Python: podstawy programowania

Dzień 1 | v. 1.1.1

Coders Lab
SZKOŁA PROGRAMOWANIA

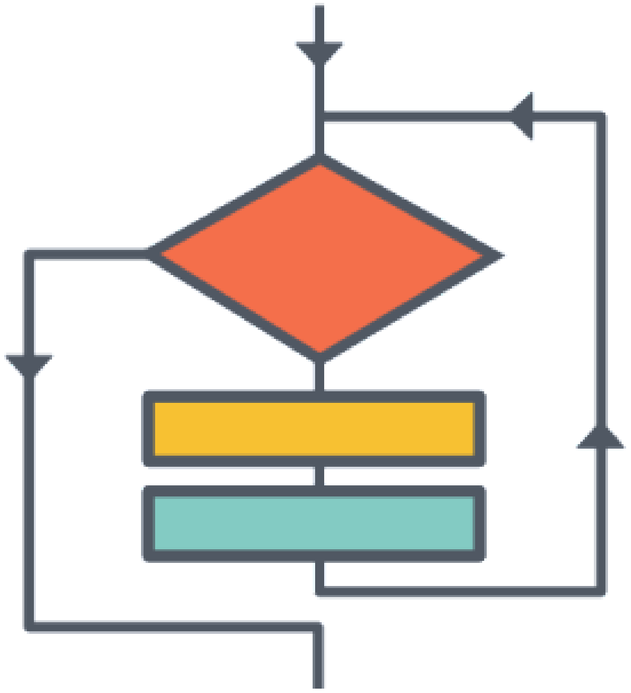
Powtórzenie z preworku

Co to jest algorytm?

Algorytm – skończony ciąg jasno zdefiniowanych czynności koniecznych do wykonania pewnego rodzaju zadań.

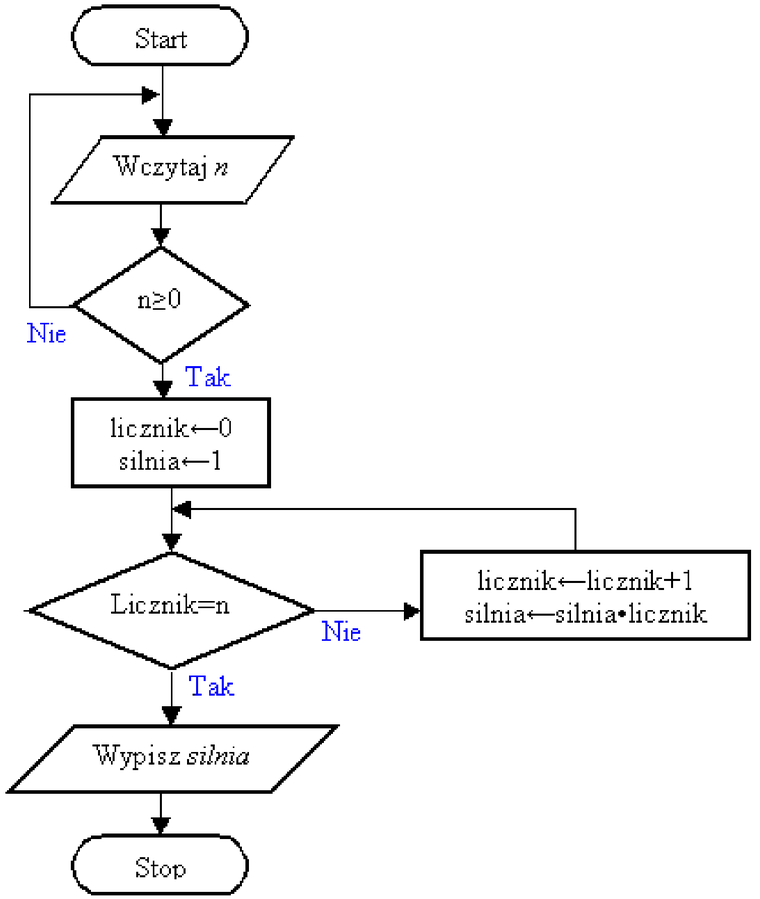
Gdzie na co dzień spotykamy się z algorytmami?

- W kuchni – wszystkie przepisy.
- Na drogach – zasady ruchu drogowego.
- Praktycznie w każdej dziedzinie naszego życia.



Schemat blokowy (flowchart)

- **Strzałka** − wskazuje jednoznacznie powiązania i ich kierunek.
- **Prostokąt** – zawiera wszystkie operacje z wyjątkiem instrukcji wyboru.
- **Równoległobok** –wejście/wyjście danych.
- **Romb** – wpisujemy wyłącznie instrukcje wyboru,
- **Owal** – oznacza początek bądź koniec schematu.



if oraz if razem z else

Tę instrukcję warunkową można porównać do obecnego w języku naturalnym stwierdzenia:

jeśli (**if**)…, to zrób…,

jeśli nie poprzednie ale (**elif**) …, to zrób …

jeśli nie wszystkie poprzednie (**else**), to zrób….

Część **elif** i **else** jest opcjonalna.

```
if wyrażenie_warunkowe:
    # blok kodu wykonywany,
    # jeśli spełniony zostanie warunek

elif inne_wyrażenie_warunkowe:
    # blok kodu wykonywany, jeśli
    # spełniony zostanie drugi warunek,
    # a pierwszy nie

else:
    # blok kodu wykonywany, jeśli
    # nie zostanie spełniony żaden
    # z poprzednich warunków
```

Blok kodu

Zwróćcie uwagę na dwukropek na końcu instrukcji i wcięcia w kodzie po każdej z nich.

Jest to tzw. **blok kodu**, czyli instrukcje, które muszą się wykonać w reakcji na **if**, **elif** lub **else** (lub inne komendy, ale o tym później).

W Pythonie wcięcia bloków kodu **są obowiązkowe**! Jeśli są niepoprawne, interpreter Pythona zgłosi błąd!

Wcięcia najlepiej robić spacją: jeden poziom wcięcia, to 4 spacje.

```
if wyrażenie_warunkowe:
    # blok kodu wykonywany,
    # jeśli spełniony zostanie warunek

elif inne_wyrażenie_warunkowe:
    # blok kodu wykonywany, jeśli
    # spełniony zostanie drugi warunek,
    # a pierwszy nie

else:
    # blok kodu wykonywany, jeśli
    # nie zostanie spełniony żaden
    # z poprzednich warunków
```

Komenda pass

Blok kodu nie może być pusty!

Jeśli z jakiegoś powodu nie chcesz umieszczać w nim żadnej sensownej komendy, użyj słowa kluczowego **pass**.

Ta komenda nic nie robi, jedynie zapełnia puste miejsce i zabezpiecza nas przed błędem, wynikającym z pustego bloku kodu.

```
if(wyrażenie_warunkowe):
    pass

elif(inne_wyrażenie_warunkowe):
    pass

else:
    pass
```

Pętle i iteracja

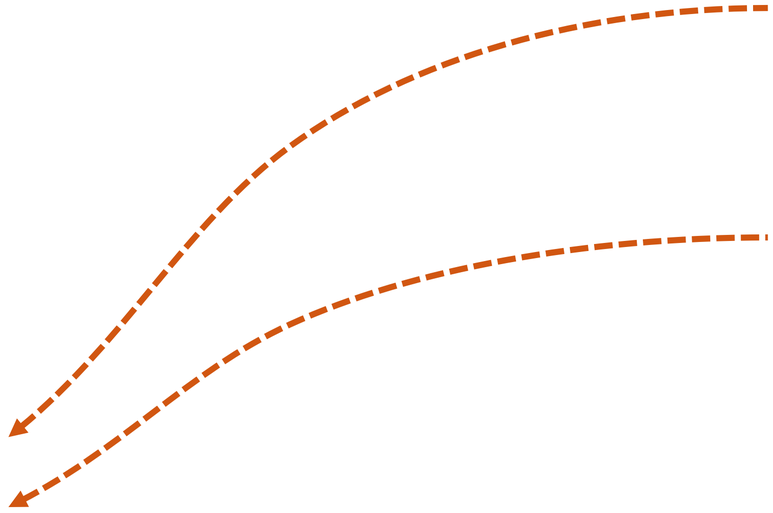
pętla for – in

pętla while

```
heroes = ["Luke Skywalker",
          "Princess Leia", "Han Solo",
          "Chewbacca", "Obi-Wan Kenobi"]

for character in heroes:
    print(character)

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



Wynik:

```
Luke Skywalker
Princess Leia
Han Solo
Chewbacca
Obi-Wan Kenobi
```

Funkcje

Funkcje

- Funkcje to odseparowany kawałek kodu wykonujący jakieś działanie i zwracający jakąś wartość.
- Funkcji używamy w celu przejrzystości i możliwości ponownego używania kodu.

Przykładowa funkcja

```
def sum(x, y):
    z = x + y
    return z

num1 = 20
num2 = 15
num_sum = sum(num1, num2)

print("Suma liczb ", num1, "i", num2, \
      "wynosi", num_sum)
```

Funkcje

Deklaracja funkcji zaczyna się od słowa kluczowego **def**, potem wpisujemy nazwę naszej funkcji (nazwy zwyczajowo są po angielsku).

```
def sum(x, y):
    z = x + y
    return z
```

Po dwukropku (pamiętajcie o wcięciach, bez tego nie zadziała!) znajduje się **ciało funkcji**, czyli kod, który uruchomi się po jej wywołaniu.

Słowo kluczowe **return** oznacza miejsce w którym **nasza funkcja się kończy**. Powoduje też zwrócenie podanej wartości do miejsca gdzie wywołaliśmy funkcję.

Wywołanie funkcji to miejsce gdzie używamy wcześniej napisanej funkcji. W tym miejscu pojawi się jej wynik który zwracamy za pomocą **return**.

```
num1 = 20
num2 = 15
num_sum = sum(num1, num2)
print("Suma liczb ", num1, "i", num2, "wynosi", num_sum)
```

Funkcje - argumenty

Podczas deklaracji funkcji możemy wskazać też jej **argumenty**. Są to zmienne które trzeba podać podczas jej użycia.

```
def sum(x, y):
    z = x + y
    return z
```

Argumentów używamy jak każdej innej zmiennej. Po prostu do chwili użycia nie będziemy znali ich dokładnych wartości.

Podczas wywołania funkcji musimy podać **wszystkie argumenty** których funkcja się spodziewa. Mogą to być zarówno nasze zmienne, jak i bezpośrednio wpisane wartości.

```
num1 = 20
num2 = 15
num_sum = sum(num1, num2)
print("Suma liczb ", num1, "i", num2, "wynosi", num_sum)
```

Funkcje - argumenty

- W Pythonie niektóre z argumentów mogą być **opcjonalne**,
- oznacza to, że możemy je podać i nadać im wartość,
- ale jeśli ich nie podamy, to przyjmą wartość domyślną.

Przykładowa funkcja z argumentami opcjonalnymi

```
def sum(x, y, z=2):
    return x + y + z

new_sum = sum(8, 20)
print("Suma liczb wynosi", num_sum)
```

Argument **z** deklarujemy jako opcjonalny, z domyślną wartością **2**.

Wynik:

```
Suma liczb wynosi 30
```

```
new_sum = sum(8, 20, 100)
print("Suma liczb wynosi", num_sum)
```

Wynik:

```
Suma liczb wynosi 128
```

Funkcje - argumenty

- Można też podawać więcej niż jeden parametr opcjonalny,
- Możemy wtedy podać część opcjonalnych parametrów, wywołując je po nazwie.

```
def sum(x, y=10, z=20):
    return x + y + z

new_sum = sum(20, z=100)
print new_sum
```

Wynik:

```
Suma liczb wynosi 130
```

Dwa typy funkcji

Funkcje w programowaniu możemy podzielić na dwa typy:

Funkcje które nie zwracają wartości:
Są to funkcje gdzie interesuje nas efekt uboczny wywołania takiej funkcji.

Może być to np.:
- Wyświetlenie wiadomości na ekranie,
- Wysłanie maila przez serwer,
- Zapisanie informacji do bazy danych.

Funkcje które zwracają wartość:
Są to funkcje gdzie interesuje nas zwracana wartość. Używają one słowa kluczowego **return** a wartość zwracaną przez nie możemy zapisać do zmiennej.

Może być to np.:
- Najróżniejsze obliczenia matematyczne (potęgowanie, pierwiastkowanie, etc.),
- Wczytywanie informacji z bazy danych,
- Wyszukiwanie danych w zbiorze.

Dwa typy funkcji

Funkcje które nie zwracają wartości:

```
def say_hello(user_name):
    print("Hello", user_name)
```

Funkcja **nie** używa słowa kluczowego **return** czyli **nie** zwraca żadnej wartości.

```
user = "Dave"
say_hello(user)
```

Funkcja nie zwraca wartości więc po prostu ją uruchamiamy.

Funkcje które zwracają wartość:

```
def sum(x, y):
    z = x + y
    return z
```

Funkcja używa słowa kluczowego **return** czyli zwraca wartość.

```
num_sum = sum(12, 30)
print("Suma liczb =", num_sum)
```

Jako że interesuje nas wynik tej funkcji to wartość przez nią zwracaną zapisujemy do zmiennej. Zmiennej tej możemy potem użyć w dalszej części naszego kodu.

Magiczne parametry funkcji

Magiczne parametry funkcji

Parametr z gwiazdką (*args)

Funkcjom Pythona możemy przekazać wiele parametrów. Niektóre z nich możemy uczynić opcjonalnymi, podając domyślną wartość podczas ich definicji.

Może zdarzyć się taka sytuacja, np. w programowaniu obiektowym, gdy chcemy przeciążyć (nadpisać) jakąś metodę, że nie wiemy, jaką liczbę parametrów będziemy potrzebować.

W takiej sytuacji możemy użyć magicznej zmiennej zaczynającej się jedną gwiazdką, np. ***args**. Jeśli mamy zdefiniowaną taką zmienną, podajemy dowolną liczbę parametrów niezależnie od tego ile ich zdefiniowaliśmy.

Wewnątrz funkcji taka zmienna jest widoczna jako krotka.

Przykład

```
def show_ship_passengers(ship, *psngrs):
    print("Ship:", ship)
    for p in psngrs:
        print(p)

show_ship_passengers("Millenium Falcon",
"Han", "Luke", "Leia", "Ben", "Chewie")
```

Wynik:

```
Ship: Millenium Falcon
Han
Luke
Leia
Ben
Chewie
```

Magiczne parametry funkcji

Parametr z dwoma gwiazdkami (**kwargs)

Podobnie działa magiczny parametr poprzedzony dwiema gwiazdkami. Pozwala on na dodanie dowolnej liczby parametrów poprzedzonych kluczem. Wewnątrz funkcji parametr taki jest widoczny jako słownik.

```
def list_me(**kwargs):
    if kwargs is not None:
        for a in kwargs:
            print(a, "-", kwargs[a])
```

Przykład

```
list_me(race="Wookie", name="Chewbacca")
```

Wynik

```
name - Chewbacca
race – Wookie
```

```
list_me(ship="TIE Fighter",
        allegiance="Empire",
        beam="green")
```

Wynik

```
allegiance - Empire
beam - green
ship - TIE Fighter
```

Poprawne tworzenie funkcji
i ponowne ich używanie
jest bardzo ważne
w programowaniu!

DRY = Don’t Repeat Yourself!

— Andy Hunt, Dave Thomas