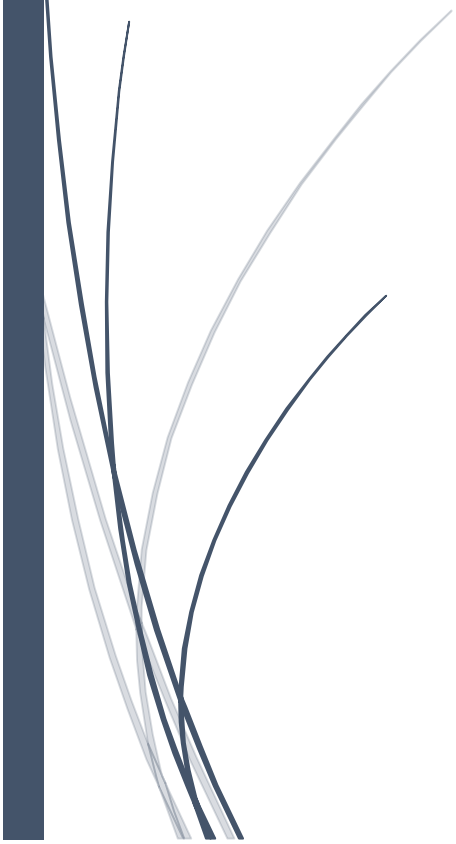
Python

Podstawy



Spis treści

Jambord 1

https://jamboard.google.com/d/1ATk96tQYgi9fvHUl5IVqgSjdpRtHO3153gGqC3EfHfk/edit?usp=sharing

Jambord 2

https://jamboard.google.com/d/1551K3zX14BQ9q_1twqW4hDMstdL1LRYKK4-fEP3YZwk/edit?usp=sharing

Jambord 3

https://jamboard.google.com/d/1E1Ehn8hi1efssPi48EcR1LoRanBXNa-0H93HxnM3jV4/edit?usp=sharing

1 Zmienne i typy

Zmienne to takie wydzielone miejsca w pamięci komputera, gdzie możesz przechowywać potrzebne ci dane. Python posiada kilka wbudowanych typów danych jak liczby całkowite, rzeczywiste itp. Przy tworzeniu zmiennej nie musisz podawać jaki typ danych będziesz przechowywać w zmiennej. Po prostu podajesz nazwę zmiennej i przypisujesz jej wartość. Z tego powodu mówi się, że Python jest językiem typowanym dynamicznie.

```
# tak tworzysz nowa zmienna

zmienna = 5
```

Omówimy kilka podstawowych typów zmiennych.

1.1 Liczby

Python obsługuje dwa typy liczbowe – liczby całkowite (ang. integers w skrócie int) i rzeczywiste (float).
Aby utworzyć nową zmienną całkowitą, użyj następującej składni:
calkowita = 7
Aby utworzyć nową zmienną rzeczywistą, użyj następującej składni:
rzeczywista = 7.5

1.1 Napisy

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

```
napis1 = 'witaj'

napis2 = "witaj"

napis 1 == napis2

# zwroci "True"
```

Różnica między nimi polega na tym, że dzięki użyciu podwójnego cudzysłowu można bez przeszkód używać apostrofów (podczas gdy oznaczałyby koniec napisu przy użyciu pojedynczego cudzysłowu).

```
napis = "Nie martw się o 'pojedyncze' cudzyslowy."
```

Na liczbach i stringach można wykonywać proste operacje:

Liczby:

```
jeden = 1
dwa = 2
trzy = jeden + dwa
print (trzy)
```

Stringi (łańcuchy)

```
witaj = "witaj"
swiecie = "swiecie"
witajswiecie = witaj + " " + swiecie
print (witaj swiecie)
```

Konkatenacja łańcuchów oznacza ich połączenie w celu utworzenia jednego nowego łańcucha. Operator + łączy dwa łańcuchy

Możliwe jest "jednoczesne" przypisanie wartości kilku różnym zmiennym w tej samej linijce jak w przykładzie poniżej.

```
a, b = 3, 4

print (a)

print (b)
```

Operacje, które mieszają ze sobą liczby i napisy nie są obsługiwane:

```
# To nie bedzie dzialac!

~~print (jeden + dwa + witaj)~~
```

2 Operatory arytmetyczne

Operatory arytmetyczne służą do wykonywania wszelkiego rodzaju działań na liczbach takich jak:

"+" - dodawanie,

"-" - odejmowanie,

"*" - mnożenie,

"/" - dzielenie,

"//" - dzielenie całkowite,

"%" - reszta z dzielenia dwóch liczb całkowitych,

"**" - potęgowanie.

```
a = 5

b = 3

c = a / b

print(c) # zostanie wypisane 1.6666666666666667

c = a // b

print(c) # zostanie wypisane 1 — część całkowita z dzielenia 5 przez 3
```

```
c = a ** b

print(c) # zostanie wypisany wynik 5 do potęgi 3 = 125
```

3 Operatory logiczne

Mają zastosowanie w miejscach, gdzie występują różnego rodzaju warunki — głównie w pętlach i instrukcjach warunkowych. Do operatorów logicznych zaliczamy:

or — lub logiczne,

and — i logiczne,

not — zaprzeczenie.

Lub logiczne zwraca prawdę, gdy przynajmniej jeden z warunków jest prawdziwy, w przeciwnym razie zwraca fałsz.

Logiczne i zwraca prawdę w przypadku, gdy wszystkie warunki są prawdziwe, w przeciwnym razie zwraca fałsz.

Logiczne nie zaprzecza otrzymaną wartość z True na False oraz z False na True .

Prześledźmy przykłady:

```
n = 5

k = 7

print(n > 3 or k > 10) # zostanie wypisane True, ponieważ przynajmniej jeden warunek musi mieć wartość logiczną True (musi być prawdziwy)

print(n > 3 and k > 10) # zostanie wypisane False, ponieważ wszystkie warunki muszą mieć wartość logiczną True

print(not(n > 3 and k > 10)) # zostanie wypisane True, ponieważ zaprzeczamy wartość logiczną False, którą otrzymujemy z warunków (n > 3 and k > 10)
```

4 Operatory relacyjne

Operatory relacyjne stosujemy w sytuacje, gdzie jest potrzeba porównania dwóch elementów. Najczęściej używane są w instrukcjach warunkowych i iteracyjnych. Wyróżniamy:

"<" - mniejszy

">" - większy

"<=" - mniejszy równy

">=" - większy równy

"==" - równy

"!=" - nie równy

Warto wspomnieć, że Python pozwala na zapis tego typu:

```
a = 5
```

```
if 4 <= a <= 6:
    print("wartość zmiennej a mieści się w przedziale [4..6]")
```

Przykładowe zastosowanie:

```
a = 5
b = 5
if a == b:
    print("wartości zmiennych a i b są takie same")
a = 11
if a != b:
    print('wartości zmiennych a i b są różne')
```

5 Operatory przypisania

Przypisanie polega na nadaniu wartości dla zmiennej znajdującej się po lewej stronie, wartości znajdującej się po stronie prawej. Dla takiej zmiennej można bezpośrednio nadać wartość, przekazać wartość z innej zmiennej lub wartość może zostać nadana poprzez wcześniejsze wykonanie pewnych operacji. Podstawowym operatorem przypisania jest "=".

```
a = 45 # przypisanie liczby 45 do zmiennej a
print(a)
b = a # przypisanie do zmiennej b wartości przechowywanej przez zmienną a, czyli liczbę 45
print(b)
a = 2 ** 10 # przypisanie do zmiennej a wyniku działania 2 do potęgi 10 = 1024
print(a)
```

Podobnie jak w C++, przypisanie można wykonać, poprzedzając je pewną operacją.

Operację przypisania połączoną z pewną operacją matematyczną na wartości zmiennej można przedstawić w następujący sposób:

zmienna [operator matematyczny][=] wartość;

Aby lepiej zrozumieć powyższą notację, prześledźmy przykłady:

```
a = 45
print(a)
a += 10 # zwiększenie wartości zmiennej a o 10 (teraz a = 55)
print(a)
a -= 50 # zmniejszenie wartości zmiennej a o 50 (teraz a = 5)
print(a)
a **= 3 # wykonanie potęgowania 5 do 3 i zapisanie wyniku w zmiennej a (teraz a = 125)
```

```
print(a)

a //= 30 # wykonanie dzielenia całkowitego 125 // 30 = 4 (teraz a = 4)

# itd.
```

6 Operatory identycznościowe

Operatory te określają, czy dwie zmienne przechowują ten sam obiekt. Wyróżniamy dwa operatory identycznościowe:

is

not is

```
x = "ala ma kota"

y = "ala nie ma kota"

if x is not y:

   print("Obiekty x i y to nie te same obiekty")

x = y

if x is y:

   print("Obiekty x i y to  te same obiekty")
```

Wyjście:

```
Obiekty x i y to nie te same obiekty

Obiekty x i y to  te same obiekty
```

Operatorem is (not is) możemy także sprawdzić, czy dany obiekt jest oczekiwanym typem np.:

```
x = "ala ma kota"

y = 25

if type(x) is str:

   print("obiekt x jest typem str")

if type(y) is int:

   print("obiekt x jest typem int")
```

Wyjście:

```
obiekt x jest typem str

obiekt x jest typem int
```

7 Operatory przynależności

Operatory tego typu sprawdzają, czy dany element zawiera się w podzbiorze wartości danego obiektu. Podobnie jak w operatorach identycznościowych mamy dwa takie operatory:

in

not in

```
x = "ala ma kota"

if "ma" in x:

   print("wyraz 'ma' występuje w ciągu'",x,"'")

y = [2, 3, 4, 100]

if 4 in y:

   print("Liczba 4 występuje w zbiorze",y)

else:

   print("Liczba 4 nie występuje w zbiorze",y)
```

Wyjście

wyraz 'ma' występuje w ciągu 'ala ma kota'

Liczba 4 występuje w zbiorze [2, 3, 4, 100]

8 Instrukcje print() i input()

Funkcja print() , służy do wyprowadzania danych z dowolnego programu w języku Python. Aby z niego skorzystać, przekaż listę argumentów oddzielonych przecinkami, które chcesz wydrukować do funkcji print() .print(12, "dzisiaj jest piatek", 123)

Aby wprowadzić dane do programu, używamy input().

Argumentem funkcji (podanym w nawiasach) jest tekst (ciąg zanków, string), który pojawi się na ekranie jako „zaproszenia" do wpisania znaków z klawiatury.

Jeżeli użyjemy funkcji input() w instrukcji przypisania, wpisany przez użytkownika ciąg znaków komputer zapamięta w zmiennej podanej po prawej stronie:

imie=input(„podaj imie „)

Aby rzutować (konwertować) ciąg cyfr na liczbę całkowitą, możemy użyć funkcji int() .

Na przykład int('23') podaje obiekt int o wartości 23 .

Aby wprowadzona za pomocą instrukcji input dana była zapamiętana jako liczba, dodajemy instrukcję, która zmieni ciąg znaków na liczbę:

a=int(input(„podaj liczbe ")

b=float(input(„podaj liczbe ")

9 Używanie sekwencji specjalnych w łańcuchach znaków

Sekwencje specjalne umożliwiają umieszczanie w łańcuchach znaków o szczególnym charakterze. Dają większą kontrolę nad wyświetlanym tekstem i elastyczność w jego tworzeniu. Sekwencje specjalne, które będziesz wykorzystywał, składają się ze znaku poprzedzonego lewym ukośnikiem. Wszystko to może wydawać się nieco tajemnicze, ale kiedy już zobaczysz kilka sekwencji specjalnych w działaniu, przekonasz się, jak łatwo ich używać.

9.1 Przesuwanie kursora tekstowego w prawo do najbliższego punktu tabulacji

Czasem chcesz odsunąć jakiś tekst od lewego marginesu, od którego standardowo zaczyna się wypisywanie. W procesorze tekstu mógłbyś użyć klawisza Tab. W przypadku łańcuchów znaków możesz wykorzystać sekwencję specjalną zastępującą znak tabulacji \t.

9.2 Wstawianie znaku nowego wiersza

Jedną z najprzydatniejszych sekwencji, jakie masz do dyspozycji, jest sekwencja nowego wiersza. Ma ona postać \n. Dzięki użyciu tej sekwencji możesz wstawiać w swoich łańcuchach znak nowego wiersza, mając na celu wyprowadzenie pustego wiersza w miejscach, gdzie to jest potrzebne. Możesz umieścić sekwencję nowego wiersza na samym początku łańcucha, aby oddzielić go od tekstu wypisanego poprzednio. Przykład:

print("\n Specjalne podziękowania należą się:")

9.3 Wstawianie znaku cudzysłowu

Wstawienie znaku cudzysłowu do łańcucha — nawet cudzysłowu tego samego typu co cudzysłów wyznaczający granice tego łańcucha — jest proste. Wystarczy użyć sekwencji \' w przypadku znaku pojedynczego cudzysłowu oraz \" w przypadku znaku podwójnego Używanie sekwencji specjalnych w łańcuchach znaków 39 cudzysłowu. Sekwencje te znaczą „wstaw w tym miejscu znak cudzysłowu” i żadna z nich nie zostanie błędnie potraktowana przez komputer jako znacznik końca Twojego łańcucha.

Przykład umieszczenia obydwu rodzajów cudzysłowów w jednym wierszu tekstu:

print("Henry\'emu \'Wielkiemu\', który nigdy nie mówi \"nie da się\".")

Sekwencja	Opis
\\	Lewy ukośnik. Powoduje wyświetlenie jednego lewego ukośnika.
\'	Pojedynczy cudzysłów. Powoduje wyświetlenie znaku pojedynczego cudzysłowu.
\'	Podwójny cudzysłów. Powoduje wyświetlenie znaku podwójnego cudzysłowu.
\a	Alarm. Wywołuje sygnał dzwonka systemowego.
\n	Nowy wiersz. Przenosi kursor na początek następnego wiersza.
\t	Tabulator poziomy. Przesuwa kursor w prawo do najbliższego punktu tabulacji.

9.4 Używanie znaku kontynuacji wiersza

Na ogół umieszczasz jedną instrukcję w każdym wierszu kodu. Ale nie jest to konieczne. Możesz rozciągnąć pojedynczą instrukcję na kilka wierszy. Wszystko, co musisz zrobić, to użyć znaku kontynuacji wiersza \ (który jest właśnie lewym ukośnikiem), tak jak w poniższym kodzie.

```
print("\nTen łańcuch " + "może nie " + "sprawiać wiel" + "kiego wrażenia. " \
      + "Ale " + "pewnie nie wiesz," + " że jest\n" + "to jeden napraw" \
      + "d" + "ę" + " długi łańcuch, utworzony przez konkatenację " \
      + "aż " + "dwudziestu dwu\n" + "różnych łańcuchów i rozbity na " \
      + "sześć wierszy." + " Jesteś pod" + " wrażeniem tego faktu?\n" \
      + "Dobrze, ten " + "jeden " + "długi" + " łańcuch właśnie się skończył!")
```

Możesz wstawić go wszędzie, gdzie normalnie zastosowałbyś odstęp (choć nie wewnątrz łańcucha), by kontynuować pisanie instrukcji w następnym wierszu. Komputer będzie działał tak, jakby to był jeden długi wiersz kodu. Z punktu widzenia komputera długość wiersza programu jest nieistotna, ale jest ważna dla ludzi. Jeśli jakiś wiersz kodu wydaje Ci się zbyt długi lub uważasz, że byłby bardziej czytelny w postaci kilku wierszy, użyj znaku \, aby go podzielić na części.

9.5 Powielanie łańcuchów

Przykład:
print("Lody!" * 8)

W tym wierszu tworzony jest nowy łańcuch, "Lody!Lody!Lody!Lody!Lody!Lody!Lody!Lody!", który zostaje wyświetlony na ekranie. Jest to łańcuch "Lody!" powtórzony 8 razy. Podobnie jak operator konkatenacji, operator powielania (*) jest dość intuicyjny. To tak, jakbyś mnożył łańcuch. Aby powielić łańcuch, wystarczy umieścić między nim a liczbą powtórzeń operator *.

9.6 Definiowanie łańcucha końcowego funkcji print

Domyślnie funkcja print() wypisuje jako wartość końcową znak nowego wiersza. To oznacza, że kolejne wywołanie funkcji print() wyświetliłoby tekst w następnym wierszu. Na ogół jest to zgodne z Twoim oczekiwaniem, niemniej jednak masz możliwość zdefiniowania swojego własnego łańcucha, który zostanie wypisany na końcu tekstu. Możesz na przykład zdefiniować go tak, że kiedy wywołasz funkcję print(), ostatnim wypisanym znakiem będzie spacja (zamiast znaku nowego wiersza). To by oznaczało, że kolejna instrukcja print() rozpoczęłaby wypisywanie wartości bezpośrednio po tej spacji.
Przykład:
print("Oto", end=" ")
print("on...")

Ten kod wypisuje tekst „Oto on...” w jednym wierszu. Efekt ten uzyskuję przez zdefiniowanie spacji jako wartości parametru end funkcji print() za pomocą kodu end=" ". W swojej własnej instrukcji print() możesz zdefiniować łańcuch, który ma być wypisany jako ostatnia wartość. Możliwość zdefiniowania swojego własnego łańcucha, który ma być wypisany przez instrukcję print() na końcu, daje Ci większą elastyczność w sposobie formatowania Twoich danych wyjściowych.

10 Instrukcja warunkowa if

Przykład:

```
wiek=int(input('Ile masz lat?'))
 if wiek>=18:
    print('Możesz wejść')
else:
    print('Za młody')
```

Do podejmowania decyzji w programowaniu służy instrukcja warunkowa if.
Program zapamięta wprowadzoną liczbę pod nazwą wiek i sprawdzi:
jeżeli wiek jest większy lub równy 18 to Napisz Możesz wejść W przeciwnym razie Napisz Za młody

Blok kodu podany po instrukcji if zostanie wykonany wtedy, gdy wyrażenie warunkowe będzie prawdziwe. W przeciwnym przypadku blok kodu zostanie zignorowany.
Część else jest przydatna, jeśli wyrażenie warunkowe było fałszywe.
Wszystkie instrukcje w bloku kodu muszą być wcięte względem instrukcji if (głębokość wcięcia nie ma znaczenia, ale trzymajcie jeden standard, np. 4 wcięcia). W ten sposób Python rozpoznaje, które instrukcje ma wykonać po sprawdzeniu prawdziwości wyrażenia.
Ważne są dwukropki na końcu linijki z if i po else.

11 Instrukcja elif

Instrukcja if else to alternatywa logiczna, która daje dwa możliwe wyniki.
W wielu przypadkach będziemy potrzebować programu, który ocenia więcej niż dwa możliwe wyniki.
W tym celu użyjemy instrukcji elif.

```
if <wyrażenie jest prawdziwe>:
    <wykonaj instrukcję 1>
elif <poprzednie wyrażenie jest fałszywe ale to jest prawdziwe>:
    <wykonaj instrukcję 2>
else:
     <wykonaj instrukcję 3>
```

Przykład
Sprawdzamy konta bankowego. Potrzebujemy trzech oddzielnych komunikatów dla trzech różnych sytuacji:

1. saldo jest mniejsze od zera
2. saldo jest równe zeru
3. saldo jest większe od zera

Instrukcja elif zostanie umieszczona między instrukcją if a instrukcją else w następujący sposób:

```
if saldo < 0:
    print("Saldo jest ujemne, dodaj środki teraz lub zostaniesz obciążony karą .")
elif saldo == 0:
    print("Saldo rowne 0, dodaj srodki.")
else:
    print("Saldo jest dodatnie.")
```

Teraz istnieją trzy możliwe rezultaty, które mogą wystąpić po uruchomieniu programu:

1. Jeśli zmienna saldo jest równa 0, otrzymamy wynik z instrukcji elif (Saldo jest równe 0, dodaj środki).
2. Jeśli zmienna saldo jest liczbą dodatnią, otrzymamy wynik z instrukcji else (Twoje saldo jest dodatnie).
3. Jeśli zmienna saldo jest liczbą ujemną, wyjściem będzie ciąg znaków z instrukcji if (Saldo jest ujemne, dodaj środki teraz lub zostaniesz obciążony karą).

Co jeśli chcemy mieć więcej niż trzy możliwości? Możemy to zrobić, wprowadzając do naszego kodu więcej niż jedną instrukcję elif.

12 Pętla for

Powiedzmy, że chcemy wypisać liczby od 1 do 10. Możemy to zrobić oczywiście tak:
print('1')
print('2')
…
print('10')
Możliwe do zrobienia? Możliwe. Ale jest to klasyczny przykład, gdzie do tego celu powinniśmy użyć pętli.

Zanim wytłumaczymy w szczegółach jej działanie, zobaczmy ją w akcji:

```
for i in range(1,11):
    print(i)
```

```
1
2
3
4
5
6
7
8
9
10
```

Po instrukcji **for**, Python wie, że ma wykonać następujący po niej kod, wielokrotnie.

1. Za każdym powtórzeniem, zmienna 'i', będzie zawierać inną liczbę. Zmienna 'i', oczywiście może się nazywać inaczej.
2. Wartości jakie będzie przyjmować zmienna 'i', będą w zakresie określonym przez funkcję range. Od 1 do 10.
3. Funkcja 'print', odwołuje się do zmiennej i, po czym wyświetla jej zawartość

Funkcja range, jest czymś co często występuje w pętli for, ale nie jest ona jedyna. Za chwilę zobaczymy inne przykłady.

Liczenie do przodu

```
for i in range(10):
    print(i, end=" ")
```

efekt:

0 1 2 3 4 5 6 7 8 9

Liczenie do tyłu

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

efekt

10 9 8 7 6 5 4 3 2 1

Liczenie co pięć

```
for i in range(0, 50, 5):
    print(i, end=" ")
```

efekt

0 5 10 15 20 25 30 35 40 45

Przykłady:

```
lista = [1,2,'abc',3,4]

for i in lista:
    print (i)
```

```
1
2
abc
3
4
```

Pętla for, 'bierze' po kolei elementy naszej listy, po czym przekazuje je do zmiennej i.

```
tekst = "Analityk"

for ttt in tekst:
    print (ttt)
```

```
A
n
a
l
i
t
y
k
```

Podobnie ma to miejsce w przypadku zmiennej tekstowej. Zmienna ttt, zawiera po kolei, wszystkie litery słowa 'Analityk'

13 Pętla while

While to kolejne polecenie, które jest informacją dla języka Python, że powinien powtarzać następujący po nim kod. Za pomocą tej pętli, można osiągnąć podobne efekty jak za pomocą pętli for, i często stosowane są zamiennie. Spójrzmy na przykład:

```
krok = 0

while krok < 10:
    print (krok)
    krok = krok + 1
```

```
0
1
2
3
4
5
6
7
8
9
```

Tak długo jak warunek po instrukcji while, jest spełniony, Python powtarza dany kod. W tym przypadku samodzielnie zwiększamy zmienną krok oraz ją wypisujemy.
Jeśli format pętli while wydaje się znajomy, nie dzieje się to bez przyczyny. Charakteryzuje ją uderzające podobieństwo do jej kuzynki — instrukcji if. Jedyna różnica polega na tym, że słowo if jest zastąpione przez while. A te podobieństwa nie są powierzchowne. W obu typach instrukcji, jeśli warunek jest spełniony, blok kodu (w przypadku pętli nazywany czasem ciałem pętli) jest wykonywany. Lecz w instrukcji while komputer sprawdza warunek i wykonuje blok kodu raz po raz, dopóki warunek nie okaże się fałszywy. Właśnie dlatego nazywamy ją pętlą.

13.1 Inicjalizacja zmiennej odgrywającej rolę wartownika

Pętle while są często kontrolowane przez wartownika — zmienną używaną w warunku i porównywaną z jakąś inną wartością lub wartościami. Możesz myśleć o swojej zmiennej w roli wartownika jak o strażniku pomagającym utworzyć barierę wokół bloku pętli while.
Ważne, aby zainicjalizować swojego wartownika. W większości przypadków zmienne odgrywające rolę wartowników są inicjalizowane bezpośrednio przed samą pętlą.

Pułapka
Jeśli zmienna (wartownik) nie będzie miała przypisanej wartości w momencie określania wartości warunku, Twój program wygeneruje błąd. Zwykle dobrym pomysłem jest inicjalizowanie zmiennych (wartowników) poprzez nadanie im pewnego typu pustej wartości.
Gdy już ustaliłeś swój warunek, zainicjowałeś swoją zmienną (wartownika) i jesteś pewien, że w pewnych warunkach blok pętli zostanie wykonany, skonstruowałeś własną działającą pętlę. W następnej kolejności upewnij się, że pętla w pewnym momencie się zakończy.
Jeśli napisałeś pętlę, która nigdy się nie zatrzymuje, utworzyłeś pętlę nieskończoną.

Witaj w klubie. Prędzej czy później wszyscy programiści utworzyli przypadkowo pętlę nieskończoną, a potem przyglądali się, jak ich program utknął, robiąc to samo bez końca.
Przykład 1:

```
licznik = 0
while licznik <= 10
        print(licznik)
```

Przykład 2:

```
while True:
    print("hi")
```

```
hi
hi
hi
hi
hi
hi
hi
hi
hi
```

Po instrukcji while, następuje warunek, który zawsze jest prawdziwy, w związku z czym, pętla powtarza się nieustannie. W ten sposób, można, przykładowo, w kodzie pętli sprawdzać różnego rodzaju sytuacje, które jak się spełnią, to dopiero taką pętle przerwać. Jak na przykład:

- czekać na zdarzenia użytkownika
- sprawdzać czy pojawił się jakiś plik
- sprawdzać czy pojawiły się jakieś dane
- i wiele innych

a następnie z takiej pętli, możemy 'wyjść' za pomocą polecenia 'break'. Spójrzmy na przykład:

```
licznik = 0

while True:
    licznik = licznik + 1
    print (licznik)
    if licznik > 7:
        print ("Wychodzimy z pętli za pomocą break")
        break

print("Pętla while została zakończona")
```

```
1
2
3
4
5
6
7
8
Wychodzimy z pętli za pomocą break
Pętla while została zakończona
```

13.2 break

Polecenie 'break', jest informację dla języka Python, aby natychmiast przerwać wykonywanie pętli. Może być stosowane w pętli **while** – tak jak powyżej, ale również w pętli **for**.

Możemy przykładowo, wykorzystać to do przetwarzania listy w pętli for, po czym wyjść z niej, gdy zostanie znaleziona konkretna wartość. Spójrzmy:

```
lista = ['A', 1, 2, 'abc', 'x']

for element in lista:

    if element == 2:
        break
    else:
        print(element)

print ("Pętla została zakończona")
```

```
A
1
Pętla została zakończona
```

Podsumowując. Pętle są jedną z najczęściej stosowanych aspektów programowania. Możemy przeglądać tekst, listy czy też inne zbiory danych.

14 Listy

Lista to zmienna zawierająca zbiór elementów. Elementami listy mogą być wszystkie dostępne w Python typy danych. Listy są jednym z najczęściej używanych typów danych.

https://pogromcykodu.pl/struktury-danych-w-pythonie-listy-i-krotki/

https://oprojektowaniu.pl/python-dla-inzynierow-listy/

14.1 Definiowanie list

Przykład. ***Definiowanie list***

```
>>> li = ["a", "b", "mpilgrim", "z", "przykład"]     #(1)
>>> li
['a', 'b', 'mpilgrim', 'z', 'przykład']
>>> li[0]                                            #(2)
'a'
>>> li[4]                                            #(3)
'przykład'
```

1. Najpierw zdefiniowaliśmy listę pięcioelementową. Zauważmy, że lista zachowuje swój oryginalny porządek i nie jest to przypadkowe. Lista jest uporządkowanym zbiorem elementów ograniczonym nawiasem kwadratowym.
2. Lista może być używana tak jak tablica zaczynająca się od 0. Pierwszym elementem niepustej listy o nazwie li jest zawsze li[0].
3. Ostatnim elementem pięcioelementowej listy jest li[4], ponieważ indeksy są liczone zawsze od 0.

Przykład. ***Ujemne indeksy w listach***

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'przykład']
>>> li[-1]                                    #(1)
'przykład'
>>> li[-3]                                    #(2)
'mpilgrim'
```

1. Za pomocą ujemnych indeksów odnosimy się do elementów idących od końca do początku tzn. li[-1] oznacza ostatni element, li[-2] przedostatni, li[-3] odnosi się do 3 od końca elementu itd. Ostatnim elementem niepustej listy jest zawsze li[-1].
2. Jeśli ciężko ci zrozumieć o co w tym wszystkim chodzi, możesz pomyśleć o tym w ten sposób: li[-n] == li[len(li) - n]. len to funkcja zwracająca ilość elementów listy. Tak więc w tym przypadku li[-3] == li[5 - 3] == li[2].

Przykład. ***Wycinanie list***

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'przyklad']
>>> li[1:3]                                   #(1)
['b', 'mpilgrim']
>>> li[1:-1]                                  #(2)
['b', 'mpilgrim', 'z']
>>> li[0:3]                                   #(3)
['a', 'b', 'mpilgrim']
```

1. Możesz pobrać podzbiór listy, który jest nazywany "wycinkiem" (ang. *slice*), poprzez określenie dwóch indeksów. Zwracaną wartością jest nowa lista zawierająca wszystkie elementy z listy rozpoczynające się od pierwszego wskazywanego indeksu (w tym przypadku li[1]) i idąc w górę kończy na drugim wskazywanym indeksie, nie dołączając go (w tym przypadku li[3]). Kolejność elementów względem wcześniejszej listy jest także zachowana.
2. Możemy także podać ujemną wartość któregoś indeksu. Wycinanie wtedy także dobrze zadziała. Jeśli to pomoże, możemy pomyśleć tak: czytamy listę od lewej do prawej, pierwszy indeks określa pierwszy potrzebny element, a drugi określa element, którego nie chcemy. Zwracana wartość zawiera wszystko między tymi dwoma przedziałami.
3. Listy są indeksowane od zera tzn. w tym przypadku li[0:3] zwraca pierwsze trzy elementy listy, rozpoczynając od li[0], a kończąc na li[2], ale nie dołączając li[3].

Przykład. ***Skróty w wycinaniu***

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'przykład']
>>> li[:3]                                    #(1)
```

```
['a', 'b', 'mpilgrim']

>>> li[3:]                                          #(2)  (3)
['z', 'przykład']

>>> li[:]                                           #(2)  (4)
['a', 'b', 'mpilgrim', 'z', 'przykład']
```

1. Jeśli lewy indeks wynosi 0, możemy go opuścić, wartość 0 jest domyślna. li[:3] jest tym samym, co li[0:3] z poprzedniego przykładu.
2. Podobnie, jeśli prawy indeks jest długością listy, możemy go pominąć. Tak więc li[3:] jest tym samym, co li[3:5], ponieważ lista ta posiada pięć elementów.
3. Zauważmy pewną symetryczność: w pięcioelementowej liście li[:3] zwraca pierwsze 3 elementy, a li[3:] zwraca dwa ostatnie (a w sumie 3 + 2 = 5). W ogólnym przypadku li[:n] będzie zwracał zawsze pierwsze n elementów (a jeśli n będzie większe od ilości elementów w liście to tyle, ile ma lista), a li[n:] pozostałą liczbę, bez względu na wielkość listy (n może być większe od ilości elementów w liście).
4. Jeśli obydwa indeksy zostaną pominięte, wszystkie elementy zostaną dołączone. Nie jest to jednak to samo, co oryginalna lista li. Jest to nowa lista, która posiada wszystkie takie same elementy. li[:] tworzy po prostu kompletną kopię listy.

14.2 Dodawanie elementów do listy

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'przykład']
>>> li.append("nowy")                                #(1)
>>> li
['a', 'b', 'mpilgrim', 'z', 'przykład', 'nowy']
>>> li.insert(2, "nowy")                             #(2)
>>> li
['a', 'b', 'nowy', 'mpilgrim', 'z', 'przykład', 'nowy']
>>> li.extend(["dwa", "elementy"])                   #(3)
>>> li
['a', 'b', 'nowy', 'mpilgrim', 'z', 'przykład', 'nowy', 'dwa',
'elementy']
```

1. Dodajemy pojedynczy element do końca listy za pomocą metody append.
2. Za pomocą insert wstawiamy pojedynczy element do listy. Numeryczny argument jest indeksem, pod którym ma się znaleźć wstawiana wartość; reszta elementów, która znajdowała się pod tym indeksem lub miała większy indeks, zostanie przesunięta o jeden indeks dalej. Zauważmy, że elementy w liście nie muszą być unikalne i mogą się powtarzać; w przykładzie mamy dwa oddzielne elementy z wartością 'nowy' -- li[2] i li[6].
3. Za pomocą extend łączymy listę z inną listą. Nie możemy wywołać extend z wieloma argumentami, trzeba ją wywoływać z pojedynczym argumentem -- listą. W tym przypadku ta lista ma dwa elementy.

Przykład. *Różnice między* `extend` *a* `append`

```
>>> li = ['a', 'b', 'c']
>>> li.extend(['d', 'e', 'f'])              #(1)
>>> li
['a', 'b', 'c', 'd', 'e', 'f']
>>> len(li)                                 #(2)
6
>>> li[-1]
'f'
>>> li = ['a', 'b', 'c']
>>> li.append(['d', 'e', 'f'])              #(3)

>>> li

['a', 'b', 'c', ['d', 'e', 'f']]
>>> len(li)                                 #(4)
4

>>> li[-1]
['d', 'e', 'f']
```

1. Listy posiadają dwie metody -- extend i append, które wyglądają na to samo, ale w rzeczywistości są całkowicie różne. extend wymaga jednego argumentu, który musi być listą i dodaje każdy element z tej listy do oryginalnej listy.
2. Rozpoczęliśmy z listą trójelementową ('a', 'b' i 'c') i rozszerzyliśmy ją o inne trzy elementy ('d', 'e' i 'f') za pomocą extend, tak więc mamy już sześć elementów.
3. append wymaga jednego argumentu, który może być dowolnym typem danych. Metoda ta po prostu dodaje dany element na koniec listy. Wywołaliśmy append z jednym argumentem, który jest listą z trzema elementami.
4. Teraz oryginalna lista, pierwotnie zawierająca trzy elementy, zawiera ich cztery. Dlaczego cztery? Ponieważ ostatni element przed chwilą do niej wstawiliśmy. Listy mogą wewnątrz przechowywać dowolny typ danych, nawet inne listy. Nie używajmy append, jeśli zamierzamy listę rozszerzyć o kilka elementów.

14.3 Przeszukiwanie list

```
>>> li
['a', 'b', 'nowy', 'mpilgrim', 'z', 'przykład', 'nowy', 'dwa',
'elementy']
>>> li.index("przykład")                                    #(1)
5
>>> li.index("nowy")                                        #(2)
2
>>> li.index("c")                                           #(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.index(x): x not in list
>>> "c" in li                                               #(4)
False
```

1. index znajduje pierwsze wystąpienie pewnej wartości w liście i zwraca jej indeks.
2. index znajduje pierwsze wystąpienie wartości w liście. W tym przykładzie, 'nowy' występuje dwa razy -- w li[2] i li[6], ale metoda index będzie zawsze zwracać pierwszy indeks, czyli 2.
3. Jeśli wartość nie zostanie znaleziona, Python zgłosi wyjątek. Takie zachowanie nie jest często spotykane w innych językach, w wielu językach w takich przypadkach zostaje zwrócony niepoprawny indeks. Taka reakcja Pythona jest dobrym zachowaniem, ponieważ umożliwia szybkie wychwycenie błędu w kodzie, a dzięki temu program nie będzie błędnie działał operując na niewłaściwym indeksie.
4. Aby sprawdzić czy jakaś wartość jest w liście używamy słowa kluczowego in, który zwraca True, jeśli wartość zostanie znaleziona lub False jeśli nie.

Przed wersją 2.2.1 Python nie posiadał oddzielnego boolowskiego typu danych. Aby to zrekompensować, Python mógł interpretować większość typów danych jako bool (czyli wartość logiczną np. w instrukcjach if) według poniższego schematu:

- 0 jest fałszem; wszystkie inne liczby są prawdą.
- Pusty łańcuch znaków ("") jest fałszem, wszystkie inne są prawdą.
- Pusta lista ([]) jest fałszem; wszystkie inne są prawdą.
- Pusta krotka (()) jest fałszem; wszystkie inne są prawdą.
- Pusty słownik ({}) jest fałszem; wszystkie inne są prawdą.

Wszystkie powyższe punkty stosowane są w Pythonie 2.2.1 i nowszych, ale obecnie można także używać typu logicznego bool, który może przyjmować wartość True (prawda) lub False (fałsz). Zwróćmy uwagę, że wartości te, tak jak cała składnia języka Python, są wrażliwe na wielkość liter.

14.4 Usuwanie elementów z listy

```
>>> li
['a', 'b', 'nowy', 'mpilgrim', 'z', 'przykład', 'nowy', 'dwa',
'elementy']
>>> li.remove("z")                                          #(1)
>>> li
['a', 'b', 'nowy', 'mpilgrim', 'przykład', 'nowy', 'dwa',
'elementy']
>>> li.remove("nowy")                                        #(2)

>>> li
['a', 'b', 'mpilgrim', 'przykład', 'nowy', 'dwa', 'elementy']
>>> li.remove("c")
#(3)
Traceback (most recent call last):

  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
>>> li.pop()                                                 #(4)
'elementy'
>>> li

['a', 'b', 'mpilgrim', 'przykład', 'nowy', 'dwa']
```

1. remove usuwa pierwszą występującą wartość w liście.
2. remove usuwa tylko pierwszą występującą wartość. W tym przypadku 'nowy' występuje dwa razy, ale li.remove("nowy") usuwa tylko pierwsze wystąpienie.
3. Jeśli wartość nie zostanie znaleziona w liście, Python wygeneruje wyjątek. Naśladuje on w takich sytuacjach postępowanie metody index.
4. pop jest ciekawą metodą, która wykonuje dwie rzeczy: usuwa ostatni element z listy i zwraca jego wartość. Metoda ta różni się od li[-1] tym, że li[-1] zwraca jedynie wartość, ale nie zmienia listy, a od li.remove(value) tym, że li.remove(value) zmienia listę, ale nie zwraca wartości.

14.5 Używanie operatorów na listach

```
>>> li = ['a', 'b', 'mpilgrim']
>>> li = li + ['przykład', 'nowy']            #(1)
>>> li
['a', 'b', 'mpilgrim', 'przykład', 'nowy']
>>> li += ['dwa']                             #(2)
>>> li
['a', 'b', 'mpilgrim', 'przykład', 'nowy', 'dwa']
>>> li = [1, 2] * 3                           #(3)
>>> li
[1, 2, 1, 2, 1, 2]
```

1. Aby połączyć dwie listy, można też skorzystać z operatora +. Za pomocą lista = lista + innalista uzyskujemy ten sam wynik, co za pomocą list.extend(innalista), ale operator + zwraca nową listę, podczas gdy extend zmienia tylko istniejącą listę. Ogólnie extend jest szybszy, szczególnie na dużych listach.
2. Python posiada także operator +=. Operacja li += ['dwa'] jest równoważna li.extend(['dwa']). Operator += działa zarówno na listach, łańcuchach znaków jak i może być nadpisany dla dowolnego innego typu danych.
3. Operator * zwielokrotnia podaną listę. li = [1, 2] * 3 jest równoważne z li = [1, 2] + [1, 2] + [1, 2], które łączy trzy listy w jedną.

14.6 Funkcje wbudowane które pozwalają wykonywać operacje na elementach listy

W Pythonie istnieją cztery wbudowane funkcje, które pozwalają wykonywać operacje na elementach listy. Oto opis tych funkcji:

len(): Zwraca liczbę elementów listy
my_list = [1, 2, 3, 4, 5]

długosc=len(my_list)
print(dlugosc)
#5
max(): Zwraca największy element z listy.
my_list = [1, 2, 3, 4, 5]
max_value = max(my_list)
# wynik: 5

min(): Zwraca najmniejszy element z listy.
my_list = [1, 2, 3, 4, 5]

min_value = min(my_list)
# wynik: 1

sum(): Zwraca sumę wszystkich elementów z listy.
my_list = [1, 2, 3, 4, 5]
sum_value = sum(my_list)
# wynik: 15

Wszystkie te funkcje mogą również działać na listach zawierających liczby zmiennoprzecinkowe.
my_list = [1.2, 2.5, 3.8, 4.1, 5.6]
max_value = max(my_list)
min_value = min(my_list)
sum_value = sum(my_list)
# wynik: max_value = 5.6, min_value = 1.2, sum_value = 17.2

Warto jednak pamiętać, że funkcja sum() działa tylko dla list zawierających liczby, a próba zastosowania jej do innej listy spowoduje błąd.

14.7 Metody, które pozwalają wykonywać operacje na elementach listy

W Pythonie lista jest jednym z najważniejszych typów danych i oferuje wiele przydatnych metod do manipulowania i przetwarzania danych. Poniżej przedstawione są niektóre z najważniejszych metod list w Pythonie:

append(x) - dodaje element x na koniec listy.
list = [1, 2, 3]
list.append(4)
print(list) # [1, 2, 3, 4]

extend(iterable) - rozszerza listę o elementy z iterowalnego obiektu iterable.
list = [1, 2, 3]
list.extend([4, 5])
print(list) # [1, 2, 3, 4, 5]

insert(i, x) - dodaje element x na pozycję i w liście.
list = [1, 2, 3]
list.insert(1, 4)
print(list) # [1, 4, 2, 3]

remove(x) - usuwa pierwszy element z listy, który ma wartość x.
list = [1, 2, 3, 2]

```
list.remove(2)
print(list) # [1, 3, 2]
```

pop([i]) - usuwa i zwraca element z pozycji i w liście. Jeśli i nie jest podane, usuwa i zwraca ostatni element z listy.

```
list = [1, 2, 3]
popped_element = list.pop(1)
print(popped_element) # 2
print(list) # [1, 3]
```

index(x) - zwraca pozycję pierwszego elementu w liście o wartości x.

```
list = [1, 2, 3, 2]
index = list.index(2)
print(index) # 1
```

count(x) - zwraca liczbę wystąpień elementu x w liście.

```
list = [1, 2, 3, 2]
count = list.count(2)
print(count) # 2
```

sort() - sortuje elementy listy w miejscu.

```
list = [3, 1, 2]
list.sort()
print(list) # [1, 2, 3]
```

reverse() - odwraca kolejność elementów w liście w miejscu.

```
list = [1, 2, 3]
list.reverse()
print(list) # [3, 2, 1]
```

copy() - zwraca kopię listy.

```
list = [1, 2, 3]
new_list = list.copy()
print(new_list) # [1, 2, 3]
```

14.8 Tworzenie listy z łańcucha znakowego (stringa).

Elementami listy są litery napisu:

```
wyraz="informatyka"
lista=list(wyraz)
print(lista)
#['i' ,' n', 'f' , 'o' , 'r' ,'m' ,'a' ,'t' ,'y' ,'k' ,'a']
lista[0]  --> 'i'
```

15 WYRAŻENIA LISTOWE

Wyrażenia listowe w Python to krótkie i eleganckie sposoby tworzenia nowych list, które opierają się na wartościach istniejących list lub innych sekwencji.

#Tworzenie listy na podstawie iterowalnej sekwencji:

```
numbers = [1, 2, 3, 4, 5]
squares = [x**2 for x in numbers]
print(squares)
```

```
ra=[x//2 for x in range(2,11)]
print(ra)
```

#Tworzenie listy ze stringa:

```
wyraz=”informatyka”
lista_liter=list(wyraz)
print(lista_liter)
```

 # Tworzenie listy na podstawie filtrowania elementów z innej listy:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = [x for x in numbers if x % 2 == 0]
print(even_numbers)
```

 # Tworzenie listy zawierającej kilka wartości lub wyrażeń:

```
a = [1, 2, 3,6]
b = [4, 5, 6]
c = [x + y for x in a for y in b]
print(c)
```

#Tworzenie listy zawierającej część wspólną dwóch list:

```
nowaa=[x for x in a for y in b if x==y]
print("wyspólna część", nowaa)
```

#Tworzenie listy na podstawie innej listy, ale tylko dla określonych elementów:

```
words = ["apple", "banana", "cherry", "orange", "kiwi", "babka", "budynek"]
new_list = [word for word in words if len(word) > 5]
print(new_list)
```

```
nowa=[wyraz for wyraz in words if wyraz[0]=='b' and wyraz[-1]!='a']
print("Zaczyna się na b, kończy nie a: ",nowa)
```

#Tworzenie listy na podstawie innej listy, ale tylko dla unikalnych elementów:

```
numbers1 = [1, 2, 3, 2, 1, 4, 5]
unique_numbers = list(set(numbers1))
print(unique_numbers)
```

#Tworzenie listy krotek zawierających wartości i ich kwadraty, dla liczb od 1 do 5:

```
values_and_squares = [(i, i**2) for i in range(1, 6)]
print(values_and_squares)
```

#Tworzenie listy list zawierających wartości, ich kwadraty i pierwiastki, dla liczb od 1 do 11:

```
v=[[i,i**2,i**(0.5)] for i in range(1,11)]
print(v)
```

16 Krotka (tuple)

Krotka (tuple) w Python to struktura danych, która służy do przechowywania kolekcji elementów. Krótka jest podobna do listy, ale jest niezmienna (niemutowalna), co oznacza, że po utworzeniu nie można zmienić jej zawartości.

Krótka jest tworzona przez umieszczenie elementów w nawiasach okrągłych i oddzielenie ich przecinkami. Na przykład:

moja_krotka = (1, 2, 3, "cztery", "pięć")

Można uzyskać dostęp do elementów krotki za pomocą ich indeksów, tak jak w przypadku listy:

pierwszy_element = moja_krotka[0]

Krotki są przydatne, gdy chcesz przechowywać niezmienne wartości, takie jak stałe lub ciągi znaków, lub gdy chcesz chronić dane przed przypadkowymi modyfikacjami.

Krotka (ang. *tuple*) jest niezmienną listą. Zawartość krotki określamy tylko podczas jej tworzenia. Potem nie możemy już jej zmienić.

Przykład. ***Definiowanie krotki***

```
>>> t = ("a", "b", "mpilgrim", "z", "element")     #(1)
>>> t
('a', 'b', 'mpilgrim', 'z', 'element')
>>> t[0]                                           #(2)
'a'
>>> t[-1]                                          #(3)
'element'
>>> t[1:3]                                         #(4)
('b', 'mpilgrim')
```

1. Krotki definiujemy w identyczny sposób jak listę, lecz z jednym wyjątkiem -- zbiór elementów jest ograniczony w nawiasach okrągłych, zamiast w kwadratowych.
2. Podobnie jak w listach, elementy w krotce mają określony porządek. Są one indeksowane od `0`, więc pierwszym elementem w niepustej krotce jest zawsze `t[0]`.
3. Ujemne indeksy idą od końca krotki, tak samo jak w listach.
4. Krotki także można wycinać. Kiedy wycinamy listę, dostajemy nową listę. Podobnie, gdy wycinamy krotkę dostajemy nową krotkę.

Przykład. ***Krotki posiadają mniej metod niż listy***

```
>>> t
('a', 'b', 'mpilgrim', 'z', 'element')
>>> t.append("nowy")                                #(1)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> t.remove("z")                                          #(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> t.index("z")                                           #(3)
3
>>> "z" in t                                               #(4)
True
```

1. Nie można dodawać elementów do krotki. Krotki nie posiadają metod typu `append`, czy też `extend`.
2. Nie można usuwać elementów z krotki. Nie posiadają one ani metody `remove` ani metody `pop`.
3. Można wyszukiwać elementy w krotce wykorzystując metodę `index`.
4. Można wykorzystać operator `in`, aby sprawdzić, czy krotka zawiera dany element.

To w końcu do czego są te krotki przydatne?

- Krotki działają szybciej niż listy. Jeśli definiujemy stały zbiór wartości, który będzie używany tylko do iteracji, skorzystajmy z krotek zamiast z listy.
- Jeśli chcielibyśmy używać danych "zabezpieczonych przed zapisem" (np. po to, żeby program był bezpieczniejszy), wykorzystajmy do tego krotki. Korzystając z krotek, zamiast z list, mamy pewność, że dane w nich zawarte nie zostaną nigdzie zmienione. To trochę tak, jakbyśmy mieli gdzieś ukrytą instrukcję `assert` sprawdzającą, czy dane są stałe. W przypadku, gdy nastąpi próba nadpisania pewnych wartości w krotce, program zwróci wyjątek.
- Pamiętasz, jak powiedzieliśmy, że klucze w słowniku mogą być łańcuchami znaków, liczbami całkowitymi i "kilkoma innymi typami"? Krotki są jednym z tych "innych typów". W przeciwieństwie do list, mogą one zostać użyte jako klucz w słowniku. Dlaczego? Jest to dosyć skomplikowane. Klucze w słowniku muszą być niezmienne. Krotki same w sobie są niezmienne, jednak jeśli wewnątrz krotki znajdzie się lista, to krotka ta nie będzie mogła zostać użyta jako klucz, ponieważ lista jest zmienna. W takim przypadku krotka nie byłaby słownikowo-bezpieczna. Aby krotka mogła zostać wykorzystana jako klucz, musi ona zawierać wyłącznie łańcuchy znaków, liczby i inne słownikowo-bezpieczne krotki.
- Krotki używane są do formatowania tekstu, co zobaczymy wkrótce.

Krotki mogą zostać w łatwy sposób przekonwertowane na listy. Wbudowana funkcja `tuple`, której argumentem jest lista, zwraca krotkę z takimi samymi elementami, natomiast funkcja `list`, której argumentem jest krotka, zwraca listę. W rezultacie `tuple` zamraża listę, a `list` odmraża krotkę.

17 Słowniki

Jednym z wbudowanych typów są słowniki (ang. *dictionary*). Określają one wzajemną relację między kluczem a wartością.

Słowniki w Pythonie są podobne do haszy w Perlu. W Perlu zmienne przechowujące hasz są reprezentowane zawsze przez początkowy znak %. W Pythonie typ danych jest automatycznie rozpoznawany. Pythonowy słownik przypomina ponadto instancję klasy Hashtable w Javie, a także instancję obiektu Scripting.Dictionary w Visual Basicu.

18 Definiowanie słowników

Przykład. *Definiowanie słowników*

```
>>> d = {"server":"mpilgrim", "database":"master"}  #(1)
>>> d
{'database': 'master', 'server': 'mpilgrim'}
>>> d["server"]                              #(2)
'mpilgrim'
>>> d["database"]                            #(3)
'master'
>>> d["mpilgrim"]                            #(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 'mpilgrim'
```

1. Najpierw utworzyliśmy nowy słownik z dwoma elementami i przypisaliśmy go do zmiennej d. Każdy element w słowniku jest parą *klucz-wartość*, a zbiór elementów jest ograniczony nawiasem klamrowym.
2. 'server' jest kluczem, a skojarzoną z nim wartością jest 'mpilgrim', do której odwołujemy się poprzez d["server"].
3. 'database' jest kluczem, a skojarzoną z nim wartością jest 'master', do której odwołujemy się poprzez d["database"].
4. Możesz dostać się do wartości za pomocą klucza, ale nie możesz dostać się do klucza za pomocą wartości. Tak więc d["server"] zwraca 'mpilgrim', ale wywołanie d["mpilgrim"] sprawi, że zostanie rzucony wyjątek, ponieważ 'mpilgrim' nie jest kluczem słownika d.

19 Modyfikowanie słownika

Przykład. *Modyfikowanie słownika*

```
>>> d
{'database': 'master', 'server': 'mpilgrim'}
>>> d["database"] = "pubs"                   #(1)
>>> d
{'database': 'pubs', 'server': 'mpilgrim'}
>>> d["uid"] = "sa"                          #(2)
>>> d
{'database': 'pubs', 'uid': 'sa', 'server': 'mpilgrim'}
```

1. Klucze w słowniku nie mogą się powtarzać. Przypisując wartość do istniejącego klucza, będziemy nadpisywać starszą wartość.
2. W każdej chwili możesz dodać nową parę klucz-wartość. Składnia jest identyczna do tej, która modyfikuje istniejącą wartość. Czasami może to spowodować pewien problem. Możemy myśleć, że dodaliśmy nową wartość do słownika, ale w rzeczywistości nadpisaliśmy już istniejącą.

Zauważmy, że słownik nie przechowuje kluczy w sposób posortowany. Wydawałoby się, że klucz 'uid' powinien się znaleźć za kluczem 'server', ponieważ litera *s* jest wcześniej w alfabecie niż *u*. Jednak tak nie jest. Elementy słownika znajdują się w losowej kolejności.

W słownikach nie istnieje określona kolejność układania elementów. Nie możemy dostać się do elementów w słowniku w określonej, uporządkowanej kolejności (np. w porządku alfabetycznym). Oczywiście są sposoby, aby to osiągnąć, ale te "sposoby" nie są wbudowane w sam słownik.

Pracując ze słownikami pamiętajmy, że wielkość liter kluczy ma znaczenie.

Przykład. ***Nazwy kluczy są wrażliwe na wielkość liter***

```
>>> d = {}
>>> d["klucz"] = "wartość"
>>> d["klucz"] = "inna wartość"                    #(1)
>>> d
{'klucz': 'inna wartość'}
>>> d["Klucz"] = "jeszcze inna wartość"            #(2)
>>> d
{'klucz': 'inna wartość', 'Klucz': 'jeszcze inna wartość'}
```

1. Przypisując wartość do istniejącego klucza zamieniamy starą wartość na nową.
2. Nie jest to przypisanie do istniejącego klucza, a ponieważ łańcuchy znaków w Pythonie są wrażliwe na wielkość liter, dlatego też 'klucz' nie jest tym samym co 'Klucz'. Utworzyliśmy nową parę klucz-wartość w słowniku. Obydwa klucze mogą się wydawać podobne, ale dla Pythona są zupełnie inne.

Przykład. ***Mieszanie typów danych w słowniku***

```
>>> d
{'bazadanych': 'pubs', 'serwer': 'mpilgrim', 'uid': 'sa'}
>>> d["licznik"] = 3          #(1)
>>> d
{'bazadanych': 'pubs', 'serwer': 'mpilgrim', 'licznik': 3, 'uid': 'sa'}
>>> d[42] = "douglas"         #(2)
>>> d
{42: 'douglas', 'bazadanych': 'pubs', 'serwer': 'mpilgrim', 'licznik': 3, 'uid': 'sa'}
```

1. Słowniki nie są przeznaczone tylko dla łańcuchów znaków. Wartość w słowniku może być dowolnym typem danych: łańcuchem znaków, liczbą całkowitą, obiektem, a nawet innym słownikiem. W pojedynczym słowniku wszystkie wartości nie muszą być tego samego typu; możemy wstawić do niego wszystko, co chcemy.
2. Klucze w słowniku są bardziej restrykcyjne, ale mogą być łańcuchami znaków, liczbami całkowitymi i kilkoma innymi typami. Klucze wewnątrz jednego słownika nie muszą posiadać tego samego typu.

20 Usuwanie pozycji ze słownika

Przykład. ***Usuwanie pozycji ze słownika***

```
>>> d
{'licznik': 3, 'bazadanych': 'master', 'serwer': 'mpilgrim', 42: 'douglas', 'uid': 'sa'}
>>> del d['42']                          #(1)
>>> d
{'licznik': 3, 'bazadanych': 'master', 'serwer': 'mpilgrim', 'uid': 'sa'}
>>> d.clear()                            #(2)
>>> d
{}
```

1. Instrukcja del każe usunąć określoną pozycję ze słownika, która jest wskazywana przez podany klucz.
2. Instrukcja clear usuwa wszystkie pozycje ze słownika. Zbiór pusty ograniczony przez nawiasy klamrowe oznacza, że słownik nie ma żadnego elementu.

21 Funkcje

Funkcje pozwalają podzielić większy problem na mniejsze części, a także przyspieszają i ułatwiają pisanie programu.

Funkcje mogą zwracać wartość do programu głównego 2) lub nic nie zwracać 1)

1)

```
def nazwa_funkcji (prametr1, parametr2, parametr3):
    lista instrukcji
```

2)

```
def nazwa_funkcji (prametr1, parametr2, parametr3):
    lista instrukcji
    return wartość
```

Jako parametr możemy podawać, dowolną 'rzecz' - liczbę, łańcuch, listę, znak, itd

Funkcji niezwracającej wartości możemy używać wtedy, gdy mamy wyświetlić dane na ekranie.
Aby wywołać funkcję niezwracającą wartości, należy umieścić nazwę funkcji (z parametrami formalnymi) w odpowiednim miejscu programu głównego. W przypadku braku parametrów nawiasy po nazwie funkcji pozostawiamy puste.

Przykład 1:

```
def linia_z_gwiazdek():
        print("8"*25)
```

Wywołanie funkcji w programie głównym:

```
linia_z_gwiazdek()
```

Przykład 2:

Jeżeli **funkcja zwraca wartość** to musi zawierać instrukcję **return** z wartością zwracaną do programu głównego.

Wartość zwróconą przez funkcję możemy przypisać zmiennej w programie głównym

Przykład:
def pole(a):
  return a*a

Wywołanie funkcji w programie głównym:
pole_kwadratu=pole(15);

li[1],li[2]=li[2],li[1]

22 Zmienne plikowe

Do obsługi plików potrzebujesz specjalnej zmiennej, która będzie pełniła rolę uchwytu do pliku.

Wszelkie operacje związane z plikiem wykonuje się właśnie poprzez wykonywanie metod oferowanych przez zmienną plikową. Aby stworzyć zmienną plikową należy otworzyć plik.

Do otwierania pliku służy metoda open. Zwraca ona uchwyt do pliku. Metoda za argument przyjmuje ścieżkę do pliku.

f = open("nazwa_pliku.txt", "r") # otwarcie pliku

Jeśli podamy nazwę nieistniejącego pliku to zostanie on utworzony.

Tryby otwarcia pliku
Drugim często stosowanym argumentem metody jest tryb otwarcia pliku.

Najczęściej stosowane tryby to:

“r” – plik otwarty do odczytu (read)
“w” – plik otwarty do zapisu (write), przed zapisem zawartość pliku jest usuwana
“a” – plik otwarty do zapisu, dodaje nową treść na końcu pliku, nie usuwa starej (append)
Domyślnie wszystkie pliki otwierane są w trybie tekstowym. Możemy otwierać je również w trybie binarnym dodając flagę ‘b‘. Na przykład “rb” otworzy nam plik do oczytu binarnie.

Użycie operatora ‘+’ pozwoli na otwarcie pliku zarówno do zapisu jak i odczytu (np “r+”).

22.1 Odczyt z pliku

Zawartość otwartego pliku możemy wczytać od razu w całości, bądź czytać go linia po linii.

Odczyt całego pliku
Aby odczytać całą treść pliku skorzystaj z metody read.

```
>>> f.read()
'Pierwsza linianDruga linianTrzecia linian123n3.14'
```

Za pomocą tej samej metody możemy przeczytać konkretną liczbę znaków podaną przez nas jako parametr metody:

```
>>> f = open(filepath, "r")
>>> f.read(8)
'Pierwsza'
```

UWAGA! Po odczytaniu zawartości pliku zostaje ona “wykasowana” ze zmiennej plikowej. W praktyce wygląda to tak, że “wskaźnik” odpowiadający za czytanie pliku przesuwa się. Można to porównać do przesuwania palcem po książce podczas czytania.

```
>>> f = open(filepath, "r")
>>> f.read(8)
'Pierwsza'
>>> f.read()
' linianDruga linianTrzecia linian123n3.14'
```

Nie zdziw się więc, że po odczytaniu całego pliku następne wywołanie metody read zwróci pusty napis.

Czytanie pliku liniami
Aby odczytać pojedyczną linię z pliku użyj polecenia readline.

```
>>> print(f.readline())
>>> f.readline()
'Pierwsza linian'
```

Jak widać zczytana została cała pierwsza linia pliku (wraz ze znakiem końca linii).

Ponowne użycie metody pozwoli na wczytanie następnej linii:

```
>>> f.readline()
'Druga linian'
```

Plik możemy czytać linia po linii za pomocą pętli:

```
>>> f = open(filepath, "r")
>>> for line in f:
    print(line)

Pierwsza linia
Druga linia
Trzecia linia
```

Inną opcją jest wczytanie wszystkich linii pliku do listy. Służy do tego metoda readlines.

```
>>> f = open(filepath, "r")
>>> lines = f.readlines()
>>> lines
['Pierwsza linia\n', 'Druga linia\n', 'Trzecia linia\n', '123\n', '3.14']
```

23 System pozycyjny

Codziennie posługujemy się systemem dziesiętnym zwanym inaczej decymalnym. Jest to system pozycyjny co oznacza, że liczba składa się z cyfr, których wartość zależy od pozycji oraz podstawy systemu. Rozpatrzmy przykładowo liczbę 123. System dziesiętny ma podstawę 10 czyli możemy liczbę zapisać jako:

$123 = 100 + 20 + 3 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$

Łatwo zauważyć, że cyfra na i-tej pozycji od prawej strony ma wartość i-ta cyfra pomnożona przez 10 do i-1.

Inne systemy pozycyjne

System pozycyjny może mieć dowolną podstawę n. Oznacza to, że na dowolnej pozycji mamy n różnych sposobów zapisania cyfry. Rezultatem tego jest, że liczba na i-tej pozycji od prawej strony ma wartość i-ta cyfra pomnożona przez n do i-1. W celu odróżnienia liczb w różnych system zapisach przyjmuje się, że liczba a o bazie n zapisuje się jako a_n.

Przykłady systemów

System	Inaczej	Podstawa	Cyfry	Przykład użycia
dwójkowy	binarny	2	0, 1	system binarny jest podstawą wszystkiego związanego z informatyką
ósemkowy	oktalny	8	0 - 7	w system UNIX do reprezentowania poziomu dostępu do pliku
szesnastkowy	heksadecymalny	16	0 - 9, A - F	określanie koloru podczas stylizowania strony

23.1 Przeliczanie systemu dziesiętnego na inny

Weźmy liczbę a z systemu dziesiętnego. Chcąc przeliczyć ją do innego systemu o podstawie n należy:

- Podziel liczbę przez n i pobierz resztę z dzielenia.

- W i-tym kroku zapisz cyfrę na i-tej pozycji od prawej strony
- Jeśli a jest różne od 0 to wróć do (1.)

Przykład

Weźmy przykładowo liczbę 21, które chcemy przeliczyć na system binarny:

Iteracja	Operacja	Wynik	Aktualna liczba
#1	21 / 2	10 r. 1	1
#2	10 / 2	5 r. 0	01
#3	5 / 2	2 r. 1	101
#4	2 / 2	1 r. 0	0101
#5	1 / 2	0 r. 1	10101

Liczba a wynosi już 0, więc nie wykonujemy iteracji #6. W ten sposób odczytujemy, że $21_{10} = 10101_2$.

23.2 Inny system na dziesiętny

Weźmy liczbę a o bazie n. Chcąc przeliczyć ją do systemu dziesiętnego należy zsumować wartości wszystkich cyfr liczby pamiętając, że i-ta cyfra ma wartość i-tej cyfry pomnożona przez ni-1.

Weźmy przykładowo 10121_3. Chcąc przeliczyć wyliczamy kolejne pozycje:$1 \cdot 3^4 + 0 \cdot 3^3 + 1 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0 = 81 + 0 + 9 + 6 + 1 = 97$. Z tego możemy odczytać, że $10121_3 = 97_{10}$.

23.3 Przeliczanie między dowolnymi systemami

Załóżmy, że mamy liczbę a z systemu m i chcemy przeliczyć ją na system n. Dokonujemy tego:

- Zamieniając liczbę a z systemu m na dziesiętny
- Powstałą liczbę przeliczamy na system n

Przykładowo chcemy przeliczyć 18_3 na system ósemkowy:

$18_3 = 1 \cdot 3^1 + 8 \cdot 3^1 = 11_{10}$

Iteracja	Krok 1	Krok 2	Aktualna liczba
#1	11 / 8	1 r. 3	3
#2	1 / 8	0 r. 1	13

Z tego odczytujemy, że $18_3 = 13_8$.

23.4 Szybka konwersja - binarny na ósemkowy

Algorytm wygląda następująco:

- Począwszy od prawej grupujemy cyfry liczby binarnej po 3
- Każdą grupkę zamieniamy na system ósemkowy

Przykładowo zamienimy liczbę 10101111_2:

Krok	Liczba		
#0	10101111_2		
#1	010_2	101_2	111_2
#2	2_{10}	5_{10}	7_{10}
#3	257_8		

Analogicznie wykonujemy zamianę systemu binarnego na szesnastkowy.

Szybka konwersja - szesnastkowy na binarny

Algorytm wygląda następująco:

- Każdą cyfrę przeliczamy na system binarny

Przykładowo zamienimy liczbę 21_{16}:

Krok	Liczba	
#0	21_{16}	
#1	2_{10}	1_{10}
#2	0010_2	0101_2
#3	100101_2	

W ten sposób wyliczamy, że $21_{16} = 100101_2$.

24 Funkcje konwertujące

Funkcje, nie metody. Przypominam o tej różnicy. W Pythonie występuje wsparcie dla systemu dziesiętnego, ale także dla binarnego, ósemkowego zwanego "oktalnym" oraz heksadecymalnego (szesnastkowego).
Oto funkcje zwracające łańcuch znaków poprzez podaną w parametrze liczbę w systemie dziesiętnym na jej odpowiednik w innym systemie liczbowym:

24.1 Funkcja "bin" w Pythonie (systemy liczbowe)

"bin" to skrót od "binary", czyli konwersja na system binarny. Zgodnie z naturą systemu dwójkowego, zwraca ciąg bitów zerojedynkowych. Oprócz wyniku, dodaje przedrostek "0b" celem zasygnalizowania, że to jest liczba binarna. Wszystkie funkcje konwertujące na swoje systemy liczbowe dodają taki przedrostek.

```
 bin(12)
'0b1100'
```

24.2 Funkcja "oct" w Pythonie (systemy liczbowe)

Gdy nie interesują Was inne systemy liczbowe niż ósemkowy, to korzystacie z "oct". Po przyjęciu podanej liczby, zwraca ciąg cyfr przyjmujących wartości od 0 do 7 według reguły "trzymania się" podstawy równej osiem. W tym wypadku też dodaje przedrostek i to jest "0o" (zero i mała litera 'o').

```
oct(12)
'0o14'
```

24.3 Funkcja "hex" w Pythonie (systemy liczbowe)

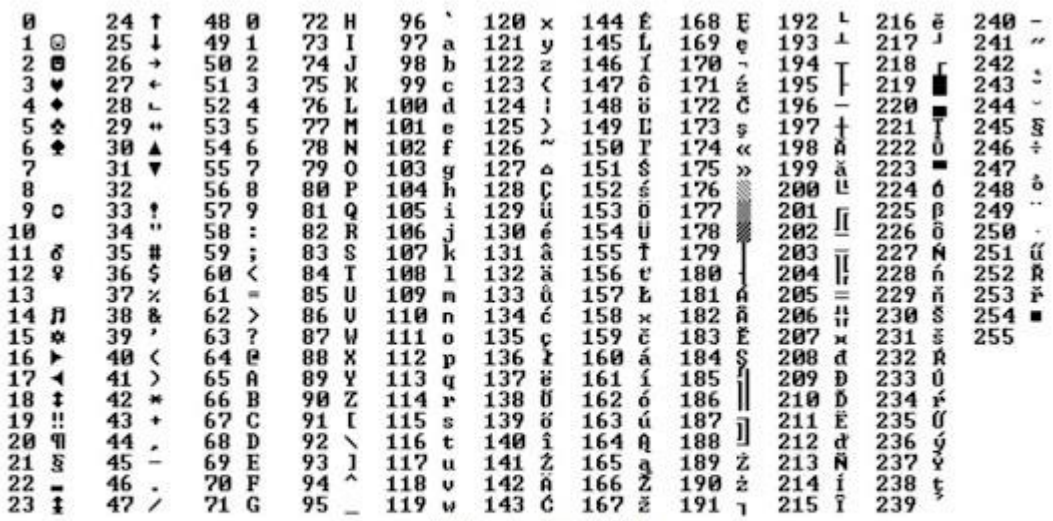
Chcecie zamienić na liczbę w systemie heksadecymalnym? To wywołujecie funkcję "hex" (ang. "hexadecimal"). Podana liczba zamieni się wówczas w ciąg cyfr i potencjalnych liter, gdyż liczby 10-15 przyjmują wówczas formę pierwszych liter alfabetu od A do F. "0x", to jest przedrostek symbolizujący system szesnastkowy. Wszystkie systemy liczbowe mają swój przedrostek.

```
hex(12)
'0xc'
```

int(x[, system]) Przekształca napis na liczbę całkowitą. Jeśli argument jest napisem, musi reprezentować liczbę całkowitą Pythona ewentualnie otoczoną białymi znakami. Parametr system definiuje system zapisu liczb i może być liczbą całkowitą w przedziale [2, 36] lub zerem. Jeśli system jest zerem prawidłowy system jest określany na podstawie zawartości napisu. Jeśli system jest zdefiniowany a x nie przedstawia liczby wywoływany jest TypeError.

25 Tablica kodów ASCII oraz funkcje konwertujące

ord('a') -> daje w wyniku LICZBĘ będącą nr znaku 'a' w tablicy kodów ASCII czyli **97**. Argumentem funkcji jest znak
chr(100) -> daje w wyniku ZNAK o kodzie 100 czyli znak **'d'**. Argumentem funkcji jest liczba naturalna z zakresu 1-255.



Tablica kodów ASCII

Zamiana kodu znaku na znak: chr(65) -> "A"
Zamiana znaku na jego kod: ord("a") -> 97

26 Zbiory (set)

https://www.obliczeniowo.com.pl/414

27 Python lambda – wszystko co trzeba wiedzieć

https://analityk.edu.pl/python-lambda-wszystko-co-trzeba-wiedziec/

28 Szyfrowanie informacji

https://szkolyete.pl/index.php/gimnazjumgilo/przedmioty/informatyka/2008-szyfrowanie-informacji

29 Moduł random

Moduł "random" zawiera w sobie metody przeznaczone między innymi do generowania liczb pseudolosowych:

"randint" pozwala otrzymać liczbę pseudolosową z podanego w dwóch parametrach przedziału. W pierwszym parametrze podajemy minimum, a w drugim, maksimum. Na wyjściu otrzymujemy zwrot liczby całkowitej.

29.1 ZASADY IMPORTOWANIA MODUŁÓW:

W Pythonie jest wskazane importować wszystko co będzie potrzebne na samej górze kodu źródłowego przed jakimikolwiek innymi operacjami. Powodem korzystania z importu jest możliwość skorzystania z już wcześniej napisanych i sprawdzonych algorytmów oraz obiektów, żeby nie musieć pisać niezbędnych funkcjonalności od nowa (określa się to mianem "wynajdywania koła na nowo").

Podstawowym słowem kluczowym importującym moduły jest "import". Zaraz po nim dopisujecie nazwę modułu posiadającego funkcje i zmienne z których będziecie korzystać np.

import random

Gdy zechcemy odwołać się do zaimportowanych danych, musimy za każdym razem dopisać przedrostek w postaci tej samej nazwy modułu. Dopiero po kropce możemy dostać się do atrybutu lub metody.

x = random.randint(5, 9)

"FROM" DRUGĄ METODĄ IMPORTOWANIA

Gdy staje się dla nas uciążliwe odwoływanie się za każdym razem do danych po kropce, możemy zastosować drugi wariant importowania polegający na połączeniu słowa "import" ze słowem "from":

from random import *

wtedy możemy napisać:

x = randint(5, 9)

```
import random
random.randint(1,49)

lub
from random import *
randint(1,49)
```

30 Biblioteka Trtle – grafika żółwia

31 Podstawowe ruchy turtle

Import biblioteki turtle:
from turtle import*

```
window=Screen()

window.onclick(click)
def click(x,y):
    exit()
```

Podstawowe operacje w turtle, to przemieszczanie się naszego żółwia. Mamy do dyspozycji dość sporą liczbę funkcji. Najważniejsze to:

- forward(x) lub fd(x) - do przodu, o podany dystans
- backward(x) lub bk(x) - do tyłu, o podany dystans
- right(x) lub rt(x), left(x) lub lt(x) - obróć się w prawo / lewo, o podany kąt
- goto(x,y) - pójdź do konkretnego punktu
- home() - wróć na środek ekranu
- circle(x) - narysuj koło o podanym promieniu
- bgcolor("kolor") - zmiana koloru tła
- pencolor("kolor") - zmiana koloru pisaka
  - colormode(255)
  - pencolor(r,g,b) np. pencolor(123,87,244)
- pensize(x) zmiana grubości pisaka
- speed(x) - prędkość poruszania się pisaka
  - 0 - najszybciej
  - 1 - najwolniej
  - 3- wolno
  - 6 - normalnie
  - 10 - szybko
- shape("turtle") - zmiana kształtu pisaka
- write("KONIEC", font=("Arial", 12))

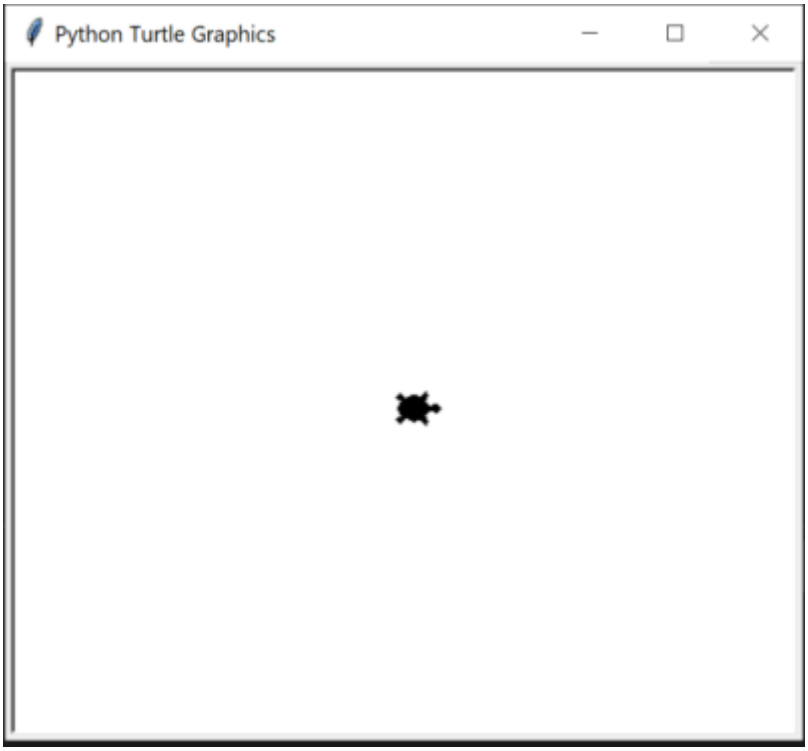
Losowe kolory:
from random import*
bgcolor(randrange(1,256),randrange(1,256),randrange(1,256))

fillcolor("kolor") - ustawia kolor wypełnienia
begin_fill() - początek wypełnienia figury
end_fill() - koniec wypełnienia figury

Stworzenie pierwszego programu, jest bardzo proste. Wystarczy utworzyć obiekt żółwia, a następnie uruchomi się okienko, natomiast na jego środku, będzie bohater biblioteki, czyli żółw. Domyślenie ma on kształt strzałki, jednak możemy w łatwy sposób nadać mu faktyczny kształt.

```
import turtle
żółw = turtle.Turtle()
żółw.shape('turtle')
turtle.exitonclick()
```

a następnie, po uruchomieniu programu, zobaczymy następujące okienko



funkcja exitonclick(), umieszczona na końcu, powoduje że okienko się nie zamknie, dopóki nie klikniemy.

32 Podstawowe ruchy turtle

Podstawowe operacje w turtle, to przemieszczanie się naszego żółwia. Mamy do dyspozycji dość sporą liczbę funkcji. Najważniejsze to:

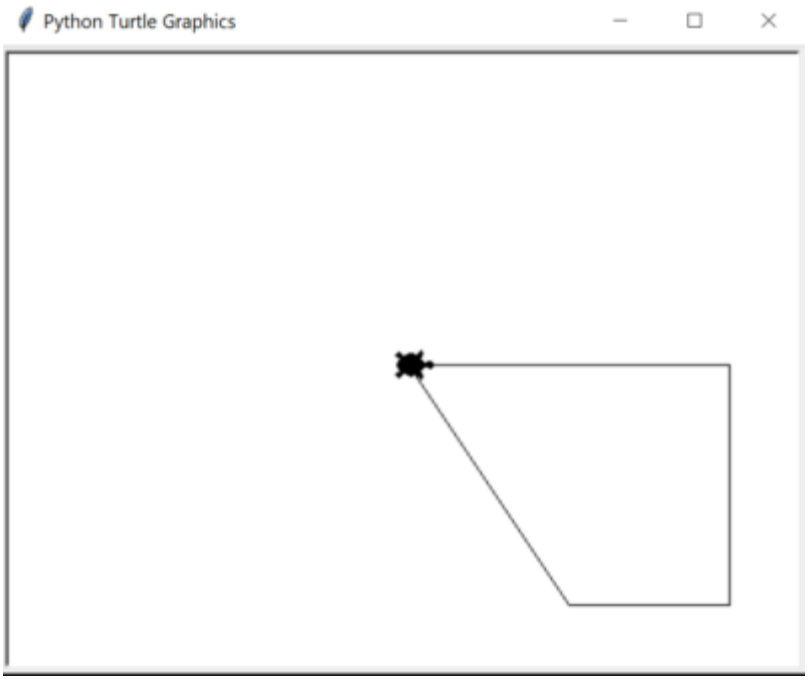
forward(x) lub fr(x)	do przodu, o podany dystans
backward(x) lub bc(x)	do tyłu, o podany dystans
right(x) lub rt(x), left(x) lub lf(x)	obróć się w prawo / lewo, o podany kąt
goto(x,y)	pójdź do konkretnego punktu
home()	wróć na środek ekranu
circle(x)	narysuj koło o podanym promieniu

W rezultacie, po dodaniu następujących linijek do naszego programu:

```
import turtle
t = turtle.Turtle()
t.shape('turtle')
t.forward(200)
t.right(90)
t.forward(150)
```

```
t.right(90)
t.forward(100)
t.home()
turtle.exitonclick()
```

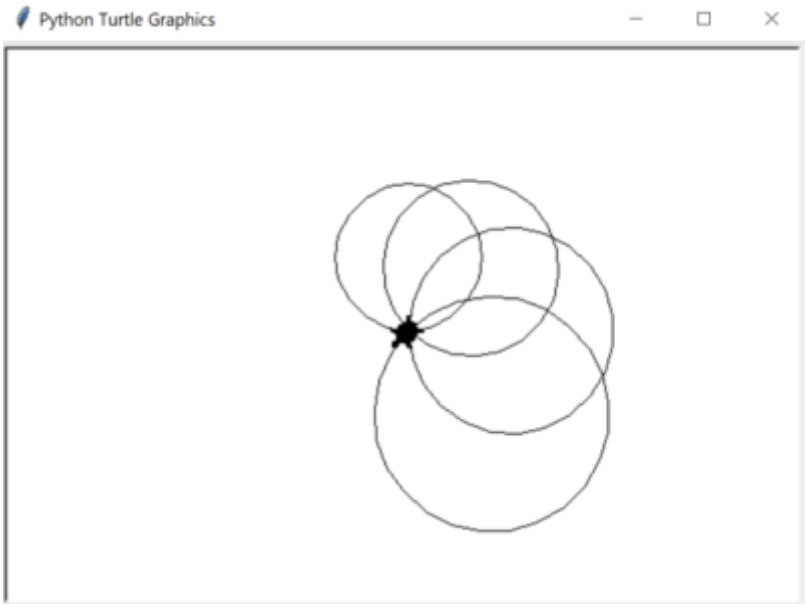
Otrzymamy następującą figurę:



W analogiczny sposób, możemy rysować dowolnie złożone figury. Dodatkowo, warto zauważyć, że funkcje mają również swoje nazwy skrócone.

I jeszcze kilka kółek, aby pokazać co nasz żółw potrafi:

```
import turtle
t = turtle.Turtle()
t.shape('turtle')
t.circle(50)
t.right(45)
t.circle(60)
t.right(45)
t.circle(70)
t.right(45)
t.circle(80)
turtle.exitonclick()
```



33 Kolory, rozmiar oraz prędkość

Jeżeli umiemy już poruszać się naszym żółwiem, możemy uatrakcyjnić jego wygląd, za pomocą kilku prostych funkcji:

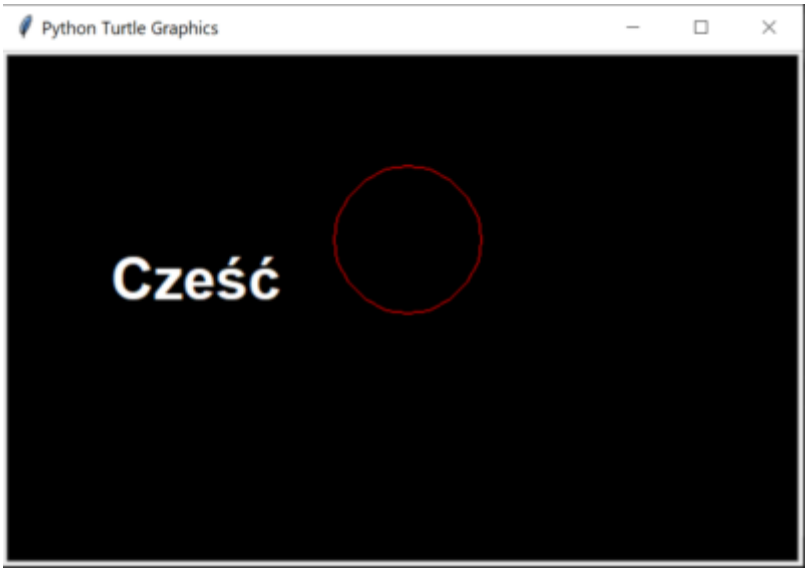
turtle.bgcolor(x)	zmienia kolor tła. Jak x możemy podać takie kolory jak 'red',’blue’,’green’,’yellow’,’black’, 'white’ i wiele innych. Możemy również podać kolor w RGB (x,y,z)
color(x)	zmienia kolor naszego żółwia i linii które rysuje
pensize(x)	ustawia grubość linii
penup()	powoduje że żółw przemieszcza się bez rysowania
pendown()	żółw znowu rysuje
hideturtle() oraz showturtle()	żółw znika / pojawia się
speed(x)	ustawia prędkość żółwia
write(tekst)	żółw pisze

I jeżeli połączymy kilka tych funkcji w jeden program:

```
import turtle
turtle.bgcolor('black')
t = turtle.Turtle()
t.shape('turtle')
t.color('red')
t.circle(50)
t.penup()
t.goto(-200,0)
t.pendown()
t.color('white')
t.write('Cześć', font=("Arial",30,"bold"))
t.hideturtle()
```

turtle.exitonclick()

Otrzymamy następująca grafikę:

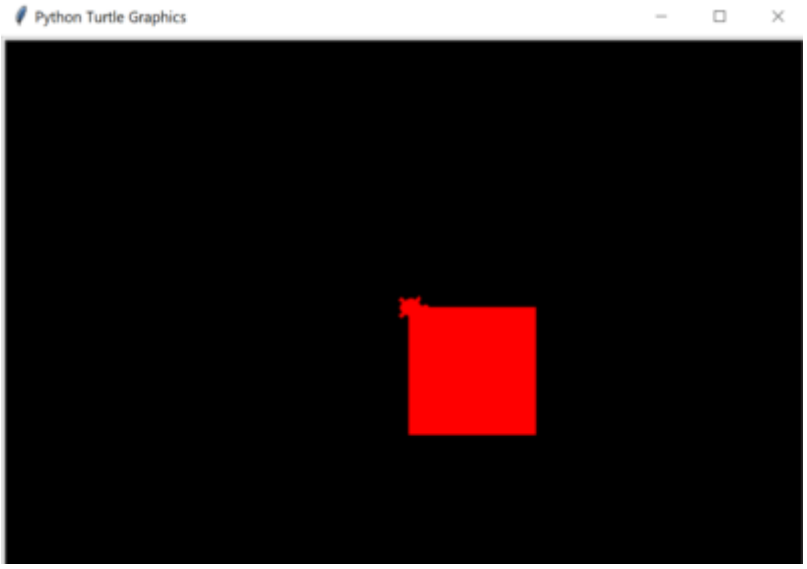


34 Wypełnianie kolorem w turtle

Biblioteke turtle, oferuje również prosty sposób wypełniania kształtów kolorem. Wystarczy, że przed rozpoczęciem rysowania figury, wywołamy funkcję begin_fill(), a po zakończeniu end_fill(), i narysowana przez nas figura się wypełni.

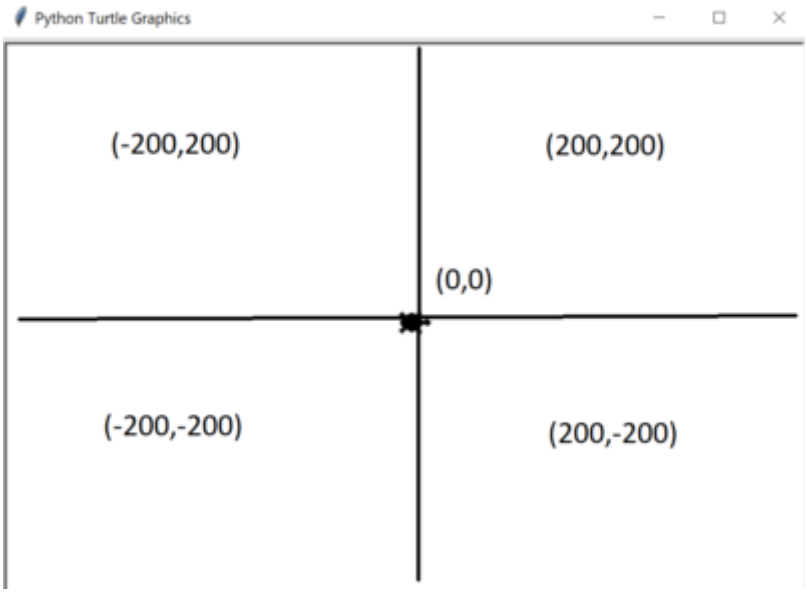
```
import turtle
turtle.bgcolor('black')
t = turtle.Turtle()
t.shape('turtle')
t.color('red')
t.begin_fill()
t.forward(100)
t.right(90)
t.forward(100)
t.right(90)
t.forward(100)
t.right(90)
t.forward(100)
t.right(90)
t.end_fill()
turtle.exitonclick()
```

A w rezultacie otrzymamy:



35 Lokalizacja żółwia

Jak już mogliśmy zauważyć, nasz żółw startuje z punktu (0,0). Jest to środek planszy. Jeżeli chcemy go przemieszczać w różnych kierunkach, przy użyciu współrzędnych, to wykonujemy to w analogiczny sposób jak znany z geometrii:



Dodatkowo, w każdym momencie, możemy uzyskać położenie naszego żółwia, za pomocą funkcji *position()*.

36 Wiele żółwi

Naszego żółwia, możemy skopiować, za pomocą funkcji clone(), jak i również możemy utworzyć kolejne żółwie za pomocą turtle.Turtle().

W ten sposób możemy tworzyć grafikę, w kilku miejscach planszy, niezależnie.

```
import turtle
t = turtle.Turtle()
t.shape('turtle')
t.color('red')
t.pensize(7)
t.penup()
t.goto(-200,100)
```

```
t.pendown()
t.forward(300)
t2 = turtle.Turtle()
t2.shape('turtle')
t2.color('blue')
t2.pensize(7)
t2.penup()
t2.goto(-200,-100)
t2.pendown()
t2.forward(150)
turtle.exitonclick()
```

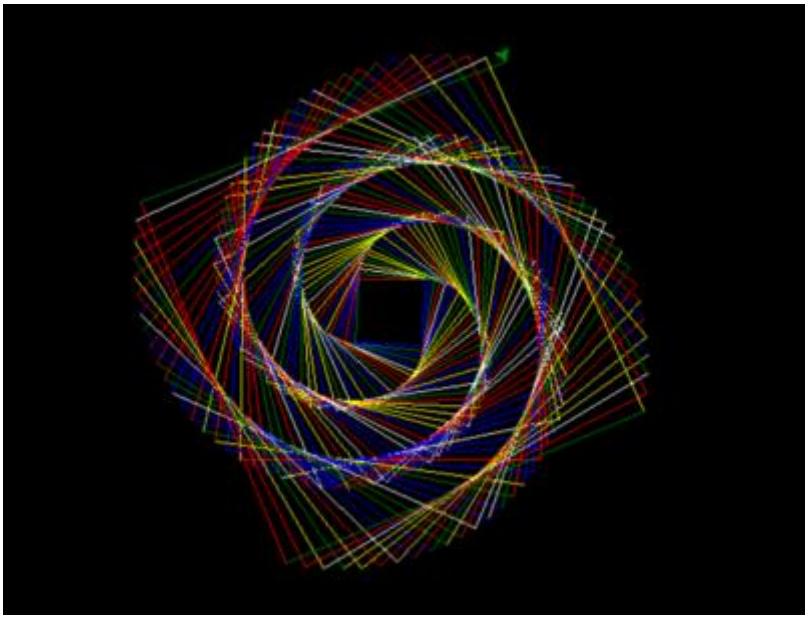
37 Pierwsza, bardziej złożona figura w Python turtle

Używając powyższej wiedzy, oraz odrobinę pętli, możemy rysować bardzo abstrakcyjne kształty. W poniższym kodzie, malujemy tło na czarno, następnie, ustawiamy prędkość żółwia na maksymalną i w pętli, rysujemy:

- losujemy kolor linii
- przemieszczamy żółwia o x
- obracamy żółwia o 91 stopni
- powtarzamy pętlę

```
import turtle
import random
turtle.bgcolor('black')
t = turtle.Turtle()
t.speed(0)
for x in range(50, 300):
t.color(random.choice(['white','red','blue','green','yellow']))
t.forward(x)
t.right(91)
turtle.exitonclick()
```

W rezultacie otrzymujemy:



38 Obsługa klawiszy w turtle

W turtle, mamy możliwość nasłuchiwania na zdarzenia, takie jak naciśnięcie klawisza, i jego obsłużenie. Czyli wykonanie odpowiedniej akcji. Pozwala to na pisanie interaktywnych programów.

Podstawową funkcją, którą trzeba użyć jest turtle.onkey(funkcja, zdarzenie). Pierwszy parametr, to funkcja która ma zostać wykonana, w momencie, kiedy nastąpi zdarzenie opisane przez drugi parametr.

Następnie na końcu naszego programu, należy poinformować Python turtle, że ma nasłuchiwać na zdarzenia, oraz całość ma być powtarzana.

turtle.listen()

turtle.mainloop()

program zakończy się, w momencie kiedy zostanie uruchomiona funkcja turtle.bye()

Zdecydowanie prościej jest to pokazać, niż wytłumaczyć, tak więc przyjrzyjmy się prostemu programowi.

Jego zadaniem, jest nasłuchiwanie na 4 klawisze – strzałka do góry, w lewo, w prawo oraz klawisz 'q':

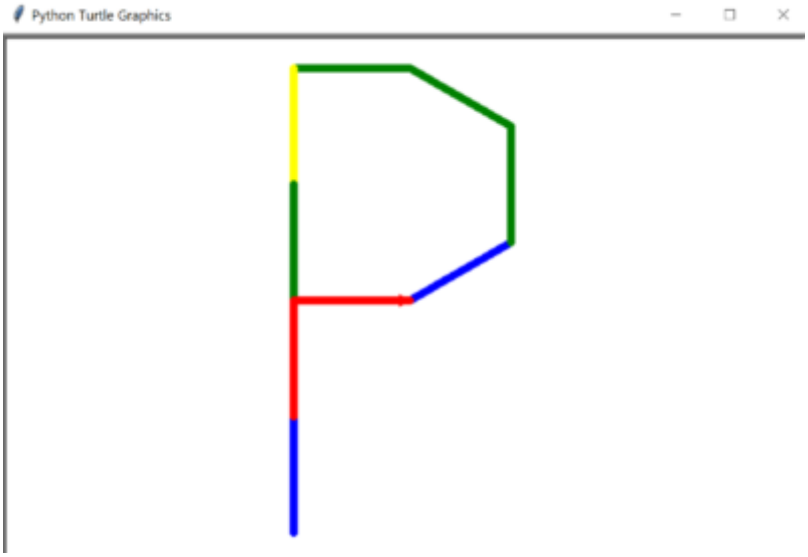
- strzałka do góry, przemieszcza żółwia o 100
- strzałka w prawo, obraca go w prawo o 30 stopni
- strzałka w lewo, obraca go w lewo o 30 stopni
- 'q' kończy działanie naszego programu

W tym celu, zdefiniowaliśmy 4 funkcje, które wykonują odpowiednie akcje. Aby program był atrakcyjniejszy, przy każdym ruchu, żółw losuje swój kolor.

```
import turtle
t = turtle.Turtle()
def up():
t.forward(100)
def left():
t.left(30)
def right():
t.right(30)
def bye():
turtle.bye()
```

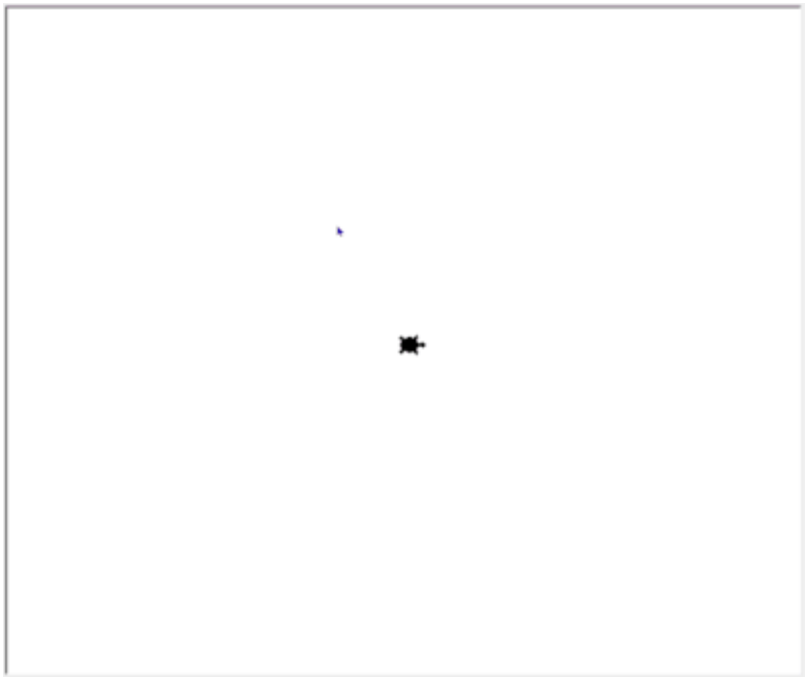
```
turtle.onkey(up,'Up')
turtle.onkey(left,'Left')
turtle.onkey(right,'Right')
turtle.onkey(bye,'q')
turtle.listen()
turtle.mainloop()
```

W rezultacie, otrzymaliśmy możliwość sterowania naszym żółwiem, co wykorzystaliśmy do próby napisania literki 'P'. 'P' jak Python



38.1 obsługa zdarzeń

Do tej pory pokazywaliśmy podstawowe operacje w Turtle, które umożliwiały nam rysowanie różnych kształtów. Tym razem zobaczymy jak obsłużyć zdarzenia. Przykładowo naciśniecie klawisza lub kliknięcie myszki. Tym samym nasze aplikacji będą mogły wchodzi w interakcję z użytkownikiem.



39 Przypisanie zdarzenia do klawisza

Najprostszy kod, który obsługuje naciśnięcie przez użytkownika klawisza, wygląda tak:

```
import turtle
window = turtle.Screen()
t = turtle.Turtle()
def up():
pass
window.onkey(up,'Up')
window.listen()
window.mainloop()
```

Najważniejsza jest funkcja window.onkey(). Jako drugi parametr przyjmuje zdarzenie które chcemy obsłużyć, czyli jaki klawisz nas interesuje. W tym przypadku jest to strzałka do góry.

Jako drugi parametr, przyjmuje funkcję która ma się wykonać. Czyli w momencie kiedy użytkownik naciśnie strzałkę do góry, wykona się funkcja up().

Na początku kodu, importujemy bibliotekę turtle, oraz tworzymy obiekt window = turtle.Screen()

Na końcu kodu, wykonujemy jeszcze 2 ważne funkcje:

- window.listen() – informujemy turtle, że ma nasłuchiwać na zdarzenia
- window.mainloop() – informujemy turtle, aby nie zamykał programu, po pojawieniu się klawisza, ale by dalej program działał

39.1 Obsługa wielu klawiszy w turtle

Jeżeli chcemy obsługiwać wiele klawiszy, robimy to analogicznie. Za pomocą funkcji onkey(), przypisujemy więcej funkcji do większej ilości klawiszy. Przykładowo:

```
def up():
t.forward(100)
def left():
t.left(45)
def right():
t.right(45)
def bye():
turtle.bye()
window.onkey(up,'Up')
window.onkey(left,'Left')
window.onkey(right,'Right')
window.onkey(bye,'q')
```

Powyższy kod:

- przypisuje funkcje do naciśnie strzałek
- przypisuje funkcje która zamyka program do klawisza 'q’
- Jeżeli użytkownik naciśnie strzałkę do góry, to żółw pójdzie do przydu. Przy strzałkach w bok, żółw skręci.

39.2 Obsługa kliknięcia myszy w turtle

Do tej pory używaliśmy funkcji window.onkey(), aby przypisać funkcję która ma się wykonać do konkretnego klawisza.

Aby obsłużyć kliknięcie myszy, mamy do dyspozycji funkcję window.onclick(), która przyjmuje jako parametr funkcję która ma się wykonać jeżeli użytkownik kliknie. Dodatkowo, funkcja ta, otrzymuje jako parametry, miejsce kliknięcia myszy, w postaci współrzędnych x, y.

Zobaczmy jak to wygląda w kodzie:

```
def click(x,y):
t.goto(x,y)
window.onclick(click)
```

1. Użyliśmy funkcji window.onclick(), i przypisaliśmy nowo utworzoną funkcję click
2. Funkcja klik, przyjmuje parametry x, y
3. W momencie kiedy użytkownik kliknie, miejsce kliknięcia jest przekazywane do funkcji
4. I w naszym przykładzie, żółw przemieści się do wskazanego miejsca

39.3 Całość kodu

Aby dodatkowo, uatrakcyjnić program, przed każdorazowym przemieszczeniem się żółwia, żółw zmienia kolor na losowo wybrany.

```
import turtle
import random
window = turtle.Screen()
t = turtle.Turtle()
t.pensize(7)
t.shape('turtle')
def up():
t.color(random.choice(['red','blue','yellow','green']))
t.forward(100)
def left():
t.left(30)
def right():
t.right(30)
def bye():
turtle.bye()
def click(x,y):
t.color(random.choice(['red','blue','yellow','green']))
t.goto(x,y)
window.onkey(up,'Up')
window.onkey(left,'Left')
window.onkey(right,'Right')
window.onkey(bye,'q')
window.onclick(click)
window.listen()
window.mainloop()
```

A my w rezultacie otrzymujemy prosty program, reagujący na naciśnięcie strzałek, klawisza 'q’ oraz kliknięcie myszki: