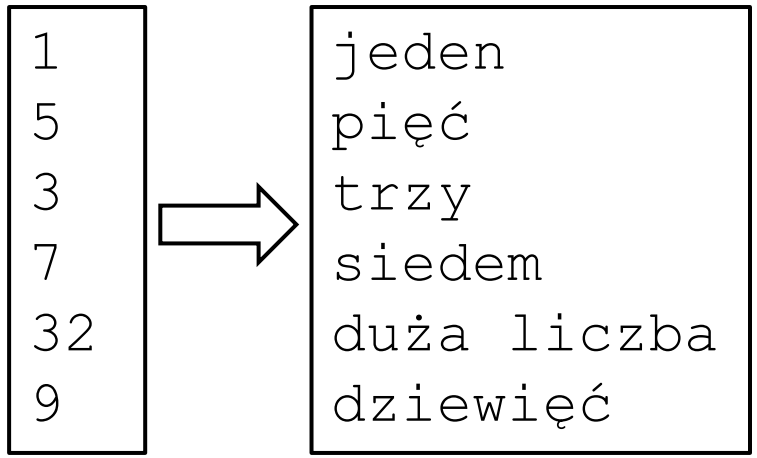
  Jest to miejsce:
  - z którego odczytywane będą dane przy użyciu następnej funkcji odczytu (`read`, `readline`);
  - w które zapisywane będą dane przy użyciu następnej funkcji zapisu (`write`, `writelines`).

```
pozycja = plik.tell() #zwróć aktualną pozycję uchwytu
plik.seek(pozycja+3) #przesuń aktualną pozycję uchwytu o 3 znaki
```

Zadanie 13

- Napisać program, który odczyta plik „plik_we.txt” zawierający liczby jednocyfrowe zapisane w osobnych liniach. Następnie program powinien zamienić wczytane liczby na ich słowne reprezentacje (np. 1 na „jeden”) i zapisać je do pliku „plik_wy.txt”.

- Przykładowa zawartość pliku wejściowego i wyjściowego:



Zadanie 13

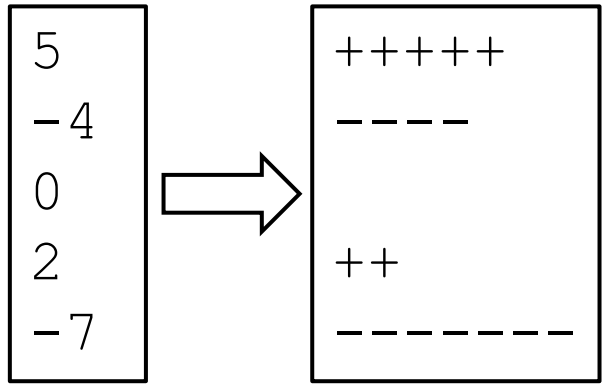
- Napisać program, który odczyta plik „plik_we.txt” zawierający liczby jednocyfrowe zapisane w osobnych liniach. Następnie program powinien zamienić wczytane liczby na ich słowne reprezentacje (np. 1 na „jeden”) i zapisać je do pliku „plik_wy.txt”.

```
with open('plik_we.txt', 'r') as plikWe, open('plik_wy.txt', 'w') as plikWy:
    for linia in plikWe:
        liczba = int(linia)
        if liczba == 0:
            plikWy.write('zero\n')
        elif liczba == 1:
            plikWy.write('jeden\n')
        #...
        elif liczba == 9:
            plikWy.write('dziewięć\n')
        else:
            plikWy.write('duża liczba\n')
```

Zadanie 14

- Napisać program, który odczyta plik „plik_we.txt” zawierający liczby całkowite. Następnie program powinien zapisywać w pliku „plik_wy.txt” w każdej linijce *n* znaków plusa, gdzie *n* jest liczbą odczytaną z odpowiedniej linii pliku "plik_we.txt". Jeśli wczytana liczba *n* jest ujemna, wtedy zamiast *n* plusów należy wpisać *n* minusów.

- Przykładowa zawartość pliku wejściowego i wyjściowego:



Wyjątki

- Wyjątki są mechanizmem pozwalającym programowi w wygodny sposób obsłużyć sytuacje wyjątkowe (np. błędy).
- Spróbujmy uruchomić następujący kod:

```
a = 2
b = 0
a = a/b
a = a+1
print(a)
```

- W konsoli pojawi się opis błędu:

```
File "D:/…/main.py", line 3, in <module>
    a = a/b
        ~^~
ZeroDivisionError: division by zero
```

- Oznacza to, że program automatycznie zgłosił wyjątek typu `ZeroDivisionError`, ze względu na próbę podzielenia liczby przez 0.

Wyjątki

- Wyjątek `ZeroDivisionError` został zgłoszony (automatycznie), ale nie został obsłużony.
- Oznacza to, że program zakończył działanie w momencie zgłoszenia wyjątku.
- Zgłoszony wyjątek możemy obsłużyć.
- Pozwoli nam to:
  - kontynuować program pomimo nieudanej operacji (np. dzielenia przez 0),
  - wypisać na konsoli własny opis błędu.

Obsługa (przechwycenie) wyjątków

- Aby obsłużyć wyjątek należy skorzystać z bloków `try` i `except`.

```
a = 2
b = 0
try:
    a = a/b
    a = a+1
except ZeroDivisionError:
    print('Nie można dzielić przez 0.')
print(a)
```

- W bloku `try` umieszczamy kod, który może powodować zgłoszenie wyjątku.
- Po jego zgłoszeniu pozostały kod aż do końca bloku nie zostanie wykonany i program przejdzie do bloku `except`.
- Kod następujący po bloku `except` zostanie wykonany niezależnie od tego, czy wyjątek został zgłoszony, czy nie (kontynuacja programu pomimo błędu).

Często spotykane typy wyjątków

- `ValueError` – nieodpowiednia wartość danych dla wykonywanej operacji (np. próba zamiany tekstu nieskładającego się z cyfr na liczbę)
- `IndexError` – odwołanie się do nieistniejącego elementu listy
- `KeyError` – odwołanie się do nieistniejącego elementu słownika
- `NameError` – użycie nieistniejącej zmiennej, funkcji lub klasy
- `AttributeError` – odwołanie się do nieistniejącego atrybutu (metody lub pola) obiektu
- `TypeError` – mieszanie niezgodnych typów danych (np. próba podzielenia tekstu przez liczbę)
- `IndentationError` – niepoprawne wcięcia (np. mieszanie spacji i tabulacji)
- `SyntaxError` – inne błędy składniowe
- `MemoryError` – wyczerpanie miejsca w pamięci operacyjnej
- Listę wszystkich wbudowanych wyjątków języka Python można znaleźć na stronie: *https://docs.python.org/3/library/exceptions.html*

Blok `finally`

- Rozważmy poniższy przykład:

```
try:
    wynik = 10 / 0
except TypeError:
    print('Nie można dzielić przez zero.')

print('Ten tekst się nie pojawi.')
```

Zgłoszenie wyjątku `ZeroDivisionError`

Obsługa wyjątku `ValueError`. `ZeroDivisionError` nie jest obsługiwany.

Ten tekst się nie pojawi, ponieważ jeśli zgłoszony wyjątek nie zostanie obsłużony, to program natychmiast się przerywa.

Można się jednak przed tym zabezpieczyć używając bloku `finally` (więcej informacji na kolejnym slajdzie).

Blok `finally`

- Zmieńmy przykład z poprzedniego slajdu, aby uwzględniał blok `finally`:

```
try:
    wynik = 10 / 0
except TypeError:
    print('Nie można dzielić przez zero ')
finally:
    print('Ten tekst pojawi się zawsze.')
```

Ten tekst pojawi się zawsze, niezależnie od tego, czy w bloku `try` został zgłoszony wyjątek i czy został on obsłużony w bloku `except`.

Oczywiście brak obsługi wyjątku zakończy program, ale przed jego zakończeniem wykona się jeszcze kod z bloky `finally`.

Kiedy stosować blok `finally`?

- Blok `finally` możemy stosować np. wtedy, gdy są jakieś ważne operacje, które powinny się wykonać zawsze przed zamknięciem programu, np. zrobienie zrzutu ekranu, wykonanie kopii obiektów na dysku (serializacja).

- Jeśli te operacje umieścimy w bloku `finally`, to wykonają się **zawsze**, tzn. nawet wtedy, gdy wystąpi jakieś nieprzewidziane zdarzenie (nieobsłużony wyjątek).

- Warto wiedzieć, że blok `finally` wykona się nawet wtedy, gdy w bloku `try` pojawi się instrukcja wyjścia z funkcji/pętli (`return`, `break`, `continue`).

Samodzielne (nieautomatyczne) zgłaszanie wyjątków

- Wyjątki możemy zgłaszać samodzielnie, w sytuacjach, które sami uznamy za wyjątkowe. Wyjątek zgłaszamy instrukcją `raise`.
- Poniższy program wczytuje w pętli dwie jednocyfrowe liczby całkowite i wywołuje funkcję obliczającą ich sumę. Jeśli użytkownik podał liczbę, która nie jest jednocyfrowa, zostanie zgłoszony wyjątek `Exception` i pętla się przerwie.

```
def sumaLiczbJednocyfrowych(a,b):
    if a < -9 or a > 9 or b < -9 or b > 9:
        raise Exception
    return a+b
while True:
    try:
        a = int(input('Podaj jednocyfrową liczbę całkowitą nr 1:'))
        b = int(input('Podaj jednocyfrową liczbę całkowitą nr 2:'))
        suma = sumaLiczbJednocyfrowych(a,b)
        print(suma)
    except Exception:
        print('Błąd wczytywania liczb.')
```

Ta instrukcja nie wykona się w przypadku zgłoszenia wyjątku.

Samodzielne (nieautomatyczne) zgłaszanie wyjątków

- Istnieje możliwość opisania wyjątku podczas jego zgłaszania.
- Taki opis można wyświetlić podczas przechwytywania wyjątku w bloku `except`.
- W tym celu należy skorzystać z obiektu reprezentującego błąd. Składnia:

```
except <typ_wyjątku> as <obiekt_wyjątku>:
```

```
def sumaLiczbJednocyfrowych(a,b):
    if a < -9 or a > 9 or b < -9 or b > 9:
        raise Exception('Podano liczbę spoza wymaganego przedziału.')
    return a+b
while True:
    try:
        a = int(input('Podaj jednocyfrową liczbę całkowitą nr 1: '))
        b = int(input('Podaj jednocyfrową liczbę całkowitą nr 2: '))
        suma = sumaLiczbJednocyfrowych(a,b)
        print(suma)
    except Exception as error:
        print('Błąd wczytywania liczb:', error)
```

opis wyjątku

obiekt przechwyconego wyjątku

Czy w Pythonie można zgłaszać wyjątki dowolnego typu (dowolnej klasy)?

- W języku Python można zgłaszać i obsługiwać wyjątki dowolnej klasy pod warunkiem, że dziedziczy ona po klasie `BaseException`.

- Klasa `Exception` jest podklasą `BaseException`, więc po niej również można dziedziczyć (jest to zalecane).

Zadanie 15

- Zmodyfikować program z zad. 14 tak, aby obsługiwał wyjątki związane z:
  - niepoprawną nazwą pliku wejściowego (`FileNotFoundError`),
  - wczytaniem linii tekstu, które nie są liczbami (`ValueError`).

Treść zadania 14 (dla przypomnienia):

- Napisać program, który odczyta plik „plik_we.txt” zawierający liczby całkowite. Następnie program powinien zapisywać w pliku „plik_wy.txt” w każdej linijce *n* znaków plusa, gdzie *n* jest liczbą odczytaną z odpowiedniej linii pliku "plik_we.txt". Jeśli wczytana liczba *n* jest ujemna, wtedy zamiast *n* plusów należy wpisać *n* minusów.

Moduły

- Z modułów korzystamy wtedy, gdy chcemy uzyskać dostęp do funkcjonalności wykraczającej poza standardowe funkcje dostępne w Pythonie.

- Moduły są zbiorami:
  - funkcji,
  - klas i obiektów,
  - stałych (specjalnych zmiennych, które mają ustalone z góry wartości).

- Moduły dołączamy (importujemy) do programu wyrażeniem:
  `import <nazwa modułu>`

- Jednym z najbardziej popularnych i podstawowych modułów jest moduł matematyczny `math`.

Moduł matematyczny `math`

- Moduł ten zawiera:
  - funkcje z zakresu trygonometrii, geometrii, kombinatoryki i ogólne funkcje matematyczne, takie jak: logarytm (`math.log`), największy wspólny dzielnik (`math.gcd`), wartość bezwzględna (`math.fabs`) itp.;
  - stałe matematyczne, takie jak: π (`math.pi`), e (`math.e`), nieskończoność (`math.inf`).
- Poniższy kod korzysta z funkcji `math.cos` i stałej `math.pi` aby obliczyć kosinus z liczby pi.

```
import math

kosinus = math.cos(math.pi)
print(kosinus)
```

Moduł matematyczny `math`

- Funkcja `math.comb(k,n)` oblicza liczbę *n*-elementowych kombinacji bez powtórzeń ze zbioru *k*-elementowego.

- Poniższy kod korzysta z funkcji `math.comb` do obliczenia prawdopodobieństwa trafienia „szóstki” w „Dużym Lotku”.

```
import math

kombinacje = math.comb(49,6)
print(f'szansa na trafienie szóstki, to 1 do {kombinacje}.')
```

- Uwaga: Funkcja `math.comb` jest dostępna w Pythonie od wersji 3.8.

Drugi sposób importowania modułów

- Modułowi można nadać własną skrótową nazwę za pomocą konstrukcji:

```
import <nazwa modułu> as <skrót>
```

```
import math as m

kombinacje = m.comb(49,6)
print(f'szansa na trafienie szóstki, to 1 do {kombinacje}.')
```

Zamiast nazwy modułu `math` można teraz podawać jego skrótowa nazwę `m`.

Trzeci sposób importowania modułów

- Jeśli chcemy korzystać ze składników modułu bez podawania jego nazwy i kropki z lewej strony nazwy składnika, to możemy użyć konstrukcji:
  `from <nazwa modułu> import <nazwy składników>`

- W ten sposób importujemy tylko konkretne składniki (oddzielone przecinkami), a nie cały moduł.

Trzeci sposób importowania modułów

- Poniższy program importuje dwie funkcje z modułu `math`:
  - `tan` – funkcja obliczająca tangens wartości podanej w radianach
  - `radians` – zamiana stopni na radiany
- Dzięki temu możemy obliczyć tangens z wartości kąta podanego w stopniach. Program automatycznie (funkcją `radians`) zamieni tą wartość na radiany.

Nie musimy podawać nazwy modułu `math`, ponieważ zaimportowaliśmy dwie konkretne funkcje zamiast całego modułu.

```
from math import radians, tan

stopnie = float(input('Określ kąt (w stopniach): '))
radiany = radians(stopnie)
tangens = tan(radiany)
print(tangens)
```

Czwarty sposób importowania modułów

- Używając składni:

```
from <nazwa modułu> import *
```

po słowie `import` zamiast nazw składników można wpisać symbol `*`.
Spowoduje to zaimportowanie wszystkich składników modułu, jednak nie będziemy musieli podawać jego nazwy przed użyciem składników.

W tym wypadku również nie musimy podawać nazwy modułu `math`.

```
from math import *

stopnie = float(input('Określ kąt (w stopniach): '))
radiany = radians(stopnie)
tangens = tan(radiany)
print(tangens)
```

Kiedy importować cały moduł, a kiedy jego konkretne składniki?

- Najwygodniej jest używać sposobów importujących cały moduł, bez podawania konkretnych składników.
- Jest to jednak zalecane jedynie w przypadku niewielkich programów, w których używamy jeden lub niewiele modułów.
- Jeśli piszemy duży, rozbudowany program, w którym używamy wiele modułów, najlepiej importować tylko te składniki, których używamy (szczególnie wtedy, gdy korzystamy z niewielu składników danego modułu).
- Pozwoli to uniknąć konfliktu nazw (funkcji, klas, obiektów) występujących w różnych modułach.
- Jeśli mielibyśmy taki konflikt, to składniki z drugiego zaimportowanego modułu „przykryją” składniki o tej samej nazwie z pierwszego modułu.

Moduł generowania liczb pseudolosowych `random`

- Liczba pseudolosowa – liczba pozornie losowa, która została wygenerowana na podstawie jakiejś informacji.
- Najczęściej do generowania liczb pseudolosowych za pomocą komputera wykorzystuje się informacje pobrane z zegara systemowego (bity odpowiadające: dacie, godzinie, sekundzie, milisekundzie, mikrosekundzie).

```
import random

a = random.randint(10,20) #losowanie liczby całkowitej z przedziału [10,20]
b = random.uniform(0,2) #losowanie liczby zmiennoprzecinkowej z przedziału [0,2]
print(a)
print(b)

karty = ['as', 'król', 'dama', 'walet']
random.shuffle(karty) #losowa zmiana kolejności elementów listy (tasowanie kart)
print(karty)
```

Moduły `math` i `random` – inne funkcje

- Listę wszystkich funkcji (wraz z opisami i przykładami użycia) dostępnych w modułach `math` i `random` można znaleźć na stronach:
  - *https://www.w3schools.com/python/module_math.asp*
  - *https://www.w3schools.com/python/module_random.asp*

Moduł `pickle` do serializacji obiektów

- Jeśli chcemy w prosty sposób zapisać dane dowolnego obiektu do pliku, możemy skorzystać z modułu `pickle`.
- Taki zapis obiektów (w tym zmiennych, kolekcji itp.) nazywamy serializacją.
- Do zapisu obiektów służy funkcja `dump`. Zapisane w ten sposób dane możemy odczytać funkcją `load`.

Moduł `pickle` do serializacji obiektów

Zapis do pliku

```
import pickle

ulubione = {'owoce':['malina','borówka'], 'warzywa':['kukurydza','papryka']}

with open('ulubione.dat','wb') as plik:
    pickle.dump(ulubione, plik)
```

Odczyt z pliku

```
import pickle

with open('ulubione.dat','rb') as plik:
    ulubione = pickle.load(plik)

print(ulubione)
```

Korzystając z modułu `pickle` należy otwierać pliki z opcją zapisu/odczytu binarnego (litera b). Domyślnie ustawiany jest tryb tekstowy.

Moduł `NumPy`

- `NumPy`, jest to moduł matematyczny bardziej zaawansowany niż `math`, który służy przede wszystkim do operacji macierzowych.

- Jego główną zaletą jest znacznie większa szybkość wykonywania operacji macierzowych niż w przypadku standardowych list.

- Powinniśmy rozważyć korzystanie z macierzy modułu `NumPy` zamiast list wtedy, gdy przetwarzamy duże ilości danych lub gdy chcemy wykonać matematyczne operacje macierzowe.

Moduł NumPy – podstawowa funkcjonalność

```
import numpy as np

macierz_1 = np.array([[1, 2, 3], [4, 5, 6]]) #tworzenie macierzy
macierz_2 = np.array([[1, 2], [3, 4], [5, 6]])
iloczynMacierzy = macierz_1 @ macierz_2 #operator @ służy do mnożenia macierzy
macierzTransponowana = macierz_1.transpose() #tranzpozycja macierzy

print('macierz_1:\n', macierz_1)
print('macierz_2:\n', macierz_2)
print('iloczynMacierzy:\n', iloczynMacierzy)
print('macierz_1Transponowana:\n', macierzTransponowana)

#wypisanie elementu z wiersza 1 i kolumny 2
print( macierz_1[1][2] )  #standardowy sposób
print( macierz_1[1,2] )   #wygodniejszy sposób, typowy dla numpy
```

Moduł NumPy – wstawianie nowego elementu na koniec macierzy

```
import numpy as np

macierz = np.array([1, 3, 5])
macierz = np.append(macierz, 7)

print(macierz)
```

Funkcja append działa podobnie jak metoda append dla list. Należy jednak pamiętać, że pracuje ona na kopii podanej macierzy, a więc jej wynik trzeba na końcu do tej macierzy przypisać .

Moduł NumPy – wybieranie pojedynczych wierszy/kolumn

```
import numpy as np

macierz = np.array([[1, 2, 3],
                    [4, 5, 6],
                    [7, 8, 9]]) #tworzenie macierzy

print( macierz[0,:] ) #wypisanie całego pierwszego wiersza
print( macierz[:,0] ) #wypisanie całej pierwszej kolumny
macierz[:,0] = [0, 0, 0] #wyzerowanie pierwszej kolumny
print(macierz)
```

Moduł NumPy – odczyt macierzy z plików tekstowych i zapis do plików

- Macierze NumPy możemy odczytać z pliku tekstowego lub zapisać do pliku tekstowego przy użyciu funkcji loadtxt i savetxt.
- Pierwszy parametr obu funkcji oznacza nazwę pliku (z opcjonalną ścieżką).
- Drugi parametr funkcji savetxt oznacza zapisywaną macierz, natomiast parametr fmt określa format zapisu ('%i' oznacza liczby całkowite).

```
import numpy as np

macierz = np.loadtxt('plik_we.txt')
np.savetxt('plik_wy.txt', macierz.transpose(),fmt='%i')
```

- Funkcja loadtxt ma również inne przydatne parametry, np. określające które kolumny i wiersze czytać, a które pomijać (usecols, skiprows), ustawiające znak separatora oddzielającego wartości (delimiter) i inne.

Moduł `time`

- Moduł `time` umożliwia zmierzenie czasu wykonywania obliczeń w programie.
- Funkcja `time` zwraca liczbę sekund, które upłynęły od dnia 01.01.1970, godz. 00:00 (UTC – Coordinated Universal Time).
- Jeśli wywołamy funkcję `time` dwa razy i obliczymy różnicę między drugim pomiarem a pierwszym, uzyskamy czas (w sekundach), który upłynął pomiędzy pomiarami.

```
import time

start = time.time()
for i in range(20000000): #20 milionów obiegów
  j = i + 1
stop = time.time()

print('upłynęło', stop-start, 'sekund')
print('upłynęło', (stop-start)*1000, 'milisekund')
```

Moduł `matplotlib`

- Moduł `matplotlib` umożliwia tworzenie wykresów.

- Poniższy program tworzy wykres liniowy (funkcja `plot`) oraz punktowy (funkcja `scatter`).

```
import matplotlib.pyplot as plt

y = [100,91,79,60,60,58,62,65,70,70,75,76,78,79,93,99,99,100]
x = range(len(y)) #[0, 1, 2, 3...]

plt.scatter(x, y)
plt.plot(y)
plt.show()
```

Moduł `matplotlib`

- Funkcja `hist` umożliwia tworzenie histogramów. Oprócz danych, należy jej podać liczbę słupków (ang. bins).

```
import matplotlib.pyplot as plt

y = [1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 7]
bins = len(set(y))

plt.hist(y, bins)
plt.show()
```

Moduł `pandas`

- Moduł `pandas` umożliwia przetwarzanie danych tabelarycznych, np. arkuszy kalkulacyjnych CSV, XLSX (Microsoft Excel, Libre Office Calc), pliki z danymi w formacie JSON.
- Przykładowy program z następnego slajdu wczytuje plik arkusza kalkulacyjnego o takiej zawartości:

produkt	ilość	cena
mleko	4	3.99
bułka	6	0.9
chleb	2	2
woda mineralna	8	2.99

- Następnie dodawana jest kolumna z łącznymi cenami każdego produktu (uwzględniającymi ilości produktów) i obliczona zostaje cena całkowita.

Moduł pandas – przetwarzanie arkuszy kalkulacyjnych

```
import pandas as pd

# wczytanie danych z pliku CSV
dane = pd.read_excel('arkusz.xlsx')
# podgląd danych
print(dane.head())

# dodanie nowej kolumny 'łącznie', czyli iloczynu ilości i cen
dane['łącznie'] = dane['cena'] * dane['ilość']

# wyświetlenie wyników - wybranie jedynie kolumn 'produkt' i 'łącznie'
print('\nCeny łączne:\n', dane.head()[['produkt', 'łącznie']])
# wyświetlenie ceny końcowej (za wszystkie produkty)
print('\nCena końcowa: ', dane['łącznie'].sum())

# zapis danych z dodaną nową kolumną do pliku wyjściowego
with pd.ExcelWriter('arkusz2.xlsx') as arkusz_wy:
    dane.to_excel(arkusz_wy)
```

Dane zostaną zapisane do obiektu `DataFrame`. Można go skojarzyć ze słownikiem z elementami o indeksach „produkt”, „ilość” i „cena”. Każdy element tego słownika będzie listą wartości występujących w danej kolumnie.

Moduł `OpenCV`

`OpenCV` jest modułem wykorzystywanym w problemach wizji komputerowej (widzenia komputerowego). Zapewnia następującą funkcjonalność:

- odczyt i zapis obrazów w różnych formatach (JPEG, PNG, BMP itp.);
- zmiana rozmiaru, obracanie, przycinanie i odwracanie obrazów;
- konwersje przestrzeni kolorów, np. z RGB do odcieni szarości lub HSV;
- filtracja, np. rozmycie, wyostrzenie;
- wykrywanie krawędzi, konturów i obiektów;
- segmentacja obrazu (oddzielenie ich części);
- analiza histogramów rozkładu jasności w obrazie (np. w celu poprawy kontrastu);
- detekcja i śledzenie obiektów; wykrywanie twarzy, oczu itp.;
- rozpoznawanie obiektów (zintegrowane z modelami uczenia maszynowego);
- detekcja punktów kluczowych i ich dopasowywanie;
- przetwarzanie plików wideo lub strumienia wideo (z kamery) w czasie rzeczywistym;
- odczyt i zapis plików wideo w różnych formatach;
- detekcja ruchu.

Moduł `OpenCV`

- Zaprezentowane zostaną dwa przykłady użycia `OpenCV`. Program z pierwszego przykładu wykrywa krawędzie obrazka, odwraca ich kolory i obraca wynikowy obraz o 180 stopni.
- Wykorzystujemy następujący obrazek:

Moduł OpenCV – wykrywanie krawędzi, odwrócenie kolorów, obrócenie obrazu

```
import cv2 #importowanie modułu OpenCV

obrazek = cv2.imread('hand.png', cv2.IMREAD_GRAYSCALE) #wczytanie obrazka hand.jpg

krawedzie = cv2.Canny(obrazek,50,150) #wykrycie krawędzi (50 i 150 to progi wykrywania)
cv2.imshow('krawedzie', krawedzie) #wyświetlenie krawędzi

#odwrócenie kolorów (odwołując się do poszczególnych pikseli)
for i, wiersz in enumerate(krawedzie):
    for j, piksel in enumerate(wiersz):
        if krawedzie[i][j] == 0:
            krawedzie[i][j] = 255
        else:
            krawedzie[i][j] = 0

krawedzie = cv2.rotate(krawedzie, cv2.ROTATE_180) #obrót o 180 stopni

#wyświetlenie obróconych krawędzi ze zmienionymi kolorami i "przytrzymanie" okna
cv2.imshow('krawedzie - negatyw', krawedzie)
cv2.waitKey(0)
```

Moduł OpenCV – wykrywanie krawędzi w strumieniu wideo

- Poniższy kod pobiera strumień wideo z kamery komputera w formie klatek (obrazów), wyznacza krawędzie i wyświetla obraz w czasie rzeczywistym.

```
import cv2

kamera = cv2.VideoCapture(0) #utworzenie obiektu kamery
klawisz = None

#wykonuj pętlę dopóki użytkownik nie wciśnie klawisza Esc (kod 27)
while klawisz != 27:
    ret, klatka = kamera.read() #pobranie klatki obrazu z kamery
    krawedzie = cv2.Canny(klatka,50,150) #wykrycie krawędzi
    cv2.imshow('obraz z kamery', krawedzie) #wyświetlenie obrazu krawędzi
    klawisz = cv2.waitKey(1) #zatrzymanie okna z obrazem
```

Moduł `withoutbg`

- Moduł, którego jedynym zadaniem jest usunięcie tła z obrazu za pomocą wyuczonego modelu AI.
- W Internecie można znaleźć wiele narzędzi tego typu, jednak najczęściej mają one ograniczenia (limit rozdzielczości, znak wodny) lub są płatne.
- `withoutbg` nie ma takich ograniczeń, jest darmowy i działa dość dobrze.

```
from withoutbg import *

wb = WithoutBG.opensource()
result = wb.remove_background('cat.png')
result.save('cat_without_bg.png')
```

- Python zawiera wiele tego typu przydatnych i prostych w użyciu modułów.

Moduły `tkinter` oraz `PyQt5` – tworzenie graficznego interfejsu użytkownika

- Za pomocą modułów `tkinter` i `PyQt5` możemy tworzyć aplikacje okienkowe z graficznym interfejsem użytkownika (ang. Graphical User Interface, GUI).
- Moduły umożliwiają wstawianie przycisków, suwaków, checkboksów, menu, list rozwijanych, pól tekstowych itp.
- Na wykładzie omówiony zostanie jeden z tych modułów – `tkinter`.
- Jego możliwości i zasadę działania najłatwiej zrozumieć pisząc przykładowy program „od zera”, a więc to zrobimy.
- Pełny kod tego programu wraz z komentarzami wyjaśniającymi poszczególne instrukcje można pobrać z:

https://dysk.kia.prz.edu.pl/d/s/174WjyIujoJ7RQvsRckOo1tM1JUtFwzN/j3KFsfz-Lj7-riY2vHHhs0L6vlWKy8Sa-JrWg3gX1-Qw

W Pythonie można nawet tworzyć prostą grafikę 3D – `PyOpenGL`

- Moduł PyOpenGL umożliwia proste, niskopoziomowe generowanie grafiki 3D. Możemy wstawiać gotowe bryły/obiekty lub tworzyć własne, ustawiać kamerę (punkt i kierunek „patrzenia”), oświetlenie, teksturowanie itp.

- Przykładowa funkcja z kolejnego slajdu generuje sześcian, którego każda ściana ma inny kolor.

- Dla każdej ściany definiujemy kolor oraz współrzędne wszystkich czterech jej wierzchołków.

PyOpenGL – rysowanie sześcianu

GL_QUADS oznacza, że grafikę tworzymy z czworokątów. Inną popularną techniką jest GL_TRIANGLES, czyli tworzenie grafiki za pomocą trójkątów. Takie wielokąty nazywają się poligonami.

Narzędzia do tworzenia zaawansowanej grafiki 3D najczęściej bazują na trójkątach (z nich składają się siatki modeli, np. postaci w grze komputerowej).

```
def rysuj_szescian():
    glBegin(GL_QUADS)
    # przód (czerwony)
    glColor3f(1, 0, 0)
    glVertex3f( 1, 1, 1)
    glVertex3f(-1, 1, 1)
    glVertex3f(-1,-1, 1)
    glVertex3f( 1,-1, 1)
    # tył (zielony)
    glColor3f(0, 1, 0)
    glVertex3f( 1, 1,-1)
    glVertex3f(-1, 1,-1)
    glVertex3f(-1,-1,-1)
    glVertex3f( 1,-1,-1)
    # lewo (niebieski)
    glColor3f(0, 0, 1)
    glVertex3f(-1, 1, 1)
    glVertex3f(-1, 1,-1)
    glVertex3f(-1,-1,-1)
    glVertex3f(-1,-1, 1)
```

```
    # prawo (żółty)
    glColor3f(1, 1, 0)
    glVertex3f( 1, 1, 1)
    glVertex3f( 1, 1,-1)
    glVertex3f( 1,-1,-1)
    glVertex3f( 1,-1, 1)
    # góra (karmazynowy - cyan)
    glColor3f(1, 0, 1)
    glVertex3f( 1, 1, 1)
    glVertex3f(-1, 1, 1)
    glVertex3f(-1, 1,-1)
    glVertex3f( 1, 1,-1)
    # dół (cyjanowy - magenta)
    glColor3f(0, 1, 1)
    glVertex3f( 1,-1, 1)
    glVertex3f(-1,-1, 1)
    glVertex3f(-1,-1,-1)
    glVertex3f( 1,-1,-1)
    glEnd()
```

PyOpenGL – animacja obracającego się sześcianu

- Z podanej poniżej strony można pobrać kod, który korzysta z funkcji przedstawionej w poprzednim slajdzie i dodatkowo ustawia odpowiednio kamerę i wprawia sześcian w ruch (animacja obracania).
- Kod zawiera komentarze wyjaśniające poszczególne instrukcje.

https://dysk.kia.prz.edu.pl/d/s/174WlV9CyP1wBmHPX1z5u6H8mnjMQ9Yl/dLupNrSy1RU2slhavmTqALlrOBTw-HtB-HbXgMgD1-Qw

- Ciekawostka: OpenGL (w Pythonie lub C++) jest podstawą niektórych silników graficznych używanych do tworzenia gier indie (mniejszych produkcji), np. silnik Panda3D.
- W przypadku większych, wysokobudżetowych gier stosuje się silniki graficzne, takie jak Unreal i Unity, które opierają się na bibliotece/module Direct3D.

Inne popularne moduły języka Python

- Przykładowe popularne moduły dostępne w języku Python:
  - `scikit-learn`, `pytorch`, `tensorflow` – moduły do uczenia maszynowego (klasyfikacja, regresja, klasteryzacja, głębokie uczenie)
  - `pillow` – przetwarzanie obrazów
  - `pydub` – przetwarzanie dźwięków
  - `vidgear` – przetwarzanie filmów
  - `mysql-connector` – zarządzanie bazami danych MySQL
  - `pygame` – tworzenie prostych gier i grafiki 2D

Instalacja modułów

- W języku Python dostępnych jest bardzo wiele modułów.

- Niektóre z nich (np. `math`, `random`, `pickle`) są domyślnie zainstalowane i wystarczy je zaimportować.

- Wiele modułów (np. `numpy`, `matplotlib`, `pandas`, `opencv-python`) wymaga pobrania i instalacji. Możemy to zrobić za pomocą wyszukiwarki modułów w środowisku programistycznym. Np. w środowisku Pycharm:
  - File → Settings → Python → Interpreter

    Jeśli interesującego nas modułu nie ma na liście, można go wyszukać klikając w przycisk „+” powyżej listy.

Pisanie programu w wielu plikach

- Dłuższe programy, zawierające wiele klas i funkcji, piszemy zazwyczaj w wielu plikach.

- Dobrą praktyką jest definiowanie każdej klasy w osobnym pliku (szczególnie jeśli klasy są rozbudowane i składają się z długiego kodu). Jeśli mamy jakieś ogólne zbiory funkcji lub stałych, to również można przechowywać je w osobnym pliku/plikach.

- Domyślnie program uruchamia się zaczynając od pliku o nazwie „main.py”.
  - W niektórych środowiskach jest to wymagane.
  - Inne środowiska (np. Pycharm) umożliwiają uruchamianie programów z plików o dowolnej nazwie.

Pisanie programu w wielu plikach

- Jeśli w danym pliku chcemy korzystać z kodu zapisanego w innym pliku, należy go dołączyć w taki sam sposób, w jaki dołączamy moduły, wyrażeniem:

```
import <nazwa pliku>
```

- Nazwa pliku nie powinna zawierać rozszerzenia. Interpreter wie, że kod znajduje się w plikach z rozszerzeniem „.py”, więc nie musimy go o tym informować.

- Pliki, które dołączamy, powinny znajdować się w tym samym katalogu, co plik „main.py”.
  - Jeśli koniecznie chcemy importować pliki z innego katalogu, to można to zrobić, jeśli dodamy do niego wcześniej ścieżkę za pomocą funkcji `sys.path.append` z modułu `sys`.

Pisanie programu w wielu plikach – przykład I

- W poniższym przykładzie program składa się z dwóch plików: „main.py” oraz „wielomiany.py”.
- Plik „wielomiany.py” zawiera funkcje obliczające wartości zgodnie ze wzorami wielomianów drugiego i trzeciego stopnia.
- Plik „main.py” korzysta z tych funkcji, a więc w pierwszej linijce importuje plik „wielomiany”.

main.py

```
import wielomiany

liczba = float(input('Podaj liczbę: '))
wynik_2 = wielomiany.drugiego_stopnia(liczba)
wynik_3 = wielomiany.trzeciego_stopnia(liczba)
print(wynik_2, wynik_3)
```

wielomiany.py

```
def drugiego_stopnia(x):
    return x**2 + 2*x + 1

def trzeciego_stopnia(x):
    return x**3 + 2*x**2 + 4*x + 1
```

Pisanie programu w wielu plikach – przykład I

- Pliki, które dołączamy działają tak samo jak moduły.

- W związku z tym, jeśli nie chcemy pisać nazwy modułu za każdym razem przed nazwą składnika, możemy zaimportować go używając składni:

```
from <nazwa pliku> import <nazwy składników>
```

main.py

```
from wielomiany import drugiego_stopnia
from wielomiany import trzeciego_stopnia

liczba = float(input('Podaj liczbę: '))
wynik_2 = drugiego_stopnia(liczba)
wynik_3 = trzeciego_stopnia(liczba)
print(wynik_2, wynik_3)
```

wielomiany.py

```
def drugiego_stopnia(x):
    return x**2 + 2*x + 1

def trzeciego_stopnia(x):
    return x**3 + 2*x**2 + 4*x + 1
```

Pisanie programu w wielu plikach – przykład II

- W drugim przykładzie program składa się z trzech plików: „main.py”, „ptak.py” i „zwierze.py”.
- Plik „zwierze.py” zawiera definicję klasy `Zwierze`.
- Plik „ptak.py” zawiera definicję klasy `Ptak`, która dziedziczy po klasie `Zwierze`. Z tego powodu importuje on zawartość pliku „zwierze.py”.
- W pliku „main.py” tworzony jest obiekt klasy `Ptak`. Z tego powodu importuje on zawartość pliku „ptak.py”.

Pisanie programu w wielu plikach – przykład II

zwierze.py

```
class Zwierze:
    masa_ciala = 0 #[kg]
    kolor = ''
    def poruszaj_sie(self, odleglosc):
        print('Poruszam się o', odleglosc, 'metrów.')
    def daj_glos(self):
        print('Daję głos.')
```

ptak.py

```
from zwierze import Zwierze

class Ptak(Zwierze):
    rozpietosc_skrzydel = 0 #[m]
    def lec(self, wysokosc):
        print('Lecę na wysokości', wysokosc, "metrów.")
    def daj_glos(self):
        print('Krrrraaa')
```

main.py

```
from ptak import Ptak

kruk = Ptak()
kruk.kolor = 'czarny'
kruk.masa_ciala = 1
kruk.rozpietosc_skrzydel = 1.2
kruk.lec(3, 4)
kruk.daj_glos()
```

Tworzenie pliku wykonywalnego programu napisanego w Pythonie

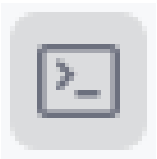
- Istnieje możliwość utworzenia pliku wykonywalnego naszego programu (np. .exe w systemie Windows).

- W ten sposób nie będziemy musieli uruchamiać go w środowisku programistycznym albo wywołując interpreter Pythona poleceniami w konsoli.

- Zamiast tego będziemy mogli uruchomić program prostym dwukrotnym kliknięciem w ikonę pliku .exe.

Tworzenie pliku wykonywalnego – program Pyinstaller

- Tworzenie pliku wykonywalnego jest możliwe przy użyciu programu Pyinstaller.

- Pyinstaller musimy najpierw zainstalować. W tym celu w środowisku Pycharm należy uruchomić terminal klikając w ikonę znajdującą się w lewym dolnym rogu okna Pycharma: 

- Następnie należy wpisać polecenie: `pip install pyinstaller`

Tworzenie pliku wykonywalnego – program Pyinstaller

- Po udanej instalacji Pyinstallera, w celu utworzenia pliku .exe, należy wpisać w terminalu polecenie:

```
pyinstaller <program> --onefile
```

- W miejscu `<program>` należy wpisać nazwę (wraz z rozszerzeniem) pliku z kodem naszego programu. Jeśli program jest pisany w wielu plikach, należy wpisać nazwę pliku głównego, od którego zaczyna się program.
- Przykładowo, jeśli plik główny programu nazywa się *main.py*, należy wpisać polecenie:

```
pyinstaller main.py --onefile
```

- Po wykonaniu tego polecenia, w katalogu z kodem programu zostanie utworzony katalog o nazwie *dist*, w którym pojawi się plik .exe z naszym programem.

Uruchamianie programu z pliku .exe

- Uruchamiając napisany program konsolowy najprawdopodobniej natrafimy na dziwny problem. Okno z konsolą zamknie się przed wypisaniem końcowych wyników. Dlaczego tak się dzieje?
- Ponieważ okno domyślnie zamyka się zawsze po wykonaniu ostatniej instrukcji programu. Jak temu zapobiec?
- Wystarczy dopisać na sam koniec programu instrukcję pobierania danych, która zatrzyma nasz program dopóki użytkownik nie wciśnie klawisza Enter. Przykład:

```
a = int(input('Podaj liczbę: '))
print(a+5)
input()
```

wstrzymanie programu przed jego zamknięciem

Tworzenie pliku wykonywalnego bez wyświetlania okna konsoli

- Jeśli tworzymy aplikację okienkową i nie chcemy, żeby po uruchomieniu programu wyświetlała się konsola, możemy skorzystać z opcji `--windowed`.

```
pyinstaller main.py --onefile --windowed
```

- Więcej informacji na temat programu Pyinstaller, w tym opis jego parametrów, znajduje się na stronie: *https://pyinstaller.org/en/latest/usage.html*