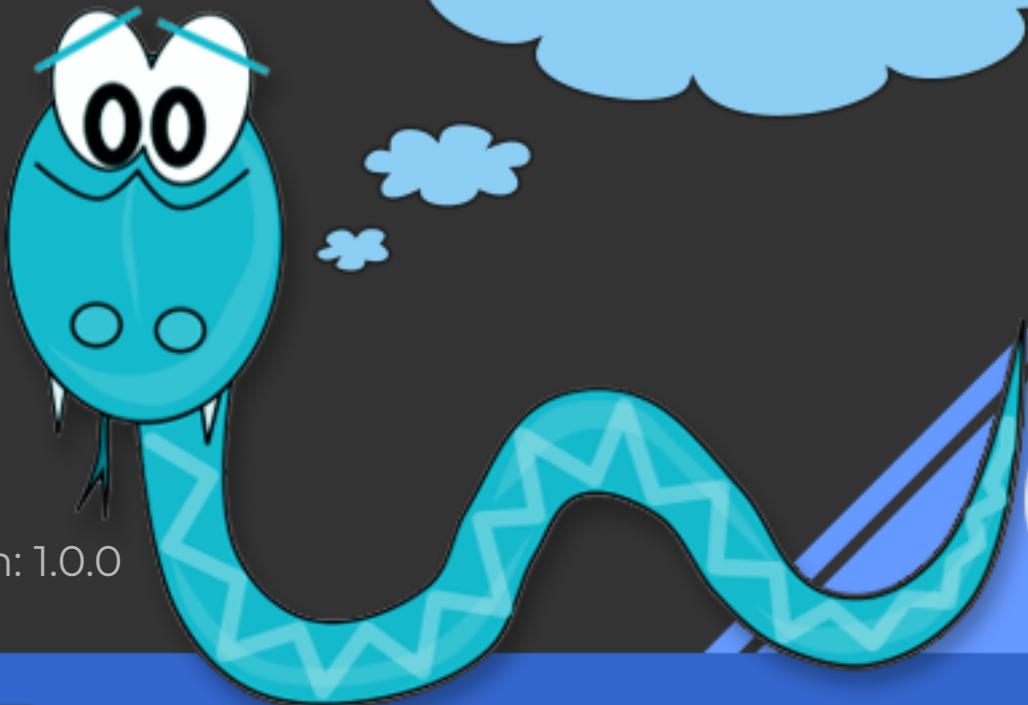


Mateusz Wiliński

CYFROWA GADZINA

Podstawy programowania w języku Python

01101100 11001101 ...



1

--version: 1.0.0



```
print("Jak okiełznać cyfrową gadzinę?")
```

CYFROWA GADZINA

Podstawy programowania w języku Python

Ebook może być dowolnie rozpowszechniany
i używany w dowolnym celu.

Zapraszamy do współpracy przy jego tworzeniu i ulepszaniu.

Kolorowanie składni kodu na podstawie VSC Light+

Link do aktualnej wersji:

[Cyfrowa gadzina](#)

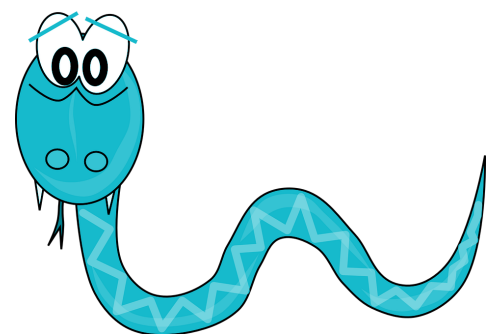
Autor: Mateusz Wiliński

Email: info.wilnox@gmail.com

Data ostatniej aktualizacji: 06.2022

Zapraszamy na stronę <https://wilnox.com/>

Odwiedź stronę: <https://edabit.com/challenges> gdzie znajdziesz mnóstwo zadań związanych z programowaniem w python.



Spis zagadnień

Spis zagadnień	2
Wprowadzenie	5
Python, lider wśród języków programowania	5
Jak tworzyć kod?	6
Typy danych	7
Wyświetlanie informacji	8
Przetwarzanie kodu	8
Zmienne	9
Tworzenie zmiennych	9
Zmiana typu	11
Sprawdzanie typu	11
Konkatenacja	12
Konkatenacja z operatorem + i ,	12
Konkatenacja przez interpolację f	13
Inkrementacja i dekrementacja	14
Operacje na wartościach tekstowych	15
Indeksowanie	15
Wieloliniowe wartości tekstowe	17
Escape characters	17
Wprowadzanie informacji do programu	18
Przykład: Obliczanie potęgi podanej liczby	18
Przykład: Obwód i pole powierzchni kwadratu	19
Instrukcja warunkowa	20
Indentacja	21
Operatory porównania	22
Operatory logiczne	23
Instrukcja warunkowa elif	24
Przykład: Sumy kątów wewnętrznych w wieloboku	25
Losowa liczba	26
	2

Aliasy	26
Losowe liczby całkowite w zakresie	27
Przykład: Działanie matematyczne	28
Pętle	29
Pętla for	29
Pętla while	31
Przykład: Podanie hasła aż do skutku	32
Pętla while i for	33
Break, continue	34
Przykład: Zadanie matematyczne z odpowiedzią	35
Funkcje matematyczne	36
Podstawowe metody matematyczne	36
Exception (wyjątki)	37
Przykład: Cena po obniżce	37
Przykład: Obliczanie pola koła	38
Listy	39
Listy z pętlą	40
Metody na listach	41
Sortowanie listy	42
Metody na listach	43
Mapowanie	43
Przykład: Czy można zbudować trójkąt?	44
Przykład: Losowanie liczb totolotka	45
Zadanie: Różnica między min a max	46
Metody wartości tekstowych	47
Sprawdzenie wystąpienia tekstu	47
Zamiana tekstu - replace()	47
Licznik wystąpień w tekście	47
Metoda split()	48
Zadanie: Średnia podanych ocen	49
Tablica znaków ASCII	49
Funkcje	51

Przykład: Losowe działanie matematyczne	53
Funkcje z return	54
Przykład: Generowanie losowego hasła	54
Przykład: Test matematyczny z punktami	55
Funkcja z parametrem	56
Funkcja z parametrem domyślnym	57
Funkcja z 2 parametrami	58
Zmienne globalne i lokalne	59
Modulo	59
Przykład: Liczba podzielna przez 3	60
Przykład: Gra - Zgadnij liczbę	61
Dictionary	63
Tuple	64
Set	65
Class	66
Data	68
Praca z plikami	70
Tworzenie / zapisywanie do pliku	70
Odczytywanie plików	72
Tworzenie / usuwanie plików i folderów	73
Lista plików i folderów	74
Dodatkowe paczki z pip	75

Wprowadzenie

Python, lider wśród języków programowania

Python jest uznawany za jeden z najłatwiejszych do nauki języków programowania.

Jego syntax (struktura kodu) jest dużo prostszy i krótszy w porównaniu z innymi językami.

Dzięki dostępnym bibliotekom, zastosowaniu i niskiej barierze nauki, język python, wg. statystyk, jest najpopularniejszym językiem programowania w 2022 roku.

Jak tworzyć kod?

Istnieje wiele narzędzi do tworzenia programów za pomocą języka python. Pliki z kodem (skrypty) muszą zostać zinterpretowane, tak, aby maszyna na której ten kod ma się wykonać, mogła go zrozumieć.

Jednym z najprostszych sposobów na rozpoczęcie nauki programowania w języku python jest skorzystanie ze środowisk programistycznych online, np:

<https://www.jdoodle.com/python3-programming-online/>

Na dalszym etapie nauki będziemy potrzebowali interpretera i edytora, który pozwoli tworzyć kod szybciej i wygodniej.

Wystarczy pobrać i zainstalować [Python](#) i środowisko programistyczne IDE, np:

[PyCharm Community](#), [Visual Studio Code](#) lub inne...

Wersja wideo jak przygotować Visual Studio Code do pracy z python (dostępne po 25.06)

<https://youtu.be/6SRQsNAhQm8>

Typy danych

W językach programowania posługujemy się danymi / informacjami różnego typu. Możemy wyróżnić:

liczby całkowite (int), np: -123, 12, 1234

liczby ułamkowe (float), np: -12.5, 0.05, 1.23, 44.95

pojedyncze znaki (chr), np: 'A', '@', '#'

wartości tekstowe (string), np: "Legnica", "Dawno, dawno temu..."

wartości logiczne (bool) oznaczające prawdę lub fałsz: True, False
oraz wiele innych...

Pamiętaj!

Części dziesiętne w liczbach ułamkowych oddzielamy za pomocą kropki!

Pojedyncze znaki i wartości tekstowe umieszczane są wewnątrz apostrofu lub cudzysłowu.

Liczby ułamkowe można również zapisać w notacji wykładniczej, np:

3.52e8 co oznacza liczbę $3.52 * 10^8$, czyli 352000000

Wyświetlanie informacji

W języku python, możemy wyświetlić informacje w konsoli, za pomocą metody **print()**

Wewnątrz nawiasu, jako argument, podajemy informację, która ma zostać wyświetlona. Może to być pojedyncza informacja, (liczba, tekst), wyrażenie matematyczne lub zbiór wielu elementów odpowiednio ze sobą połączonych.

Przykłady użycia metody print do wypisania różnych informacji w konsoli

```
print(12)
```

```
print(36.6)
```

```
print('@')
```

```
print("hello")
```

```
print(12*6)
```

Przetwarzanie kodu

Pamiętaj, że kod w skrypcie, interpretowany jest od góry w dół, od lewej do prawej strony.

Kolejność wykonywania poleceń zależy więc od miejsca ich wywołania. Polecenia wywołane na samej górze zostaną wykonane jako pierwsze.

Zmienne

Tworzenie zmiennych

Możemy sobie wyobrazić, że zmienna, to niewidzialne pudełko, wewnątrz którego możemy przechowywać informacje. Jak nazwa wskazuje, zawartość pudełka może się zmieniać.

Pudełko (czyli zmienna) musi mieć unikalną nazwę, tak, abyśmy mogli je w każdej chwili odnaleźć i zajrzeć do środka. Za każdym razem kiedy będziemy korzystać z nazwy zmiennej, interpreter odnajdzie to pudełko w pamięci komputera, zajrzy do środka i zwróci jego zawartość.

Zawartość pudełka, czyli wartość zmiennej przypisujemy za pomocą operatora przypisania, czyli znaku =.

Zmienną tworzymy podając jej nazwę, a następnie przypisujemy do niej pierwszą zawartość. W niektórych językach programowania musimy jeszcze podać jaki typ informacji będziemy przechowywać w danej zmiennej. Python sam określa jakiego typu jest wartość przypisana do zmiennej, dlatego tutaj nie jest to wymagane.

Przykłady tworzenia zmiennych:

```
imie = "Janusz"
```

```
wiek = 23
```

```
kwotaDoZaplaty = 102.99
```

Język python jest case-sensitive, to znaczy, że rozróżnia małe i wielkie litery alfabetu.

Zmienne: imie, Imie, IMIE to trzy zupełnie inne zmienne.

Inny sposób utworzenia wielu zmiennych:

```
x, y, z = "Orange", "Banana", "Cherry"
```

Ustalając nazwę zmiennej powinniśmy przestrzegać kilku zasad:

- zaczynamy od małej litery
- używamy liter podstawowych (bez polskich akcentów)
- możemy używać cyfr lub podkreślenia (_), ale raczej się tego unika. Nie można zaczynać nazwy od cyfry
- nazwy, składające się z wielu wyrazów, łączymy zaczynając kolejne wyrazy wielką literą, np(mojAdres, liczbaPunktow).

Nazewnictwo takie nosi miano camelCase

- w nazwach nie stosujemy spacji

Nazwy muszą być deskryptywne, aby od razu było wiadomo, do czego służą!

Utworzone zmienne przechowują w sobie dane, do których będziemy chcieli się dostać w dalszej części kodu.

Dostęp do zmiennej uzyskujemy na podstawie jej nazwy.

Możemy pobrać zawarte w niej dane lub przypisać do niej nową wartość.

Przykład utworzenia zmiennej, wyświetlenie zawartości, nadpisanie zmiennej i ponowne wypisanie w konsoli:

```
liczbaPunktow = 0
print(liczbaPunktow)
liczbaPunktow = 3
print(liczbaPunktow)
```

Zmiana typu

Castowanie to zmiana typu danej wartości na inną.

```
x = str(3)    # wartość stypu string '3'
y = int(3)    # wartość stypu integer 3
z = float(3)  # wartość stypu float 3.0
print(x)
print(y)
print(z)
```

Sprawdzanie typu

W każdym momencie możemy sprawdzić jakiego typu wartość jest przechowywana w danej zmiennej:

```
liczba = 5
imie = "Ania"
print(type(liczba))
print(type(imie))
```

Konkatenacja

Konkatenacja z operatorem + i ,

Często będziemy chcieli połączyć kilka informacji w jedno zdanie. Łączenie takie, nazywamy konkatenacją. Możemy zrobić to na wiele sposobów, np. za pomocą operatora + lub przecinka. Tworząc konkatenację z liczbami, używając operatora +, musimy zamieniać je na wartości tekstowe za pomocą metody str().

Przykłady konkatenacji z operatorem + i przecinkiem.

```
liczbaPunktow = 10
```

```
# Konkatenacja z użyciem operatora +
```

```
print("Liczba punktow: " + str(liczbaPunktow))
```

```
# Konkatenacja z użyciem przecinka
```

```
print("Liczba punktow:",liczbaPunktow)
```

Konkatenacja, ostatecznie, tworzy treść typu tekstowego (string).

Konkatenacja przez interpolację f

Innym, eleganckim, sposobem konkatenacji, jest użycie interpolacji f.

Na początku tworzonego ciągu dodajemy literę f, a następnie kolejne zmienne podajemy wewnątrz nawiasu klamrowego.

Jest to preferowana opcja i będzie wykorzystywana w większości przykładów przedstawionych w tej publikacji.

przykład 1

```
uczen = "Marcel"
```

```
punkty = 99
```

```
print(f"Uczeń {uczen} zdobył {punkty} punktów z egzaminu.")
```

przykład 2

```
print(f"2 + 4 = { 2 + 4 }")
```

Inkrementacja i dekrementacja

Wartości zmiennych możemy modyfikować zwiększając jej wartość, bazując na jej poprzedniej wartości.

W takich przypadkach wykonuje się operację zwiększenia / zmniejszenia, a następnie przypisania nowej wartości do zmiennej. Zabieg taki nazywamy inkrementacją lub dekrementacją.

Najpierw podaje się operację arytmetyczną, następnie znak przypisania i wartość zwiększającą.

Przykłady inkrementacji / dekrementacji

```
liczbaPunktow = 0
```

```
liczbaPunktow += 1    #zwiększenie o 1
```

```
liczbaPunktow *= 3    #zwiększenie x3
```

```
liczbaPunktow **= 2   #zwiększenie ^2
```

```
liczbaPunktow -= 5    #zmniejszenie o 5
```

```
liczbaPunktow /= 2    #zmniejszenie x2
```

```
print(liczbaPunktow)
```

Operacje na wartościach tekstowych

Indeksowanie

Często zachodzi potrzeba, by wyłonić z wyrazu lub zdania tylko część znaków. W wartościach tekstowych mamy do czynienia z łańcuchem znaków i każdy symbol ma swój numer porządkowy, tzw. numer indeksu.

W programowaniu, indeksowanie, czyli liczenie, zaczynamy od 0. Numer indeksu oznaczamy wewnątrz nawiasu kwadratowego []

Przykład:

w wyrazie "python", znak 'h' ma numer indeksu równy 3.

```
# Wypisz pierwszy znak wyrazu "tekst"
```

```
print("tekst"[0])
```

```
# Wypisz trzeci znak wyrazu "domek"
```

```
print("domek"[2])
```


Za pomocą dwukropka(:) ustalamy zakres.

Dzięki wartościom ujemnym możemy wybrać liczby zaczynając od końca łańcucha znaków.

```
# Wypisz trzy pierwsze znaki wyrazu "tekst"  
print("tekst"[0:3])
```

```
# Wypisz 2 i 3 znak podanej zmiennej  
imie = "Ania"  
print(imie[1:3])
```

```
# Wypisz trzy ostatnie znaki wyrazu "Programowanie" (nie)  
print("Programowanie"[-3:])
```

```
# Wypisz 3 znaki wyrazu "Komputerek" licząc od tyłu, poza  
ostatnim znakiem (ere)  
print("Komputerek"[-4:-1])
```

Wieloliniowe wartości tekstowe

Za pomocą potrójnego cudzysłowu (lub apostrofu) możemy dodać wieloliniowe wartości tekstowe, które zawierają przejścia do nowej linii.

```
tekst = """Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua."""  
print(tekst)
```

Escape characters

Znaki wyjścia pozwalają na formatowanie tekstu (przejście do nowej linii, dodanie cudzysłowu, itp).

\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\xhh	Hex value

Wprowadzanie informacji do programu

Do wprowadzenia treści do programu używamy metody **input()**. Wewnątrz nawiasu metody, możemy podać treść jaka zostanie wyświetlona użytkownikowi, która skłoni go do wpisania odpowiedniej treści. Najczęściej metodę `input()` przypisuje się do zmiennej, tak, aby móc wpisaną wartość zapisać i przetwarzać w programie.

```
#Wyświetlenie informacji na podstawie wprowadzonych danych
wprowadzoneImie = input("Podaj imię: ")
print("Witaj " + wprowadzoneImie)
```

Przykład: Obliczanie potęgi podanej liczby

```
#Obliczenie potęgi dowolnego stopnia podanej liczby
liczba = int(input("Podaj liczbę: "))
potega = int(input("Podaj potęgę: "))
wynik = liczba**potega
print(f"{liczba} ^ {potega} = {wynik}")
```

Przykład: Obwód i pole powierzchni kwadratu

```
#Obliczanie pola i obwodu kwadratu o podanym boku
bok = float(input("Podaj bok kwadratu: "))
pole = bok * bok
obwod = 4 * bok
print(f"Pole wynosi: {pole}")
print(f"Obwód wynosi: {obwod}")
```

Instrukcja warunkowa

Instrukcja warunkowa służy do podejmowania decyzji w oparciu o podane warunki. Sprawdzane warunki zawsze muszą być w takiej formie, aby ich wynik zwracał wartość typu bool, czyli prawdę lub fałsz.

Najprostsza instrukcja posiada jeden warunek po spełnieniu którego wykona się instrukcja. Używając pseudokodu możemy przytoczyć sytuację z życia wziętą:

jeżeli(dostanę co najmniej piątkę z matematyki):
 otrzymam świadectwo z czerwonym paskiem

A teraz w języku python mogłoby to wyglądać w ten sposób:

```
if(ocena >= 5):  
    print("Otrzymujesz świadectwo z cz. paskiem")
```

Syntax instrukcji warunkowej:

```
warunek = True  
if(warunek):  
    print("Instrukcja do wykonania, jeśli warunek jest  
prawdziwy")  
elif(warunek):  
    print("Instrukcja wykona się, jeśli warunek prawdziwy,  
a poprzednie warunki były fałszywe")  
else:  
    print("Instrukcja do wykonania, jeśli wszystkie  
poprzednie warunki były fałszywe")
```

Jeżeli warunek (warunki) będzie prawdziwy, wykona się instrukcja do niego przypisana, a reszta instrukcji warunkowej zostanie pominięta.

Pamiętaj, że w instrukcji warunkowej, może zostać wykonany tylko jeden blok. Będzie to pierwszy blok od góry, w którym warunek zostanie spełniony.

Jeżeli żaden z warunków nie zostanie spełniony, wykona się blok `else` lub nie wykona się nic, jeśli takiego bloku nie dodamy.

Więcej warunków podaje się za pomocą instrukcji **`elif`**

Indentacja

Język python działa na zasadzie indentacji, czyli wcięć kodu (tabulacji). Główne bloki kodu zaczynają się tuż przy krawędzi ekranu, a kolejne, wewnętrzne bloki kodu, tworzy się za pomocą tabulacji.

```
warunek = True
if(warunek):
    # wewnętrzny blok kodu
    print("Instrukcje")
else:
    # wewnętrzny blok kodu
    print("Instrukcje")
```

Operatory porównania

Operatory porównania służą do porównywania wartości w instrukcji warunkowej. Zauważ, że takie porównanie zawsze zwróci wartość typu bool, czyli prawdę lub fałsz.

==	Równe
!=	Nie jest równe
>	Większy niż
<	Mniej niż
>=	Większy niż lub równy
<=	Mniejszy niż lub równy

Zauważ, że sprawdzamy czy wyrażenia są równe za pomocą podwójnego znaku ==.

Pojedynczy znak = służy do przypisywania wartości, np. do zmiennej.

Operatory logiczne

Często zachodzi potrzeba, by w instrukcji warunkowej połączyć kilka warunków.

Przykład 1.

Autem można jechać, gdy kierowca ma prawo jazdy i kluczyki.

Stosujemy operator **and**, ponieważ musimy mieć zarówno prawo jazdy jak i kluczyki.

Przykład 2.

Możemy pojechać do sklepu autem lub rowerem.

Stosujemy operator **or**, ponieważ mamy alternatywę, możemy jechać autem lub rowerem.

and

Stosując operator and oba warunki muszą być prawdziwe, aby całe wyrażenie również było prawdziwe.

or

Stosując operator or wyrażenie będzie prawdziwy, gdy choć jeden z warunków będzie prawdziwy

not

Stosując operator not całe wyrażenie logiczne zmienia wartość (True zmienia się na False i odwrotnie)

Instrukcja warunkowa elif

Instrukcja warunkowa może posiadać wiele osobnych bloków instrukcji, z których ZAWSZE wykona się pierwszy, spełniający warunki instrukcji lub blok else, jeżeli żaden z postawionych warunków nie będzie prawdziwy.

Więcej warunków podaje się za pomocą instrukcji **elif**

#założmy, że mamy kilka przycisków, z których może być aktywny tylko jeden

```
przycisk = "żółty"
```

```
if(przycisk == "zielony"):
    print("Wszystko ok")
elif(przycisk == "żółty"):
    print("Uwaga")
elif(przycisk == "czerwony"):
    print("Stop")
else:
    print("Awaria systemu")
```

Przykład: Sumy kątów wewnętrznych w wieloboku

Przykład wyświetli sumę kątów wewnętrznych w wieloboku o podanej ilości boków.

Sprawdzamy, czy wprowadzona wartość jest liczbą oraz upewniamy się, że wielobok ma co najmniej 3 boki.

Sumę kątów określa się za pomocą wzoru: $(n-2) * 180$

```
n = input("Podaj ilość boków: ")
if(n.isnumeric() and int(n) >= 3):
    sumaKatow = (int(n)-2)*180
    print(f"Suma kątów w {n}-boku wynosi: {sumaKatow}")
else:
    print("Wprowadzono złą wartość")
```

Interpreter stara się wykonać kod w najszybszy sposób.

Rozpatrując warunki w każdej instrukcji następują pewne procesy przyspieszające analizę kodu.

W przypadku użycia operatora logicznego and (w którym wszystkie warunki muszą być prawdziwe) przy napotkaniu pierwszego warunku fałszywego, całe wyrażenie od razu traktowane jest jako fałszywe i sprawdzanie reszty warunków też zostaje pominięte.

W przypadku operatora or przy napotkaniu pierwszego warunku prawdziwego, całe wyrażenie jest prawdziwe, więc sprawdzanie reszty warunków też zostaje pominięte.

Losowa liczba

W celu korzystania z generatora losowych liczb, należy dodać do programu bibliotekę random. Robi się to na samej górze skryptu za pomocą instrukcji import

```
import random
```

do zmiennej przypisujemy metodę generującą losową liczbę:

```
liczba = random.random()
```

Metoda random() zwraca wartość ułamkową od 0 do 1

```
#importujemy bibliotekę random
import random
#losujemy liczbę ułamkową od 0 do 1
liczba = random.random()
print(liczba)
```

Alias

Importując moduły możemy skorzystać z aliasów, czyli zamienić nazwę modułu na krótszą (lub inną bardziej deskryptywną).

Robimy to tak jak w przykładzie poniżej:

```
import random as ran
losowa = ran.random()
print(losowa)
```

Losowe liczby całkowite w zakresie

Chcąc losować liczby całkowite z podanego przedziału możemy posłużyć się metodą `randrange()`, która przyjmuje różne argumenty określające zakres losowanych liczb.

Standardowo wewnątrz nawiasu podajemy wartość minimalną, maksymalną i skok(zazwyczaj 1).

Wartość maksymalna nie jest wliczana do przedziału, zatem należy ją zwiększyć o 1.

```
liczba = random.randrange(min, max, 1)
```

```
#importujemy bibliotekę random
```

```
import random
```

```
#losujemy liczbę od 1 do 10
```

```
losowaLiczba = random.randrange(1, 11, 1)
```

```
#wypisujemy liczbę
```

```
print("Wylosowana liczba: ",losowaLiczba)
```

Przykład: Działanie matematyczne

Przykład programu, wyświetlającego działanie dodawania z losowymi liczbami. Jeśli użytkownik poda prawidłową odpowiedź, wyświetli się komunikat Brawo, w przeciwnym razie wyświetli się komunikat Błąd.

```
import random
l1 = random.randrange(1,11,1)
l2 = random.randrange(1,11,1)
suma = l1 + l2
trescZadania = f"{l1} + {l2} = "
odpowiedz = input(trescZadania)
if(int(odpowiedz) == suma):
    print("Brawo")
else:
    print("Błąd")
```

Pamiętaj, że należy porównywać wartości tego samego typu. Dlatego w przykładzie powyżej castujemy odpowiedz na typ int, ponieważ suma jest wartością liczbową. Moglibyśmy też zrobić odwrotnie, czyli zamienić sumę na wartość tekstową, co w tym przypadku, byłoby nawet lepszym rozwiązaniem, ponieważ uchroniłoby to program przed wprowadzeniem przez użytkownika odpowiedzi innej niż liczbowa.

Pętle

Pętla for

Komputery zostały stworzone po to, by ułatwiać i usprawniać życie ludziom.

Pętla służy do tego, by wykonywać podobne operacje w szybki i łatwy sposób. Wyobraź sobie w jaki sposób napisałbyś program, który wyświetla 1000 kolejnych liczb od 1 do 1000?

```
for i in range(1, 1001):  
    print(i)
```

2 linie kodu sprawiają, że wyświetlimy 1000 kolejnych liczb. Zauważ, że podobnie jak z losową liczbą, tutaj też wartość maksymalną zwiększamy o 1.

i to zmienna lokalna, tzw. iteracyjna (dlatego nazywa się **i** :).

Z każdym przejściem pętli wartość tej zmiennej **i** zmienia się. W tym przypadku zwiększa się o 1, aż dojdzie do 1000, wtedy pętla zakończy działanie.

W pierwszym cyklu pętli została jej przypisana wartość 1 bo taka wartość jest podana w nawiasie. Kompilator sprawdza czy zachodzi tutaj warunek prawdziwy, czyli czy **i** (aktualnie 1) jest w przedziale 1 : 1000, jeśli tak, to przechodzi do wnętrza bloku i drukuje wartość **i** w konsoli.

Zaczyna się kolejny cykl (iteracja) pętli. Teraz **i** przyjmuje wartość

o 1 większą, czyli 2. Ponownie dzieje się to samo, sprawdzane jest czy dwójka jest w zakresie i jeśli tak jest, to kod przechodzi do bloku i drukuje dwójkę, itd..

Przy ostatnim cyklu i zwiększy się do 1001, ale nie będzie już w zakresie (bo wartość maksymalna jest zawsze zwiększana o 1) i wtedy warunek staje się fałszywy, w związku z tym, pętla przestaje działać, a interpretator wychodzi z tego bloku kodu i przechodzi do kolejnych instrukcji w programie.

W tym typie pętli wartość **i** zwiększa się automatycznie o 1, w innych pętlach sami możemy decydować jak ta wartość ma się zmieniać.

Przykład użycia pętli for do wypisania liczb od 1 do 10 w jednym wierszu, oddzielonych przecinkiem.

```
liczby = ""
for i in range(1, 11):
    liczby += str(i) + ", "
print(liczby[:-2])
```

Pętla while

Pętli while możemy użyć wtedy, gdy nie wiemy, ile razy ta pętla powinna się wykonać. Za jej pomocą, możemy wyświetlać działanie lub komunikat, dopóki użytkownik nie wpisze prawidłowej odpowiedzi.

Krótko mówiąc, używamy jej, by wykonywać ciągle jakąś instrukcję, dopóki podany warunek nie będzie prawdziwy.

To jest jej główne zadanie, ale możemy za jej pomocą, robić to, co robimy za pomocą pętli for :)

Pętla while wymaga, aby zmienna iteracyjna została zadeklarowana poza blokiem funkcji, a w bloku pętli będziemy tą zmienną samodzielnie zwiększać lub zmniejszać.

Przykład użycia pętli while do wypisania liczb od 1 do 100

```
i = 1
while (i < 101):
    print(i)
    i += 1
```


Przykład: Podanie hasła aż do skutku

Przykład programu, który wyświetla komunikat o podanie hasła, dopóki prawidłowe hasło nie zostanie wpisane.

```
# powtarzaj komunikat, aż zostanie podane prawidłowe hasło
haslo = "1234"
podaneHaslo = ""
while(haslo != podaneHaslo):
    podaneHaslo = input("Podaj hasło: ")
print("Hasło prawidłowe")
```

Pętla while i for

Przykład programu, który wyświetla liczby parzyste od 0 do 50, za pomocą obu pętli

```
# wyświetla liczby parzyste od 0 do 50
liczba = 0
while(liczba < 51):
    print(liczba)
    liczba += 2
```

```
# wyświetla liczby parzyste od 0 do 50
for i in range(0,51,2):
    print(i)
```

Przykład programu, który wyświetla liczby malejące od 10 do 1

```
i = 10
while(i>=1):
    print(i)
    i-=1

for i in range(10,0,-1):
    print(i)
```

Break, continue

Instrukcja break powoduje przerwanie pętli, a instrukcja continue pozwala pominąć wynik pętli na podstawie podanych warunków.

Przykład programu, który będzie losował 10 liczb od 0 do 10 i jeżeli zostanie wylosowana 1, przerwie działanie programu.

```
import random
for i in range(1,11):
    los = random.randrange(1,11,1)
    if los == 1:
        break
    print(los)
```

Przykład programu, który wyświetla liczby od 1 do 10 poza liczbą 5

```
for i in range(1,11):
    if i == 5:
        continue
    print(i)
```

Przykład: Zadanie matematyczne z odpowiedzią

Przykład, który wyświetla losowe działanie matematyczne do czasu, aż zostanie udzielona prawidłowa odpowiedź.

```
import random
liczba1 = random.randrange(1,20,1)
liczba2 = random.randrange(1,20,1)
suma = liczba1 + liczba2
zadanie = f"{liczba1} + {liczba2} = "
odp = input(zadanie)
while(odp != str(suma)):
    print("Odpowiedź niepoprawna, spróbuj ponownie")
    odp = input(zadanie)
print("Brawo Ty")
```

Funkcje matematyczne

Podstawowe metody matematyczne

Python posiada wiele metod matematycznych, niektóre z nich są dostępne po dodaniu modułu math

```
import math
```

```
# Wybór najmniejszej / największej liczby z podanych
```

```
x = min(5, 10, 25, 34, 1, 101)
```

```
y = max(5, 10, 25, 34, 1, 101)
```

```
# Wartość bezwzględna
```

```
x = abs(-7.25)
```

```
# Potęga / pierwiastek
```

```
import math
```

```
x = pow(4, 3)
```

```
y = math.sqrt(64)
```

```
# Wartość liczby pi
```

```
import math
```

```
x = math.pi
```

Exception (wyjątki)

Wyjątki pozwalają nam zabezpieczyć się przed nieprawidłowym działaniem programu (np. wtedy, gdy zerwie się połączenie z internetem lub ktoś będzie chciał wprowadzić tekst zamiast liczby, itp).

Najłatwiej zrobić to za pomocą bloków try i except. W bloku try: próbujemy wykonać kod, a jeśli się to nie uda, uruchomi się blok except:

Przykład: Cena po obniżce

Przykład programu, który pobierze cenę towaru, wartość procentową obniżki i zwróci cenę po obniżce.

```
cenaPoczat = input("Podaj cenę przed obniżką: ")
obnizka = input("Wartość procentowa obniżki: ")
try :
    cenaPoczat = float(cenaPoczat)
    obnizka = float(obnizka)
    cenaPoObnizce = cenaPoczat - (cenaPoczat * obnizka / 100)
    print("Cena po obniżce = ", cenaPoObnizce)
except:
    print("Nie można obliczyć ceny")
```

Przykład: Obliczanie pola koła

Przykład poniżej pokazuje prosty mechanizm, który wyświetli pole koła tylko wtedy, gdy użytkownik poda liczbę większą od 0.

```
print("Program do obliczania pola koła")

def ObliczPole(r):
    try:
        r = float(r)
        if r <= 0:
            print("Promień musi być większy od 0")
        else:
            pole = 3.14 * r*r
            print(f"Pole wynosi: {pole}")
    except:
        print("Podaj wartość liczbową")

r = input("Podaj promień koła:")
ObliczPole(r)
```

Listy

Listy służą do tworzenia kolekcji, uporządkowanego zbioru danych, które służą pewnemu celowi.

Chcąc przechować wszystkie imiona dzieci z klasy, najłatwiej byłoby je zapisać w liście, bo w przeciwnym razie musielibyśmy stworzyć osobne zmienne, np: uczen1, uczen2, uczen3, itd., a to byłoby bardzo złe. Dzięki temu, że zawrzemy imiona w jednej tablicy, będziemy mogli robić z nimi wiele operacji, np: łatwo je wypisać, posortować alfabetycznie czy sprawdzić ich ilość.

Listy tworzymy przy użyciu nawiasów kwadratowych []. W innych językach programowania, taką kolekcję danych nazywamy Tablicą.

Elementy listy posiadają numery porządkowe (indeksy), a ich zliczanie zaczyna się od 0. Chcąc wyświetlić dany element tablicy używa się jej nazwy z numerem indeksu podanym wewnątrz nawiasu kwadratowego.

tworzenie listy z imionami uczniów i wyświetlenie drugiego elementu (Ania)

```
uczniowie = ["Sara", "Ania", "Maciek", "Zenek"]
```

```
print(uczniowie[1])
```



```
#Pobranie losowego elementu z listy
import random
uczniowie = ["Sara", "Ania", "Maciek", "Zenek"]
print(random.choice(uczniowie))
```

Listy z pętlą

Zawartość listy możemy z łatwością wypisać za pomocą pętli for. W tym przypadku, zmienna **i** nie będzie liczbą porządkową, a danym elementem tablicy.

```
# wypisanie elementów listy uczniowie
uczniowie = ["Sara", "Ania", "Maciek", "Zenek"]
for i in uczniowie:
    print(i)
```

```
# wypisanie elementów listy uczniowie za pomocą indeksu
uczniowie = ["Sara", "Ania", "Maciek", "Zenek"]
for i in range(0, len(uczniowie)):
    print(uczniowie[i])
```

```
# wypisanie elementów listy liczbyTotka
liczbyTotka = [12, 7, 29, 2, 41, 36]
for i in liczbyTotka:
    print(i)
```

Metody na listach

```
import random

kolekcja = 24,45,67
lista = ["python", "C++", "JavaScript"]

lista.append("C#")    #dodaje element na końcu listy
lista.pop()           #usuwa element o podanym w nawiasie
                      #indeksie (lub ostatni, jeżeli brak arg.)
len(lista)            #zwraca liczbę elementów w liście
max(lista)            #zwraca największą wartość na liście
min(lista)            #zwraca najmniejszą wartość na liście
sum(lista)            #zwraca sumę elementów na liście
lista.sort()          #sortujemy rosnąco listę
lista.reverse()       #odwraca kolejność elementów na liście
list(kolekcja)        #zamiana kolekcji na listę
random.shuffle(lista) #zmienia losowo kolejność elementów
listy
```

Sortowanie listy

Zawartość listy możemy sortować, np: liczby rosnąco, wartości tekstowo alfabetycznie, itp.

Sortujemy odnosząc się do nazwy listy, a następnie dopisując po kropce nazwę metody **sort()**

```
# sortowanie i wypisanie elementów listy uczniowie
oddzielonych przecinkiem
uczniowie = ["Sara", "Ania", "Maciek", "Zenek"]
uczniowie.sort()
listaUczniow = ""
for i in uczniowie:
    listaUczniow += i + ", "
print(listaUczniow[:-2])
```

Jeżeli chcemy posortować tablicę w odwrotnym kierunku to wewnątrz nawiasu, jako argument metody `sort()`, możemy dodać opcję `reverse = True`, np:

```
uczniowie.sort(reverse = True)
```

lub po metodzie `sort()` możemy użyć metody `reverse()`

Metody na listach

Python pozwala przypisać elementy tablicy do osobnych zmiennych:

```
uczniowie = ["Ania", "Maciek", "Antoś"]  
x, y, z = uczniowie  
print(x)  
print(y)  
print(z)
```

Mapowanie

Czasami zajdzie potrzeba zmienić tablicę danego typu na inny. W takim przypadku należy posłużyć się metodą mapowania - `map()`.

W przykładzie poniżej zamienimy tablicę typu string na int.

```
oceny = ['4','5','2','1','5','3']  
oceny = list(map(int,oceny))  
print(oceny)
```

Przykład: Czy można zbudować trójkąt?

Przykład programu, który sprawdza czy z podanych długości można zbudować trójkąt.

Aby zbudować trójkąt, suma długości dwóch krótszych boków musi być większa od najdłuższego boku.

```
bokiTrojkata = []
try:
    for i in range(1,4):
        bokiTrojkata.append(float(input(f"Bok nr {i}: ")))
    bokiTrojkata.sort()
    if(bokiTrojkata[0] > 0):
        if(bokiTrojkata[2] < (bokiTrojkata[0] + bokiTrojkata[1])):
            print("Można zbudować trójkąt o podanych bokach.")
            print(f"Obwód trójkąta wynosi: {sum(bokiTrojkata)}")
        else:
            print("Nie można zbudować trójkąta")
    else:
        print("Boki muszą być większe od 0")
except:
    print("Podaj długości boków w postaci liczby")
```

Przykład: Losowanie liczb totolotka

Program, który losuje 6 liczb z 49 i wypisuje je w kolejności rosnącej. Losowane liczby nie mogą się powtarzać.

```
import random

#Tworzymy liczby od 1 do 49 i mieszamy tablice
wszystkieLiczby = []
for i in range(1, 50):
    wszystkieLiczby.append(i)
random.shuffle(wszystkieLiczby)

#Losujemy 6 liczb, wylosowaną liczbę dodajemy do nowej tablicy
wygranych i jednocześnie usuwamy ją z tablicy wszystkich liczb
by uniknąć powtórek
wygraneLiczby = []
for i in range(1, 7):
    wybranaLiczba = random.randrange(0, len(wszystkieLiczby), 1)
    wygraneLiczby.append(wszystkieLiczby[wybranaLiczba-1])
    wszystkieLiczby.pop(wybranaLiczba-1)

wygraneLiczby.sort()
print(wygraneLiczby)
```

Zadanie: Różnica między min a max

Wyświetl liczby rosnąco oraz wyświetl jaka jest różnica między najmniejszą i największą wartością w tablicy:

```
liczby = [8,2,-23,9,56,-29,44,3,-76,12,89,-65,57]
```

Metody wartości tekstowych

Sprawdzenie wystąpienia tekstu

```
txt = "The best things in life are free!"  
if "free" in txt:  
    print("Yes, 'free' is present.")
```

```
txt = "The best things in life are free!"  
if "expensive" not in txt:  
    print("No, 'expensive' is NOT present.")
```

Zamiana tekstu - replace()

```
# Przykład zamienia kropki na przecinki  
txt = "Bułki kosztują 0.99zł, a oranżada 1.29zł."  
txt = txt.replace(".", ",")  
print(txt)
```

Licznik wystąpień w tekście

```
# Przykład zlicza ilość kropek w tekście  
txt = "Bułki kosztują 0.99zł, a oranżada 1.29zł."  
dotCounter = txt.count(".")  
print(dotCounter)
```


Metoda split()

Służy do dzielenia tekstu wg. określonego wzorca, np. możemy dzielić tekst przy każdym wystąpieniu średnika lub przecinka. Jest to bardzo istotne przy przetwarzaniu informacji zaczerpniętych z plików lub innych źródeł (np. internetu). W wyniku tej funkcji powstaje nam lista zawierająca podzielone elementy, którą następnie można odpowiednio przetworzyć. Np. możemy pobrać jakieś dane z excela i wyłuskać z nich tylko to co potrzebujemy.

W przykładzie mamy kilka dat, oddzielonych średnikiem. Za pomocą metody split oddzielimy daty, a następnie za pomocą pętli wyciągniemy z każdej daty sam rok.

```
tekst = "12-05-2022;22-05-2022;18-06-2022"
rok = tekst.split(';')
for i in range(0, len(rok), 1):
    rok[i] = rok[i][-4:]
print(rok)
```

Istnieje wiele innych, użytecznych metod na wartościach typu string, które z łatwością można wyszukać w internecie w razie potrzeby. Wystarczy wpisać np. python string methods, aby znaleźć ich listę z przykładami zastosowania.

Zadanie: Średnia podanych ocen

Napisz program, który pobierze listę ocen oddzielonych spacją, a następnie obliczy ich średnią.

Podpowiedź: Spróbuj użyć metody rozdzielającej wartości tekstowe oraz zmapować tablicę na typ int.

Tablica znaków ASCII

Znaki ASCII odzwierciedlają znaki na klawiaturze i są często wykorzystywane w programowaniu. W tabeli przedstawione zostały podstawowe znaki, które można zapisać za pomocą typu char lub liczby (w tym przypadku dziesiętnej, ale oczywiście można je również przedstawić w popularnym zapisie szesnastkowym czy binarnym).

Aby przedstawić zapis liczbowy w postaci znaku, należy castować liczbę na typ chr. W przykładzie wyświetlimy liczbę 36 jako znak, co powinno, wg. tabeli, wyświetlić znak \$.

```
znak = 36
print(chr(znak))
```

Operacja odwrotna, czyli zamiana znaku na liczbę, odbywa się za pomocą metody ord()

```
znak = '$'
print(ord(znak))
```

Dec	Char	Dec	Char	Dec	Char	Dec	Char
32	[space]	58	:	84	T	110	n
33	!	59	;	85	U	111	o
34	"	60	<	86	V	112	p
35	#	61	[=]	87	W	113	q
36	\$	62	>	88	X	114	r
37	%	63	?	89	Y	115	s
38	&	64	@	90	Z	116	t
39	'	65	A	91	[117	u
40	(66	B	92	\	118	v
41)	67	C	93]	119	w
42	*	68	D	94	^	120	x
43	=+	69	E	95	_	121	y
44	,	70	F	96	`	122	z
45	-	71	G	97	a	123	{
46	.	72	H	98	b	124	
47	/	73	I	99	c	125	}
48	0	74	J	100	d	126	~
49	1	75	K	101	e	127	
50	2	76	L	102	f		
51	3	77	M	103	g		
52	4	78	N	104	h		
53	5	79	O	105	i		
54	6	80	P	106	j		
55	7	81	Q	107	k		
56	8	82	R	108	l		
57	9	83	S	109	m		

Funkcje

Funkcja to taki blok kodu, który odpowiada za wykonanie pewnej czynności. Uważa się, że funkcja powinna być bardzo krótka i rozwiązywać 1 konkretny problem.

Najpierw definiujemy funkcję, czyli piszemy słowo kluczowe `def`, a następnie podajemy nazwę funkcji.

Przechodzimy do wnętrza bloku funkcji i dodajemy kolejne instrukcje, które ta funkcja ma wykonywać.

Samo zdefiniowanie funkcji nie powoduje jej wywołania, tzn., że po jej stworzeniu ona sama się nie wykona. Kompilator ją rozpozna, ale nic z nią nie zrobi. W związku z powyższym musimy ją wywołać podając jej nazwę i nawias okrągły. Zobaczmy jak to działa na przykładzie funkcji, która wyświetla powitanie w konsoli.

```
# definicja funkcji
```

```
def WyswietlPowitanie():  
    print("Hellowina")
```

```
# wywołanie funkcji
```

```
WyswietlPowitanie()
```

Patrząc na utworzoną funkcję, możecie powiedzieć, że zwykłe wyświetlenie wyrazu "Hellowina" moglibyśmy uzyskać poprzez jedną linię kodu, a nie 3!!! Zatem, po co, to wszystko? O ile nasz program jest na tyle krótki i prosty, że tą wiadomość wyświetlamy tylko raz lub 2 to rzeczywiście, tworzenie osobnej funkcji nie ma sensu.

W bardziej rozbudowanych aplikacjach będziemy się ciągle spotykać z sytuacjami, gdzie zajdzie potrzeba wykorzystywania tych samych instrukcji z różnych miejsc w kodzie. Wtedy właśnie funkcje okazują się niezbędne. Wyobraź sobie, że to powitanie używasz w 10 różnych miejscach w kodzie, więc bez tworzenia funkcji musiałbyś 10-krotnie wstawić linię kodu:

`print("Hellowina")`. Po tygodniu jednak stwierdzasz, że ten wyraz nie pasuje, więc chcesz go zmienić na inny. Musisz to zrobić w 10 różnych miejscach. Myślisz, że łatwo zapamiętać te miejsca? I don't think so :) Dzięki funkcji, zmieniasz ten wyraz tylko raz i wszystkie miejsca w których ją wywołujesz dostosowują się automatycznie. I o to właśnie chodzi :)

```
# definicja funkcji
```

```
def WyswietlPowitanie():  
    print("Hellowina")
```

```
# wywołanie funkcji
```

```
WyswietlPowitanie()
```

Przykład: Losowe działanie matematyczne

Przykład funkcji, która generuje losowe działanie matematyczne.
Wielokrotne wywołanie funkcji za pomocą pętli.

```
import random

# definicja funkcji
def StworzDzialanie():
    l1 = random.randrange(1,21)
    l2 = random.randrange(5,21)
    print(str(l1) + " + " + str(l2) + " = ")

# wywołanie funkcji w pętli
for i in range(1,11):
    StworzDzialanie()
```

Funkcje z return

Funkcje mogą zwracać wartości, które można wykorzystać wewnątrz innych funkcji lub przypisać je do zmiennych.

Wewnątrz funkcji należy podać słowo kluczowe `return`, które określa co powinno być zwrócone.

Instrukcja `return` kończy działanie danego bloku kodu. Podanie samej instrukcji `return` powoduje przerwanie wykonywania bloku i wyjście do bloku zewnętrznego.

Przykład: Generowanie losowego hasła

Przykład funkcji, która generuje losowe hasło stworzone z 10 znaków. Wynikiem jej wywołania jest ciąg znaków, który możemy wyświetlić lub przekazać do kolejnych funkcji.

```
import random
def GenerujHaslo():
    haslo = ""
    for i in range(0,10):
        losowyZnak = chr(random.randrange(48,123,1))
        haslo += losowyZnak
    return haslo

print(GenerujHaslo())
```

Przykład: Test matematyczny z punktami

Przykład programu, który wyświetli pewną ilość zadań matematycznych. Ilość tych zadań może wynosić np.5, ale powinna być łatwo zmieniana. Po podaniu odpowiedzi, przechodzimy do kolejnego zadania. Przy udzieleniu poprawnej odpowiedzi dostajemy punkt. Po udzieleniu wszystkich odpowiedzi, na koniec testu pokazuje nam się wynik poprawnie udzielonych odpowiedzi.

```
import random
liczbaPunktow = 0
iloscZadan = 3
def GenerujLosowaLiczbe():
    return random.randrange(1,21,1)
def TworzZadanie():
    l1 = GenerujLosowaLiczbe()
    l2 = GenerujLosowaLiczbe()
    zadanie = f"{l1} + {l2} = "
    odp = int(input(zadanie))
    if(odp == l1 + l2):
        global liczbaPunktow
        liczbaPunktow += 1
for i in range(1,iloscZadan+1):
    TworzZadanie()
print(f"Poprawnych odpowiedzi: {liczbaPunktow} / {iloscZadan}")
```


Funkcja z parametrem

Funkcje stają się jeszcze bardziej uniwersalne, kiedy dołączymy do nich parametry.

Jako przykład stworzymy funkcję, która będzie wyświetlać, podaną jako argument, wiadomość.

Każde wywołanie funkcji, z innym argumentem, daje inny wynik w konsoli. Niby funkcja ta sama, a jednak staje się bardziej uniwersalna.

Podczas definiowania funkcji podajemy parametry, a przy wywołaniu funkcji podajemy argumenty.

```
def WyświetlWiadomosc(trescWiadomosci):  
    print(trescWiadomosci)
```

```
WyświetlWiadomosc("Hejka")
```

```
WyświetlWiadomosc("Dzień dobry")
```

```
WyświetlWiadomosc("Cześć")
```

Funkcja z parametrem domyślnym

Funkcja może również przyjmować domyślny parametr, który zostanie użyty jeśli nie podamy argumentu przy jej wywołaniu.

```
def WyswietlWiadomosc(trescWiadomosci = "Hello"):  
    print(trescWiadomosci)
```

```
WyswietlWiadomosc()
```

Funkcja z 2 parametrami

W tym przykładzie stworzymy funkcję z dwoma parametrami. Jej zadaniem będzie zwracanie losowej liczby z różnych przedziałów. Przedziały te będziemy sami zmieniać podczas wywołania funkcji, za pomocą argumentów.

Pamiętaj, że jeżeli funkcja została stworzona z dwoma parametrami, to podczas jej wywołania musimy podać 2 argumenty.

```
import random

def GenerujLiczbe(min,max):
    losowaLiczba = random.randrange(min,max,1)
    return losowaLiczba

# wyświetla liczbę od 10 do 20
print(GenerujLiczbe(10,21))

# wyświetla liczbę od 100 do 1000
print(GenerujLiczbe(100,1001))
```

Zmienne globalne i lokalne

W programowaniu mamy do czynienia z zakresem dostępności, tzw. scope.

Zmienne tworzone wewnątrz danego bloku kodu (np. funkcji) mają zasięg lokalny i żyją (są dostępne) tylko w tym bloku. Nie można się do nich dostać z zewnątrz.

Zmienne tworzone poza funkcją mają zasięg globalny i są dostępne wewnątrz wszystkich funkcji w danym skrypcie.

Modulo

Modulo to operator, który zwraca resztę z dzielenia.

Np. w działaniu $12/5$, otrzymujemy resztę 2. Zatem modulo $12/5$ wynosi 2.

Symbolem modulo jest znak %

```
print(12 % 5)
```

Modulo przydaje się, by sprawdzić czy dana liczba jest parzysta (bo wtedy $liczba \% 2 == 0$) lub czy dana liczba jest podzielna przez inną liczbę.

Przykład: Liczba podzielna przez 3

Przykład programu, który sprawdza czy podana liczba jest podzielna przez 3

```
liczba = int(input("Podaj liczbę całkowitą: "))  
if(liczba % 3 == 0):  
    print("Liczba jest podzielna przez 3")  
else:  
    print("Liczba nie jest podzielna przez 3")
```

Przykład: Gra - Zgadnij liczbę

Przykład gry , w której będziemy starali się odgadnąć losowo wybraną liczbę z przedziału od 1 do 100.

Program losuje liczbę, a następnie prosi użytkownika o wpisanie liczby. Po udzieleniu niepoprawnej odpowiedzi program podpowie czy zgadywana liczba jest większa od podanej czy mniejsza i ponownie poprosi o wpisanie liczby.

Po udzieleniu poprawnej odpowiedzi, gra się kończy, a użytkownik dostaje komunikat ile razy udzielił złej odpowiedzi.

```
import random
szukanaLiczba = random.randrange(1,101)
ostatniaOdpowiedz = 0
iloscProb = 0

def PodajOdpowiedz():
    global ostatniaOdpowiedz
    if(ostatniaOdpowiedz == 0):
        trescPytania = "Podaj liczbę od 1 do 100: "
    elif(ostatniaOdpowiedz > szukanaLiczba):
        trescPytania = "Podaj mniejszą liczbę: "
    else:
        trescPytania = "Podaj większą liczbę: "
    ostatniaOdpowiedz = int(input(trescPytania))
    global iloscProb
    iloscProb += 1

while(ostatniaOdpowiedz != szukanaLiczba):
    PodajOdpowiedz()
print(f"Brawo! Zgadłeś! Ilość prób: {iloscProb}")
```

Dictionary

Słownik (dictionary) pozwala tworzyć tablice asocjacyjne za pomocą pary key : value.

Dla każdego klucza, można przypisać konkretną wartość.

```
daneOsobowe = {  
    "imie" : "Ania",  
    "wiek" : 36  
}
```

#wypisanie pojedynczych wartości

```
print(daneOsobowe["imie"])  
print(daneOsobowe["wiek"])
```

#zmiana wartości w słowniku

```
daneOsobowe["wiek"] = 30  
print(daneOsobowe["wiek"])
```

Wypisanie kluczy / wartości danego słownika:

```
print(list(daneOsobowe.keys()))  
print(list(daneOsobowe.values()))
```



```
#Pobranie losowego klucza dictionary
import random
daneOsobowe = {
    "imie" : "Ania",
    "wiek" : 36
}

print(random.choice(
    list(daneOsobowe.keys())))
)
```

Tuple

Tuple pozwala zapisać wiele różnych wartości w jednej zmiennej. Tuple są niemutowalne, czyli niezmiennie. Raz ustawione elementy nie mogą być później nadpisane.

```
tupelek = 3,22,21
print(tupelek[0])
print(tupelek[1])
```

```
tuple1 = ("Mateusz", "Legnica", 120)
print(tuple1)
print(tuple1[0]) #element z indeksem 0
print(tuple1[1:3]) #element o indeksie 1 i 2
```

Set

Set to zbiór w którym każdy element jest unikatowy.

Elementy zbioru są niemutowalne, nie indeksowane i nie uporządkowane, przez co nigdy nie ma pewności w jakiej kolejności zostaną pobrane.

Set tworzy się za pomocą nawiasów klamrowych.

```
setOwocowy = {"apple", "banana", "cherry"}  
print(setOwocowy)
```

Set może zawierać różne typy danych, np:

```
set1 = {"abc", 34, True, 40, "male"}
```

Operacje na wielu zbiorach:

```
uczniowie = {"Ania", "Kasia", "Marek", "Zenek"}  
kursanci = {"Zosia", "Kasia", "Marek"}  
print(uczniowie - kursanci) #różnica zbiorów  
print(uczniowie & kursanci) #część wspólna zbiorów  
print(uczniowie | kursanci) #suma zbiorów
```

Class

Klasa jest instrukcją tworzenia obiektów. Nazwy klas zaczynamy z wielkiej litery.

Wewnątrz klasy znajduje się konstruktor (init), który wywołuje się automatycznie w chwili tworzenia nowego obiektu klasy.

W przykładzie obok stworzymy klasę osoby, która może zawierać takie dane jak imię i wiek, a następnie stworzymy obiekt tej klasy. Słowo self, oznacza aktualną instancję klasy. Należy je dodawać jako pierwszy argument konstruktora.

Zmienna (name, age) jest również określana jako pole.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# tworzenie obiektu klasy Person z danymi
p1 = Person("Marian", 36)

# wyświetlanie danych osoby p1
print(p1.name)
print(p1.age)
```

Klasy, poza polami, posiadają jeszcze metody (funkcje), które można wywoływać na tworzonych obiektach.

W przykładzie z osobą, stworzymy metodę, która wyświetla dane

osobowe.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def wyswietlDaneOsobowe(self):
        print(f"Imię:{self.name}, wiek:{self.age}")

p1 = Person("Marian", 36)
p1.wyswietlDaneOsobowe()
```

Data

Zarządzanie datami w python wymaga pobrania modułu datetime.

```
import datetime
```

Tworzymy zmienną z aktualną datą i godziną:

```
aktualnaDataGodzina = datetime.datetime.now()  
print(aktualnaDataGodzina)
```

Utworzony obiekt daty zawiera wiele informacji, które możemy przefiltrować za pomocą odpowiednich metod i argumentów, których część przedstawiona jest w tabeli poniżej, np:

```
print(aktualnaDataGodzina.strftime("%x"))
```

Tworzenie daty

W przypadku, gdy chcemy stworzyć datę, wywołujemy konstruktor datetime() i podajemy parametry opisujące rok, miesiąc i dzień, godzinę, minutę, itd..

Poniżej przykład stworzenia daty: 21 lipiec 2020, 18:55

```
import datetime  
dataUr = datetime.datetime(2020, 7, 21, 18, 55)  
print(dataUr.strftime("%d %B %Y, %H:%M"))
```

Argument	Opis	Przykład
%a	Dzień tygodnia, krótka wersja	Wed
%A	Dzień tygodnia, pełna wersja	Wednesday
%w	Dzień tygodnia jako numer 0-6, 0-Niedziela	3
%d	Dzień miesiąca 01-31	31
%b	Nazwa miesiąca, krótka wersja	Dec
%B	Nazwa miesiąca, długa wersja	December
%m	Miesiąc jako numer 01-12	12
%y	Rok, krótka wersja	18
%Y	Rok, pełna wersja	2018
%H	Godzina 00-23	17
%I	Godzina 00-12	5
%p	AM/PM	PM
%M	Minuta 00-59	41
%S	Sekunda 00-59	8
%j	Numer dnia w roku 001-366	365
%U	Numer tygodnia w roku, Niedziela pierwszym dniem, 00-53	52
%W	Numer tygodnia w roku, Poniedziałek pierwszym dniem, 00-53	52
%C	Wiek	20
%x	Lokalna wersja daty	12/31/18
%X	Lokalna wersja czasu	17:41:00

Praca z plikami

Tworzenie / zapisywanie do pliku

Tworzone programy często pracują z plikami, np. tekstowymi. Otwieranie plików przebiega za pomocą metody `open()`, która przyjmuje pewne argumenty: ścieżkę dostępu do pliku i sposób pracy na pliku. Możemy otworzyć plik tylko do odczytu (read - r), po to by dodać coś do istniejącej treści (append - a) lub zapisać coś od nowa (write - w). Możemy też sprecyzować czy plik jest zwykłym plikiem tekstowym (t) czy binarnym, np. grafiką (b). Pliki, na których pracujemy, powinny być zamknięte, kiedy skończymy operacje odczytywania / zapisywania. Obok przykład otwarcia do zapisu lub utworzenia, jeśli nie istnieje, pliku o nazwie `demo.txt` i zapisanie do niego aktualnej daty i godziny

```
import datetime
f = open("demo.txt", "w")
d = datetime.datetime.now()
f.write(d.strftime("%d %m %Y, %H:%M:%S"))
f.close()
```

lub otworzenie pliku i dodawanie kolejnych dat w nowej linii

```
import datetime
f = open("demo.txt", "a")
d = datetime.datetime.now()
f.write(d.strftime("%d %m %Y, %H:%M:%S"))
f.write('\n')
f.close()
```


Odczytywanie plików

Odczytywanie całej zawartości pliku tekstowego:

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

Odczytywanie jednej linii

```
f = open("demo.txt", "r")
print(f.readline())
f.close()
```

Odczytywanie określonej ilości znaków w pliku

```
f = open("demo.txt", "r")
print(f.read(10))
f.close()
```

Odczytywanie wszystkich linii w pętli

```
f = open("demo.txt", "r")
for line in f:
    print(line)
f.close()
```

Tworzenie / usuwanie plików i folderów

```
# usuń plik jeśli istnieje
```

```
import os
```

```
if os.path.exists("demo.txt"):
```

```
    os.remove("demo.txt")
```

```
else:
```

```
    print("Plik nie istnieje")
```

```
# utwórz folder
```

```
import os
```

```
if not os.path.exists("myfolder"):
```

```
    os.mkdir("myfolder")
```

```
else:
```

```
    print("Folder jest już utworzony")
```

```
# usuń folder jeśli istnieje i jest pusty
```

```
import os
```

```
if os.path.exists("myfolder"):
```

```
    os.rmdir("myfolder")
```

```
else:
```

```
    print("Folder nie istnieje")
```

Lista plików i folderów

```
#lista plików i folderów
```

```
import os
```

```
listaPlikowIFolderow = os.listdir()
```

```
print(listaPlikowIFolderow)
```

```
#lista plików w aktualnym folderze
```

```
import os
```

```
filenames = next(os.walk(os.getcwd()))[2]
```

```
print(filenames)
```

```
#lista plików o podanym rozszerzeniu
```

```
import glob
```

```
txtfiles = []
```

```
for file in glob.glob("*.txt"):
```

```
    txtfiles.append(file)
```

```
print(txtfiles)
```

```
#lista plików w aktualnym folderze zapisana w pliku txt
```

```
import os
```

```
filenames = next(os.walk(os.getcwd()))[2]
```

```
print(filenames)
```

```
f = open("lista.txt", "w")
```

```
for file in filenames:
```

```
    f.write(file)
```

```
    f.write('\n')
```

```
f.close()
```

Dodatkowe paczki z pip

Możemy w łatwy sposób zwiększyć możliwości programu za pomocą pobranych z internetu bibliotek w postaci paczek (packages). Obszerna baza dostępnych paczek znajduje się na stronie <https://pypi.org/> ale wiele ciekawych projektów można również znaleźć na githubie.

Paczki takie można pobrać i zainstalować manualnie lub skorzystać z tzw. pip, czyli python managera. W najnowszych wersjach pythona, pip jest instalowany automatycznie wraz z interpreterem. Możemy sprawdzić czy znajduje się on na naszym komputerze i jaką ma wersję uruchamiając w konsoli instrukcję:

```
pip --version
```

powinien pokazać się numer wersji lub poniższy komunikat, który oznacza, że manager pip nie jest zainstalowany.

'pip' is not recognized as an internal or external command, operable program or batch file.

Pythona można zainstalować pobierając plik ze strony <https://www.python.org/downloads/>

Wersja wideo, w jaki sposób zainstalować python:

<https://youtu.be/BjoBDW4U294>

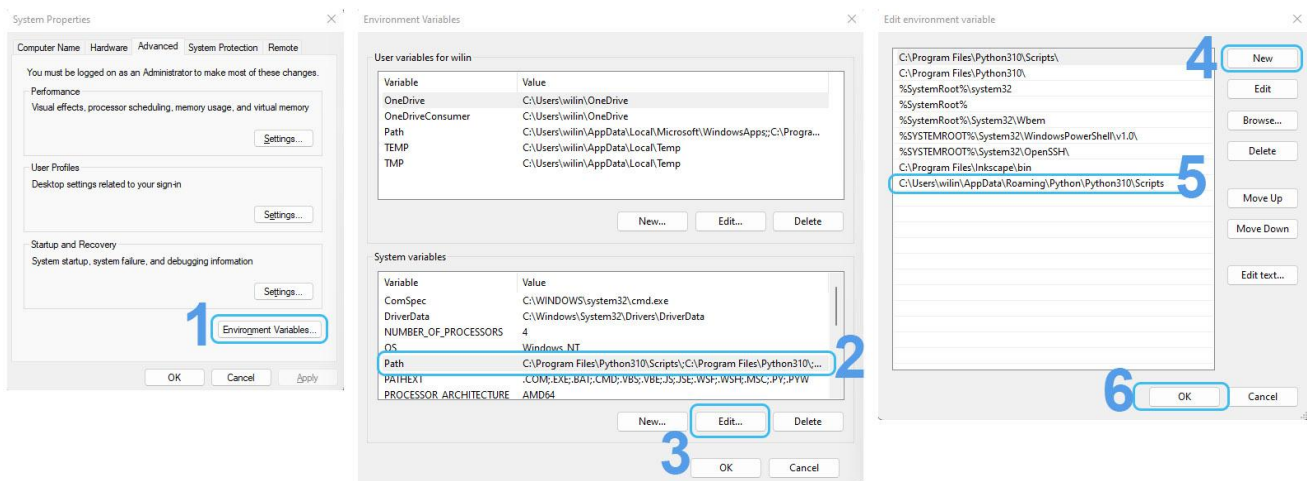
Wersja wideo, jak zainstalować przez pip i użyć paczki pyinstaller do konwersji skryptów .py do pliku wykonawczego .exe

<https://youtu.be/rcuxn6Xizbc>

Aby zainstalować paczkę za pomocą managera pip należy wpisać `pip install nazwaPaczki`, w tym przypadku pokażemy jak zainstalować paczkę `pyinstaller` do zamiany skryptu `.py` na `.exe`
`pip install pyinstaller`

Po instalacji należy dodać ścieżkę do zmiennych systemowych, która powinna się pokazać jako ostrzeżenie w trakcie instalacji. Może to wyglądać podobnie do linii poniżej (ale z pewnością będzie trochę inna ;)

`C:\Users\wilin\AppData\Roaming\Python\Python310\Scripts`



Zamiana skryptu `.py` na `.exe` następuje po wpisaniu instrukcji:
`pyinstaller --onefile main.py`

gdzie `main.py` to nazwa pliku, która może być inna