Python - początki

> Motywacja jest tym, co pozwala Ci zacząć.
> Nawyk jest tym, co pozwala Ci wytrwać.
> ~Autor nieznany

1. Stringi

Nie ma nic prostszego od zadeklarowania zmiennej typu string - po nazwie zmiennej wpisujemy treść w cudzysłowie (lub apostrofie) jaką zmienna ma zawierać.

```
suchar1 = "Co robi strażak na siłowni? Spala kalorie."
suchar2 = 'Ulubiony owoc żołnierza? Granat.'
```

Żeby wyświetlić zawartość zmiennej, używamy funkcji **print()**

```
print(suchar1)
```

Jeśli chcesz wpisać do zmiennej treść zawartą w kilku liniach, użyj potrójnego pojedynczego cudzysłowia! (brzmi śmiesznie, na pewno śmieszniej niż potrójny apostrof) Wtedy przejście do nowej linii odbywa się poprzez naciśnięcie ENTER.

```
suchar3 = '''Jak piją matematycy?
Na potęgę.'''
```

Obsługiwanie cudzysłowia zawartego w tekście? Nic prostszego! Wystarczy, że przed cudzysłowiem/apostrofem będącym w tekście dasz tzw. bekslesza - " \ " (prawy ukośnik, jak kto woli).

```
tekst1 = 'I ja mu wtedy mówię: \"Mikołaj, a może ty zacznij prezenty przynosić?\"'
print(tekst1)

# w konsoli wyświetli się:
# I ja mu wtedy mówię: "Mikołaj, a może ty zacznij prezenty przynosić?"
```

Zmienne zawarte w tekście - formatowanie stringów

Python pozwala na wypisywanie tekstu ze zmienną w nim zawartą. Tłumacząc na przykładzie wygląda to tak:

```
mojWynik = 1000
wiadomosc1 = 'Masz aktualnie {} punktów!'
print(wiadomosc1.format(mojWynik))
#w konsoli zostanie wypisane: Masz aktualnie 1000 punktów!
#
#funkcji format można użyć przy deklaracji zmiennej jak w przykładzie
#poniżej.
wiadomosc2 = 'Masz aktualnie {} punktów!'.format(mojWynik)
print(wiadomosc2)
#w konsoli zostanie wypisane: Masz aktualnie 1000 punktów!
```

Możesz również ponumerować wpisywane zmienne:

```
print("{1} + {2} + {3}".format(5,32,73))
#w konsoli zostanie wypisane: 5 + 32 + 73
```

Lub nazwać je według własnego uznania:

```
print("Wartość X: {x_val}, wartość Y: {y_val}".format(x_val=2, y_val=3))
#w konsoli zostanie wypisane: Wartość X: 2, wartość Y: 3
```

Istnieje również **starsza metoda** formatowania tekstu korzystająca z operatora **%.** Przy procencie wpisujemy typ zmiennej.

```
mojeImie = "Jan"
mojWynik = 999
wiadomosc = "%s, masz aktualnie %d punktów!"
print(wiadomosc % (mojeImie, mojWynik))
#w konsoli zostanie wypisane: Jan, masz aktualnie 999 punktów!
```

%s odpowiada za stringi, %d za integery --- jest to starsza metoda, ale działa tak samo dobrze

Mnożenie tekstu - jeśli z jakiegoś powodu będziecie potrzebować pomnożenia jakiegoś tekstu ileś razy - jest to w Pythonie możliwe! Zademonstruję na przykładzie:

```
print(10 * "a")
#w konsoli zostanie wypisane: aaaaaaaaaa
```

Prawda, że bardzo przydatne?

2. Listy w pythonie

Nieodłącznym elementem naszego życia są listy. Każdy ma do czynienia z co najmniej jedną listą: zadań, obecności, zakupów itd. Dobra to jak wiemy, że mamy z listami do czynienia to teraz może określmy czym to jest.

```
Lista - struktura danych służąca do reprezentacji zbiorów dynamicznych,
w której elementy są ułożone w liniowym porządku.
```

Dobra to tyle w temacie definicji z wikipedii. Łatwiej powiedzieć, że **lista** to **zbiór elementów** ułożonych w kolejności.

W pythonie listy tworzy się w bardzo prosty sposób. Oto on:

```
shopping_list = ['pomidory', 'cytryny', 'woda']
```

Osoby znające inne języki programowania stwierdzą, że to jest tablica. Guzik prawda. Dlaczego? Tablica to zbiór danych **tego samego typu**, w którym poszczególne elementy dostępne są za pomocą indeksów oraz rozmiar tablicy jest z góry ustalony. Lista w pythonie pozwala nam przechowywać dane różnego typu, jak również możemy dodawać do niej elementy.

```
 numbers_and_strings = [1, 2, 3, 'Witaj', 'w', 'LearnIT']
```

Z list dane możemy odczytywać na różne sposoby:

Możemy odwołać się do zmiennej po indeksie.

```
numbers_and_strings = [1, 2, 3, 'Witaj', 'w', 'LearnIT']

print(numbers_and_strings[2]) # wypisana zostanie 3
```

Co? Przecież na drugiej pozycji jest 2, a nie 3. To prawda, ale listy numerowane są od zera i dlatego 1 ma indeks 0, 2 ma indeks 1, a 3 ma indeks 2. Logiczne prawda? Nie ma za co.

Możemy wypisać wszystkie zmienne z podanego zakresu:

```
numbers_and_strings = [1, 2, 3, 'Witaj', 'w', 'LearnIT']

print(numbers_and_strings[3:5]) # wypisana zostanie: Witaj, w, LearnIT
```

Wiemy już jak odczytywać z list. Teraz pora nauczyć się dodawać do listy wartości.

Wartości możemy dodawać na wiele sposobów:

Pierwszą możliwością jest dodanie wartości na końcu listy. Aby to zrobić użyjemy funkcji **append(***element***)**.

```
list = ['Co robi zaatakowany kucharz?']

list.append('Wzywa posiłki') #dodajemy wartość do naszej listy
```

Drugą możliwością jest wstawienie elementu na żądanej pozycji. Możemy to zrobić za pomocą funkcji **insert(***index,element***)**.

```
letters = ['x', 'y', 'z']
letters.insert(1,'w') # na pozycji 1 zostanie wpisana litera w
                      # pozostałe elementy zostaną przesunięte o 1
print(letters[2]) # wypisana zostanie litera y
```

Pamiętacie?

Kolejną opcją jest dodanie innej listy. Aby to zrobić musimy użyć funkcji **extend(**lista2**)**.

```
numbers_and_strings = [1, 2, 3]
strings = ['Witaj', 'w', 'LearnIT']

numbers_and_strings.extend(strings)
```

Jak się pewnie domyślacie, skoro można dodawać, to można też usuwać wartości z tablicy. Robimy to tak:

```
numbers = [1, 2, 3]

del numbers[1] # po usunięciu lista wygląda tak: [1, 3]
```

Ostatnią rzeczą jaką możemy robić z listami to operacje arytmetyczne. *Jupii!*

Pierwszą operacją jest… dodawanie!

Tak, dwie listy możemy do siebie dodać. Rezultat będzie taki sam jak użycie funkcji **extend(***lista2***)**. Oznacza to, że możemy dodawać do siebie tylko listy. Próba dodanie wartości spowoduje błąd.

```
numbers = [1, 2, 3]
strings = ['Witaj', 'w', 'LearnIT']

numbers_and_strings = numbers + strings

print(numbers_and_strings)
#wypisana zostanie taka lista: [1, 2, 3, 'Witaj', 'w', 'LearnIT']
```

Kolejne jest mnożenie. Co tu dużo pisać, zobaczcie sami:

```
list = [1, 2]
print(list1 * 5)
# rezultat: [1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

Są jeszcze dwie operacje arytmetyczne nam znane. Jest to mnożenie i dzielenie, ale jak będziemy chcieli wykonać taką operację na liście to jedyne co zobaczymy to błąd. Np. taki:

```
list1 = [1, 2]
list1 / 20

TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

3. Tuple (tupel vs lista)

Tuple są jak listy, ale z użyciem zwykłych nawiasów. **( )**

```
liczby_fibiego = (1, 2, 3, 5, 8)
```

Główną różnicą między tuplami a listami jest to, że tupli nie można **zmienić** po utworzeniu. Czyli jeśli chcielibyśmy zmienić pierwszą liczbę z naszego "tupla", dostaniemy ładny czerwony error:

```
liczby_fibiego[0] = 9
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Więc po co w takim razie używać ograniczonego typu danych, jakim są tuple? A na przykład po to, gdy chcemy mieć pewność że dana lista (tudzież "tupel") nigdy się nie zmieni. Jeśli stworzymy tuplę z dwoma elementami w środku, możemy mieć pewność, że zawartość nigdy się nie zmieni.
Czas na parę przykładów:

```
tupel0 = ()
type(tupel0) #type() to funkcja wypisująca w konsoli typ zmiennej
#wypisane zostanie: <class 'tuple'>

tupel1 = 'a',
type(tupel1) #wypisane zostanie: <class 'tuple'>

tupel2 = ('a') #<-- uwaga na ten przykład
type(tupel2) #wypisane zostanie: <class 'str'>

tupel2_1 = ('a',) #tupel2 staje się tuplem po dodaniu przecinka
#jeśli chcesz stworzyć 1-elementową tuplę, pow. sposób jest rekomendowany
type(tupel2_1) #wypisane zostanie: <class 'tuple'>
```

Mimo braku możliwości zmiany zawartości tupli, można do nich dodawać kolejne wartości i łączyć dwa tuple (dodane zostaną na końcu tupla):

```
tupel3= (1, 'Ala ma kota')
tupel3 += (2,) #<-- dodajemy tuple, nie string lub liczbę
print(tupel3) #w konsoli zostanie wypisane: (1, 'Ala ma kota', 2)
```

4. Mapy, które nigdzie Cię nie zaprowadzą

W pythonie mapy (nazywane również słownikami) są kolekcją elementów taką jak listy czy tuple. Różnica jest taka, że w mapach każdy element ma *klucz* i

wartość. Dla przykładu powiedzmy, że mamy listę osób i ich ulubionych sportów. Moglibyśmy użyć do tego listy i wpisywać raz imię naszej osoby, a zaraz za nią sport. W taki sposób:

```
favorite_sports = ['Patryk Pogorzelczyk', 'Piłka nożna',
                   'Mikołaj Korbanek', 'Koszykówka']
```

Przy takiej liście nie ma problemu, jesteś w stanie zapamiętać, że Patryk jest na pozycji 0, a Mikołaj na pozycji 1. Tylko co zrobisz jak będziesz posiadać listę, w której będzie 100, albo 1000 osób? Tu właśnie z pomocą przychodzą mapy. Imię i nazwisko może posłużyć nam za klucz, a ulubiony sport danej osoby jako wartość. Kod wygląda tak:

```
favorite_sports = {'Patryk Pogorzelczyk' : 'Piłka nożna',
                   'Mikołaj Korbanek' : 'Koszykówka'}
```

Używamy tutaj dwukropka, aby oddzielić klucz od wartości i każda taka para jest oddzielona od siebie przecinkiem. Zauważ również, że mapa jest pomiędzy nawiasami klamrowymi **{ }**.

Powyższą mapę można przedstawić w następujący sposób:

Key	**Value**
Patryk Pogorzelczyk	Piłka nożna
Mikołaj Korbanek	Koszykówka

Aby wyświetlić jakąś wartość z naszej mapy musimy odwołać się do niej po kluczu.

```
favorite_sports = {'Patryk Pogorzelczyk' : 'Piłka nożna',
                   'Mikołaj Korbanek' : 'Koszykówka'}
print(favorite_sports['Patryk Pogorzelczyk'])
# wypisane zostanie Piłka nożna
```

Wartości z map można również usuwać. Robimy to w taki sam sposób jak przy listach:

```
del favorite_sports['Patryk Pogorzelczyk']
# teraz mapa wygląda tak: {'Mikołaj Korbanek' : 'Koszykówka'}
```

Zmiana wartości również wygląda tak samo jak w listach:

```
favorite_sports['Patryk Pogorzelczyk'] = 'Siatkówka'
# teraz mapa wygląda tak: {'Patryk Pogorzelczyk' : 'Siatkówka',
#                          'Mikołaj Korbanek' : 'Koszykówka'}
```

Na listach nie można wykonywać operacji arytmetycznych. A szkoda…

5. Podsumowanie typów danych w pythonie

Dobra, to już nauczyłeś się jak w Pythonie można pracować z tekstem (przy użyciu stringów), oraz jak wykorzystuje się listy i tuple do obsługie wielu elementów. Przeczytałeś, że do listy możesz dodawać i usuwać elementy, ale w tuplach nie możesz zmieniać wartości. Nauczyłeś się również, jak używać map do przechowywania wartości za pomocą klucza, który je identyfikuje.

6. Zagadki programistyczne

#1 Ulubione

Stwórz listę o nazwie **hobbies** z twoimi ulubionymi hobby. Następnie stwórz listę o nazwie **foods** z twoimi ulubionymi potrawami. Na koniec stwórz listę **favorites**, która będzie zawierać listę hobby oraz potraw. Teraz wypisz tą listę.

#2 Powitanie

Stwórz dwie zmienne, jedna ma przechowywać twoje imię, a druga nazwisko. Teraz stwórz zmienną tekstową, która będzie zawierała wiadomość razem z miejscem na zmienne. Przykładowe rozwiązanie: “Witaj, Mikołaj Korbanek!”.

7. Kiedy nawet program ma pytania

W programowaniu bardzo często padają pytania typu “tak/nie” i bazując na odpowiedzi coś robimy. Na przykład możemy zadać pytanie: “Masz więcej niż 18 lat?” i jeśli odpowiedź brzmi: “tak” to możemy napisać: “Jesteś dorosły!”. Takie pytania nazywamy warunkami i możemy umieścić je w warunku logicznym **IF**. Warunki mogą być bardziej skomplikowane niż pojedyncze pytanie, oraz nie musimy ograniczać się do jednej odpowiedzi. Wszystkiego dowiesz się w tym punkcie.

Wyrażenie IF

Wyrażenia logiczne w pythonie mogą wyglądać tak:

```
age = 18
if age >= 18:
      print('Jesteś dorosły!')
```

Wyrażenie if w pythonie zbudowane jest z słowa kluczowego **if**, po którym wpisujemy warunek, a na końcu jest dwukropek **:**. Linie kodu, które mają zostać wykonane tylko gdy warunek jest prawdą (**true**) muszą być w bloku.

Bloki kodu w pythonie

Blok kodu jest zestawem poleceń programistycznych. Np. Kiedy **if age >= 18:** jest prawdą chcesz wypisać coś więcej niż “Jesteś dorosły!”. Możesz to zrobić w taki sposób:

```
age = 18
if age >= 18:
      print('Jesteś dorosły!')
      print('Teraz jesteś odpowiedzialny za to co robisz!')
```

Ten blok kodu składa się z dwóch poleceń “print”, które zostaną wypisane tylko wtedy gdy wiek jest większy lub równy 18. Każda linia kodu w bloku musi być poprzedzona 4 spacjami lub tabem (w większości **IDE** tab odpowiada 4 spacją). Spójrzmy jeszcze raz na kod z widocznymi spacjami:

```
age = 18
if age >= 18:
····print('Jesteś dorosły!')
····print('Teraz jesteś odpowiedzialny za to co robisz!')
```

W pythonie białe znaki takie jak spacja czy tab są bardzo znaczące. Kod "na tej samej wysokości" jest grupowany w blok i kiedy zaczynasz nową linię z większą ilością spacji niż poprzednia, to zaczynasz nowy blok.

```
linia kodu                                          #blok 1
linia kodu                                          #
linia kodu                                          #
    linia kodu                          #blok 2     #
    linia kodu                          #           #
    linia kodu                          #           #
        linia kodu              #blok 3 #           #
        linia kodu              #       #           #
        linia kodu              #       #           #
    linia kodu                          #           #
```

Grupujemy instrukcje w blok, ponieważ chcemy, aby zostały uruchomione razem.
Zmieniając wcięcie tworzysz nowe bloki. Powyższy przykład pokazuje 3 bloki kodu stworzone po prostu przez zmianę wcięcia.

```
linia kodu                                          #blok 1
linia kodu                                          #
linia kodu                                          #
    linia kodu                          #blok 2     #
    linia kodu                          #           #
    linia kodu                          #           #
linia kodu                                          #
    linia kodu                          #blok 3     #
    linia kodu                          #           #
    linia kodu                          #           #
```

Tutaj blok 2 i 3 mają to samo wcięcie, ale uważane są za dwa różne bloki, ponieważ istnieje blok z mniejszą liczbą wcięć (mniej spacji) między nimi.

Żeby pokazać, że każda spacja ma znaczenie posłużę się prostym przykładem:

```
age = 18
if age >= 18:
    print('Jesteś dorosły!')
      print('Teraz jesteś odpowiedzialny za to co robisz!')

#ten kod spowoduje błąd wcięcia (indentation error)
```

Więc kiedy zaczniesz blok z czterema spacjami powinieneś się tego konsekwentnie trzymać.

Warunki pomagają nam porównywać rzeczy

Warunek to instrukcja programistyczna, która porównuje rzeczy i mówi nam czy kryteria ustalone przez porównanie są prawdziwe (**true**) lub fałszywe (**false**). Na przykład:
age > 10 jest to warunek i można to przeczytać w taki sposób: "*Czy wartość zmiennej age jest większa niż 10?*" W pythonie używamy symboli nazywanych **operatorami** do utworzenia naszych warunków.

Symbol	Definicja
==	Równe
!=	Nie równe
>	Większe niż
<	mniejsze niż
>=	Większe lub równe
<=	Mniejsze lub równe

Jeśli to

Oprócz użycia instrukcji **if**, aby coś zrobić, gdy stan jest spełniony (**true**), możemy również użyć tej instrukcji, aby coś zrobić gdy warunek nie jest prawdziwy. Na przykład możemy wydrukować jedną wiadomość, gdy masz 18 lat, a drugą gdy nie masz 18 lat. Wystarczy użyć instrukcji **if-else**, która mówi nam: "Jeśli coś jest prawdą, zrób to; albo, zrób to".

```
age = 19
if age == 18:
    print("Masz 18 lat!")
else:
    print("Nie masz 18 lat.")
```

Ponieważ ustawiliśmy naszą zmienną age na 19, to wypisany zostanie tekst o treści: "Nie masz 18 lat.".

IF i ELIF

Możemy iść o krok dalej za pomocą instrukcji **elif** (która jest skrótem od else if). Ta instrukcja różni się od **else** tym, że może być użyta więcej niż jeden raz.

```
age == 11
if age == 10:
      print("Masz 10 lat.")
elif age == 11:
      print("Masz 11 lat.")
else:
      print("I co teraz? Nie wiem ile masz lat")
```

W tym przykładzie instrukcja **if** występuje w drugiej linii i sprawdza czy zmienna **age** jest równa 10. Nie jest to prawdą więc omijamy blok kodu "należący" do tej instrukcji warunkowej i idziemy dalej. Napotykamy **elif** i sprawdzamy czy nasza zmienna jest równa 11. Okazuje się, że jest, więc wypisujemy komunikat: "Masz 11 lat.". W tym momencie kończymy sprawdzanie warunków. Instrukcja print znajdująca się po **else** nie zostanie wypisana.

Łączenie warunków

Co mamy zrobić jeśli chcemy sprawdzić więcej niż jeden warunek?
Warunki możemy łączyć używając słów kluczowych **and** i **or**.

Przykład rozbudowanego sprawdzania warunków:

```
age = 11

if age == 10 or age == 11 or age == 12:
    print("Masz 10, 11 albo 12 lat.")
```

Powyższy kod wypisze tekst tylko wtedy gdy zmienna age będzie równa 10 lub 11 lub 12.

Kolejną możliwością jest napisanie tego if'a w taki sposób:

```
age == 11
if age >= 10 and age <=12
     print("Masz 10, 11 albo 12 lat.")
```

Ten warunek będzie zwracał prawdę (**true**) gdy wiek będzie z przedziału od 10 do 12 obustronnie domkniętego (*w matematyce zapisane tak: <10; 12>*)

Zmienne bez wartości

Do wartości możemy również przypisać puste wartości. W pythonie pusta wartość to **None** (*jeśli miałeś do czynienia z innymi językami programowania None to odpowiednik NULL)*. Ważne, żeby pamiętać, że wartość **None** różnie się od wartości 0, ponieważ jest to brak wartości a nie wartość 0. **None** jest niczym.

Możesz wykorzystać przypisanie **None** do zmiennej, kiedy wiesz, że będziesz jej potrzebować w dalszej części programu, ale wszystkie zmienne chcesz zdefiniować na początku programu, ponieważ ułatwi Ci to odczytanie ich nazw.

Możemy też sprawdzić przy użyciu instrukcji **if** czy nasza zmienna jest równa **None:**

```
value = None
if value == None
    print('Zmienna nie posiada wartości')
```

Jest to przydatne, gdy chcesz obliczyć wartość dla zmiennych, które wartości nie posiadają.

Różnice pomiędzy tekstem i liczbą

Użytkownik jest osobą, która korzysta z naszego programu za pomocą np. klawiatury. Wszystko co na niej pisze jest interpretowane przez program jako ciąg znaków. Obojętnie, czy jest to klawisz z cyfrą "1" czy z literą "W".
Jaka jest różnica pomiędzy liczbą 10 a tekstem '10'? Dla nas wygląda to tak samo, z tą tylko różnicą, że jedno jest w cudzysłowie, ale dla komputera są to dwie **bardzo** różne rzeczy.

```
age = 10
if age == 10:
   print('Masz 10 lat')
# wypisany zostanie tekst 'Masz 10 lat'
#
# teraz sprawdźmy co się stanie jak podamy '10'

age = '10'
if age == 10:
    print('Masz 10 lat')
# cisza, nic się nie stanie
```

W drugim przypadku kod instrukcji **print** nie zadziała, ponieważ Python nie widzi liczby w cudzysłowie. Na szczęście w pythonie są magiczne funkcje, które mogą przekształcić ciąg w liczby i liczby w ciągi.

Na przykład możemy zmienić tekst ‘10’ w liczbę za pomocą funkcji **int()**:

```
age = '10'
converted_age = int(age)
```

Zmienna **converted_age** przechowuje liczbę 10.

Aby zamienić liczbę w tekst korzystamy z funkcji **str()**:

```
age = 10
converted_age = str(age)
```

8. Podsumowanie

W tym temacie nauczyłeś się jak pracować z wyrażeniami logicznymi, jak tworzyć bloki kodu. Dowiedziałeś się też jak budować rozbudowane wyrażenia logiczne z wykorzystaniem **elif** i **else**, oraz jak łączyć warunki za pomocą słów kluczowych **and** i **or**. Odkryłeś również, że nic w pythonie też ma znaczenie i możemy je przedstawić za pomocą wartości **None**. Ostatnie czego się dowiedziałeś to jak zmieniać typy danych.

9. Zagadki programistyczne

#3 Czy jesteś bogaty?

```
Jaki będzie rezultat poniższego kodu?
money = 2000
if money > 1000:
    print("I'm rich!!")
else:
    print("I'm not rich!!")
      print("But I might be later...")
```

#4 Ciastka!

Utwórz instrukcję, która sprawdza czy liczba ciastek jest mniejsza od 100 lub większa od 500. Jeśli warunek jest prawdziwy niech program wypisze “Za mało lub za dużo ciastek.”.

#5 Prawidłowy numer

Stwórz instrukcję warunkową, która sprawdza czy kwota pieniędzy zawarta zmiennej wynosi od 100 do 500 lub od 1000 do 5000.

#6 Mogę walczyć z tymi ninja

Napisz instrukcję, która wypisuje ciąg “To za dużo.”, jeśli zmienna ninja jest większa od 50, jeśli zmienna jest większa niż 30, ale mniejsza od 50 to wypisuje “To będzie ciężka walka, ale dam radę.” i jeśli jest mniejsz niż 30 to wypisuje “Mogę walczyć z tymi ninja!”.

10. Pętle

Nie ma nic gorszego od pisania tego samego "w kółko". Na pomoc programistom przyszły pętle, dzięki którym nie muszą pisać powtarzalnego kodu i które pozwalają im zdrzemnąć się choć na chwilę.

Pętle for

Zamiast pisać pięciokrotnie kolejne liczby za pomocą funkcji print(), możesz użyć pętli for, żeby się nie powtarzać:

```
for x in range(1,6): #L1
    print("hello {}".format(x)) #L2
#w konsoli zostanie wypisane:
#hello 1
#hello 2
#hello 3
#hello 4
#hello 5
```

Funkcja range() przyjmuje dwa argumenty - początek i koniec liczenia.
L1 (linia 1) w takim razie mówi:

- zacznij liczyć od 1 i skończ przed osiągnięciem 6 (ciekawostka - range() może przyjąć również wartości ujemne, druga liczba zawsze musi być większa)
- dla każdej liczby (czyli dla każdego obiegu pętli), zachowaj jej wartość w zmiennej x

Potem wykonuje się L2 (omawiana linia nr2, warto nauczyć się znaczenia tych skrótów, będziemy tak opisywać poszczególne linie programu w dalszych notatkach).

W taki sam sposób możemy wypisać listy, tuple lub mapy(zostaną wypisane nazwy kluczy bez wartości):

```
lista_zakupow = ['chleb', 'maslo', 'czosnek', 'śledź']
for rzecz in lista_zakupow: #L2
    print(rzecz)
#w konsoli zostanie wypisane:
#chleb
#maslo
#czosnek
#śledź
```

L2 - dla każdej rzeczy zawartej w liście zakupów, zachowaj jej wartość w zmiennej *rzecz*

Pętla while

Pętle for mają określoną długość, pętli while możemy za to użyć, gdy nie wiemy do końca kiedy owa pętla powinna się skończyć.
Wyobraź sobie, że wchodzisz na klatkę schodową do swojego mieszkania. Wiesz, że potrzebujesz 20 kroków do dojścia na swoje piętro.
Można to zapisać pętlą for:

```
for nrkroku in range(1,21):
    print("krok {}".format(nrkroku))
```

A teraz wyobraź sobie że wchodzisz na klatkę schodową Burdż Chalifa do pokoju swojej ciotki. Nie wiesz ile będziesz potrzebował kroków, ale wiesz, że jedno piętro ma 20 kroków (powiedzmy, że podali taką informację na parterze) a pokój ciotki znajduje się na 132 piętrze.
Można w tym przypadku zapisać to pętlą while:

```
pietro = 0 #L1
krok = 0 #L2
while pietro != 132: #L3
    krok += 1 #L4
    if krok % 20 == 0: #L5
        pietro += 1 #L6
    print(krok) #L7
```

L1,L2 - deklaracja zmiennych liczbowych
L3 - dopóki piętro nie będzie równe 132:
L4 - dodaj 1 do liczby kroków (wykonaj krok)
L5 - jeśli liczba kroków podzielona przez 20 zwraca resztę 0 (czyli jeśli weszliśmy na kolejne piętro)
L6 - dodaj 1 do liczby pięter
L7 - wypisz wykonaną dotychczas liczbę kroków.

Żeby wypisać sumę kroków dopiero po osiągnięciu piętra ciotki, wykonaj linię przerwy i funkcję print() napisz bez wcięcia, jak na przykładzie poniżej:

```
pietro = 0 #L1
krok = 0 #L2
while pietro != 132: #L3
    krok += 1 #L4
    if krok % 20 == 0: #L5
        pietro += 1 #L6

print(krok) #L7
```

Pętla while może sprawdzać kilka warunków żeby się wykonać:

```
pietro = 0
krok = 0
while pietro != 132 and czy_zmeczony == false: #L3
    krok += 1
    if krok % 20 == 0:
        pietro += 1

print(krok)
```

L3 - sprawdzamy tutaj, czy nadal nie jesteśmy na 132. piętrze oraz czy nie jesteśmy zmęczeni.
Powyższy program nie zadziała, gdyż przed pętlą while nie mamy zadeklarowanej zmiennej *czy_zmeczony*. Możemy wprowadzić kilka instrukcji warunkowych, w których moglibyśmy zmienić wartość *czy_zmeczony* z false na true, ale to pozostawiamy Waszej inwencji twórczej.

Przerywanie/kontynuowanie obiegu pętli - instrukcje **break** i **continue**

Podczas kolejnych obiegów w pętli while (lub for) mogą dziać się rzeczy, którym chcemy np. zapobiec lub je pominąć. Przykład:

```
while True: #L1
    dużo kodu1
    dużo kodu1
    if jakas_wartosc1 == 20: #L2
        continue
    dużo kodu2
    dużo kodu2
    dużo kodu2
    if jakas_wartosc2 == True: #L3
        break
    dużo kodu3
```

L1 - nieskończona pętla while, można ją przerwać tylko dzięki instrukcji break.

L2 - sprawdzamy czy zmienna jest równa np. 20. jeśli tak, pomijamy wszystko co jest po instrukcji warunkowej (“dużo kodu2” i druga instrukcja warunkowa nie wykonają się)

L3 - sprawdzamy czy zmienna przyjmuje wartość prawdy (true). Jeśli tak, pętla while przerywa się (“dużo kodu3” nie wykona się)

Zagnieżdżanie pętli

Jak można było zauważyć, w pętlach możemy zagnieżdżać instrukcje warunkowe. Możemy zrobić to samo z innymi pętlami. Spróbujmy w takim razie przerobić nasz kod, by użyć pętli w pętli:

```
pietro = 0
krok = 0
while pietro != 132:
    for nrkroku in range(1,21): #L4.1
        krok += 1 #L5.1
    pietro += 1

print(krok)
```

Jako że wiemy, że piętro zwiększa się co 20 kroków, możemy naszą instrukcję warunkową (L5) zastąpić pętlą for (L4.1) - dla każdego liczby w zakresie od 1 do 20 włącznie, przypisz jej wartość do nrkroku.
L5.1 - zwiększamy sumę kroków o 1, czyli wykonujemy krok.

Ciekawscy mogą się w tym miejscu zapytać - *a co jeśli w pętli **for** zamiast **nrkroku** napisalibyśmy **krok**?*

Dobre pytanie, już spieszymy z odpowiedzią. Nie możemy tutaj napisać krok, gdyż wystąpi tzw. “konflikt nazw”. Co prawda program się wykona, ale po osiągnięciu 21. kroku (20. obieg pętli for) ilość kroków nie będzie się już zwiększać, ponieważ nie będzie znajdować się w zakresie podanym w funkcji range(). Ilość kroków wyniesie na koniec programu 21.

To tak, jakbyśmy założyli odrzutowy plecak na 1. piętrze i polecieli na nim aż do piętra ciotki. Wykonalne, ale wątpię że jakiś wieżowiec oferuje przeloty odrzutowym plecakiem w środku budynku… Można też na owym 1. piętrze wejść do windy, ale jest to mniej skrajny przykład. Każdy przecież przyzna, że odrzutowe plecaki są ciekawsze niż winda. No, może trochę mniej bezpieczne.

11.Podsumowanie pętli

Nauczyłeś się teraz: czym są pętle oraz do czego można ich używać. W środku pętli wykonujemy bloki kodu, w których możemy tą pętlę przerywać (lub pomijać dany obieg). Nauczyłeś się też używania pętli w środku pętli. No i przede wszystkim… Skończyłeś notatkę, woohoo! Teraz jeszcze szybkie zadanka i możesz w nagrodę (za ciężką pracę jaką wykonałeś) odprężyć się przy ulubionej książce lub grze :)

#7 - Liczby parzyste

Napisz pętlę,która wypisze wszystkie parzyste liczby mniejsze od Twojego wieku.
Przykładowe wyjście kodu:

```
2
4
6
8
10
12
14
16
```

#8 - Twoja waga na Księżycu

Jeśli stałbyś teraz na Księżycu, Twoja waga wynosiłaby 16,5% tego, ile ważysz na Ziemii. Jeśli Twoja waga zwiększałaby się o 2 kilo rocznie przez 15 lat, ile ważyłbyś na Księżycu po 15 latach? Napisz program używając pętli, która wypisze Twoją wagę dla każdego roku. Wypisz wartości za pomocą formatowania tekstu.
Przykładowe wyjście kodu:

```
"Moja waga na Ziemii: 100, moja waga na Księżycu: 16,5 -- rok 1"
"Moja waga na Ziemii: 102, moja waga na Księżycu: 16,83 -- rok 2"
itd...
```