Poniższy rozdział pochodzi z książki:

Charles R. Severance - "Python dla wszystkich: Odkrywanie danych z Python 3"

Pełna wersja podręcznika znajduje się na stronie https://py4e.pl/book

Rozdział 5

Pętle i iteracje

5.1. Aktualizowanie zmiennych

Częstym schematem w instrukcjach przypisywania jest instrukcja, która aktualizuje zmienną w taki sposób, że nowa wartość zmiennej zależy od starej.

```
x = x + 1
```

Oznacza to "pobierz bieżącą wartość `x`, dodaj 1, a następnie zaktualizuj `x` o nową wartość".

Jeśli próbujesz zaktualizować zmienną, która nie istnieje, otrzymasz błąd, ponieważ Python ewaluuje prawą stronę, zanim przypisze wartość do zmiennej po lewej stronie:

```
>>> z = z + 1
NameError: name 'z' is not defined
```

Zanim będziesz mógł zaktualizować zmienną, musisz ją *zainicjalizować*, zwykle za pomocą prostego przypisania:

```
>>> x = 0
>>> x = x + 1
```

Aktualizacja zmiennej poprzez dodanie wartości 1 nazywana jest *inkrementacją*; odjęcie wartości 1 nazywane jest *dekrementacją*.

5.2. Instrukcja `while`

Komputery często są używane do automatyzacji powtarzalnych zadań. Powtarzanie identycznych lub podobnych zadań bez popełniania błędów jest czymś, co komputery robią dobrze, a ludzie źle. Ponieważ w programach iteracja występuje często, Python oferuje kilka elementów, które ułatwiają z nią pracę.

Jedną z form iteracji w Pythonie jest instrukcja `while`. Poniżej znajduje się prosty program, który odlicza od pięciu, a następnie wyświetla "Odpalamy!".[1]

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Odpalamy!')
```

[1] Kopiując kod do interaktywnej sesji interpretera Pythona, należy pamiętać, że wcięty blok musi zakończyć się nową linią. W przeciwnym razie, w tym programie przy funkcji `print()` zostanie zgłoszony błąd "SyntaxError".

Instrukcję `while` można przeczytać prawie tak, jakby była zapisana w języku angielskim. Oznacza to: "Dopóki `n` jest większe od 0, wyświetl wartość `n`, a następnie zmniejsz wartość `n` o 1. Gdy dojdziesz do 0, opuść instrukcję `while` i wyświetl komunikat `Odpalamy!`".

Poniżej znajduje się bardziej ścisły opis przepływu sterowania w instrukcji `while`:

1. Ewaluuj warunek, otrzymując `True` lub `False`.
2. Jeśli warunek jest fałszywy, opuść instrukcję `while` i kontynuuj wykonywanie programu od następnej instrukcji.
3. Jeśli warunek jest prawdziwy, wykonaj ciało instrukcji `while`, a następnie wróć do kroku 1.

Taki przepływ sterowania nazywany jest *pętlą*, ponieważ trzeci krok zapętla się z powrotem do góry. Za każdym razem, gdy wykonujemy ciało pętli, nazywamy je *iteracją*. Dla powyższej pętli powiedzielibyśmy: "Miała ona pięć iteracji", co oznacza, że ciało pętli zostało wykonane pięć razy.

Ciało pętli powinno zmieniać wartość jednej lub więcej zmiennych, tak aby ostatecznie warunek stał się fałszywy i pętla się zakończyła. Zmienną, która zmienia się z każdym wykonaniem pętli i kontroluje, kiedy pętla się kończy, nazywamy *zmienną sterującą*. Jeśli nie ma zmiennej sterującej, pętla będzie się powtarzać w nieskończoność, powodując powstanie *nieskończonej pętli*.

5.3. Nieskończone pętle

Niekończącym się źródłem rozrywki dla programistów jest obserwacja, że wskazówki na szamponie "namydl, wypłucz, powtórz" są nieskończoną pętlą, ponieważ nie ma *zmiennej sterującej*, mówiącej ile razy tę pętlę należy wykonać.

W przypadku przykładu z odliczaniem możemy udowodnić, że pętla się zakończy, ponieważ wiemy, że wartość `n` jest skończona, i widzimy, że wartość `n` zmniejsza się z każdą iteracją pętli, więc ostatecznie musimy dotrzeć do 0. W innych przypadkach pętla jest oczywiście nieskończona, ponieważ w ogóle nie ma zmiennej sterującej.

Czasami nie wiesz, że już czas zakończyć pętlę, dopóki nie przejdziesz do połowy ciała instrukcji. W takim przypadku można celowo napisać nieskończoną pętlę, a następnie użyć instrukcji `break` do wyskoczenia z pętli.

Poniższa pętla jest oczywiście *nieskończoną pętlą*, ponieważ wyrażenie logiczne w instrukcji `while` jest po prostu stałą logiczną `True`:

```
n = 10
while True:
    print(n, end=' ')
    n = n - 1
print('Zrobione!')
```

Jeśli popełnisz błąd i uruchomisz ten kod, szybko nauczysz się, jak zatrzymać wymykający się spod kontroli proces Pythona w Twoim systemie[2] lub przypomnisz sobie, gdzie znajduje się przycisk wyłączania komputera. Program ten będzie działał w nieskończoność (lub aż do wyczerpania baterii w Twoim laptopie), ponieważ wyrażenie logiczne na górze pętli jest zawsze prawdziwe, bo jest stałą wartością `True`.

Mimo że jest to dysfunkcyjna nieskończona pętla, możemy nadal używać tego schematu do budowania użytecznych pętli. Musimy tylko starannie dodać kod do ciała pętli, tak aby wyraźnie z niej wyjść, używając polecenia `break` wtedy, gdy osiągniemy warunek wyjścia.

Na przykład, załóżmy, że chcesz pobierać od użytkownika dane wejściowe, dopóki nie wpisze "zrobione". Można by napisać taki program:

```
while True:
    line = input('> ')
    if line == 'zrobione':
        break
    print(line)
```

[2] Można spróbować wcisnąć kombinację klawiszy `<Ctrl+C>`.

```
print('Zrobione!')

# Kod źródłowy: https://py4e.pl/code3/copytildone1.py
```

Warunkiem pętli jest `True`, co jest zawsze prawdą, więc pętla działa wielokrotnie tak długo, aż trafi na instrukcję `break`.

Program za każdym razem prosi użytkownika o dane wyświetlając nawias trójkątny. Jeśli użytkownik wpisze "zrobione", to instrukcja `break` wyjdzie z pętli. W przeciwnym razie program powtórzy to, co użytkownik wpisał, i wróci na górę pętli. Oto przykładowe uruchomienie:

```
> no hejka
no hejka
> zakończone
zakończone
> zrobione
Zrobione!
```

Taki sposób pisania pętli `while` jest dość częsty, ponieważ można sprawdzić warunek w dowolnym miejscu pętli (nie tylko na górze) i można wyrazić warunek stopu w sposób twierdzący ("stop, gdy to się stanie"), a nie w sposób zaprzeczający ("nie przestawaj, dopóki to się nie stanie.").

Jeżeli chcemy natychmiast przerwać taką pętlę, to możemy wysłać sygnał do naszego programu kombinacją klawiszy `<Ctrl+C>`.

```
$ python3 copytildone1.py
> halo
halo
> nie wiem jak wyjść
nie wiem jak wyjść
> ^CTraceback (most recent call last):            # tutaj wciśnięto <Ctrl+C>
  File "copytildone1.py", line 2, in <module>
    line = input('> ')
KeyboardInterrupt

$
```

5.4. Kończenie iteracji przy pomocy `continue`

Czasami jesteś w iteracji pętli i chcesz zakończyć bieżącą iterację, tak aby natychmiast przejść do następnej iteracji. Aby przejść do następnej iteracji bez kończenia ciała pętli w bieżącej iteracji, użyj `continue`.

Oto przykład pętli, która wyświetla na ekranie swoje dane wejściowe aż do momentu, gdy użytkownik wpisze "zrobione", ale traktuje linie, które zaczynają się od znaku haszu (#), jako linie, które nie mają być wyświetlane (coś w rodzaju komentarzy Pythona).

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'zrobione':
        break
    print(line)
print('Zrobione!')

# Kod źródłowy: https://py4e.pl/code3/copytildone2.py
```

Oto przykładowy przebieg tego programu z dodanym `continue`.

```
> no hejka
```

```
no hejka
> # nie wyświetlaj tego
> wyświetl to!
wyświetl to!
> zrobione
Zrobione!
```

Wszystkie linie są ponownie wyświetlane na ekranie z wyjątkiem tej, która zaczyna się znakiem haszu, ponieważ kiedy wykonywana jest instrukcja `continue`, kończy to bieżącą iterację i przeskakuje z powrotem do instrukcji `while`, tak aby rozpocząć następną iterację, pomijając tym samym funkcję `print()`.

5.5. Definiowanie pętli przy użyciu `for`

Czasami chcemy przejść w pętli po *zbiorze* rzeczy, takich jak lista słów, wiersze w pliku lub lista liczb. Gdy mamy listę rzeczy do przetworzenia w pętli, możemy skonstruować pętlę *określoną*, używając instrukcji `for`. Nazywamy instrukcję `while` pętlą *nieokreśloną*, ponieważ po prostu zapętla się do momentu, gdy jakiś warunek przyjmie wartość `False`. Pętla `for` zapętla się przez znany zestaw elementów, więc przechodzi przez tyle iteracji, ile jest elementów w zestawie.

Składnia pętli `for` jest podobna do pętli `while` w tym, że mamy instrukcję `for` i ciało pętli:

```
friends = ['Józek', 'Gienek', 'Staszek']
for friend in friends:
    print('Szczęśliwego Nowego Roku', friend)
print('Zrobione!')
```

W terminologii Pythona zmienna `friends` jest listą[3] składającą się z trzech napisów, a pętla `for` przechodzi przez listę i wykonuje ciało raz dla każdego z tych trzech napisów w liście, co w rezultacie produkuje takie dane wyjściowe:

```
Szczęśliwego Nowego Roku Józek
Szczęśliwego Nowego Roku Gienek
Szczęśliwego Nowego Roku Staszek
Zrobione!
```

Tłumaczenie tej pętli `for` na ludzki język nie jest tak bezpośrednie jak w przypadku `while`, ale jeśli pomyślisz o zmiennej `friends` jako o *zbiorze*, to idzie to tak: "Uruchom instrukcje w ciele pętli `for` raz dla każdego przyjaciela (`friend`) znajdującego się *w* (`in`) zbiorze przyjaciół (`friends`)".

Patrząc na poniższą pętlę, słowa `for` i `in` są zastrzeżonymi słowami kluczowymi Pythona, a `friend` i `friends` są zmiennymi.

```
for friend in friends:
    print('Szczęśliwego Nowego Roku', friend)
```

W szczególności `friend` jest *zmienną sterującą* dla pętli `for`. Zmienna `friend` zmienia się w każdej iteracji pętli i kontroluje, kiedy pętla `for` zostanie zakończona. *Zmienna sterująca* przechodzi kolejno przez trzy napisy zapisane w zmiennej `friends`.

5.6. Schematy pętli

Często używamy pętli `for` lub `while`, by przejść przez listę elementów lub przez zawartość pliku i szukamy czegoś takiego jak największa lub najmniejsza wartość w analizowanych danych.

Tego typu pętle są na ogół konstruowane przez:

[3] Przyjrzymy się listom bardziej szczegółowo w dalszych rozdziałach.

- zainicjalizowanie jednej lub więcej zmiennych przed uruchomieniem pętli;
- wykonanie w ciele pętli pewnych obliczeń na każdym elemencie, ewentualnie zmiana wartości zmiennych w ciele pętli;
- patrzenie na wynikowe zmienne po zakończeniu pętli.

Użyjemy listy liczb, tak aby zademonstrować koncepcje i budowę wspomnianych na początku schematów pętli.

5.6.1. Pętle zliczające i sumujące

Na przykład, aby zliczyć ile elementów znajduje się na liście, moglibyśmy napisać następującą pętlę `for`:

```
count = 0
for itervar in [9, 41, 12, 3, 74, 15]:
    count = count + 1
print('Ile:', count)
```

Powyższy program demonstruje schemat obliczeń zwany *licznikiem*. Przed rozpoczęciem pętli ustawiamy zmienną `count` na zero, a następnie piszemy pętlę `for`, tak aby przebiegała przez listę liczb. Nasza *zmienna sterująca* ma nazwę `itervar` i choć nie używamy `itervar` w pętli, to kontroluje ona pętlę i powoduje, że ciało pętli jest wykonywane raz dla każdej z wartości występującej na liście.

W ciele pętli dodajemy 1 do bieżącej wartości `count` dla każdej wartości z listy. Podczas wykonywania pętli, wartość `count` jest liczbą elementów, które widzieliśmy "do tej pory".

Po zakończeniu wykonywania pętli wartość `count` jest liczbą wszystkich elementów. Łączna liczba elementów występujących na liście "wpada nam w ręce" na końcu pętli. Konstruujemy pętlę tak, aby mieć to, czego chcemy, gdy pętla się zakończy.

Oto kolejna podobna pętla, która oblicza sumę liczb z danej listy:

```
total = 0
for itervar in [9, 41, 12, 3, 74, 15]:
    total = total + itervar
print('Suma:', total)
```

W powyższej pętli używamy *zmiennej sterującej*. Zamiast po prostu dodawać jeden do `count` jak w poprzedniej pętli, tym razem podczas każdej iteracji pętli dodajemy jakąś liczbę całkowitą (9, 41, 12 itd.) do sumy bieżącej. Jeśli pomyślisz o zmiennej `count`, to zawiera ona "bieżącą sumę wartości napotkanych do tej pory". Tak więc przed rozpoczęciem pętli zmienna `total` jest zerem, ponieważ nie widzieliśmy jeszcze żadnych wartości; w trakcie działania pętli zmienna `total` jest sumą bieżącą, a na końcu pętli `total` jest sumą wszystkich wartości z listy.

Podczas wykonywania pętli `total` kumuluje sumę elementów; używana w ten sposób zmienna jest czasami nazywana *akumulatorem*.

W rzeczywistości ani pętla zliczająca, ani sumująca nie są szczególnie przydatne w praktyce, ponieważ istnieją wbudowane funkcje `len()` i `sum()`, które obliczają odpowiednio liczbę elementów na liście i sumę elementów na liście.

5.6.2. Pętle typu maksimum i minimum

Aby znaleźć największą wartość na liście, tworzymy następującą pętlę:

```
largest = None
print('Przed:', largest)
for itervar in [9, 41, 12, 3, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print('Pętla:', itervar, largest)
print('Po:', largest)
```

Gdy uruchomimy program, wyjście będzie następujące:

```
Przed: None
Pętla: 9 9
Pętla: 41 41
Pętla: 12 41
Pętla: 3 41
Pętla: 74 74
Pętla: 15 74
Po: 74
```

Najlepiej myśleć o zmiennej `largest` jako o "największej wartości, którą do tej pory widzieliśmy". Przed pętlą ustawiamy `largest` na stałą `None`, będącą specjalną stałą wartością, którą możemy przechowywać w zmiennej, tak by oznaczyć ją jako "pustą".

Przed rozpoczęciem pętli największą wartością, którą do tej pory widzieliśmy, jest `None`, ponieważ nie widzieliśmy jeszcze żadnych liczb. Podczas wykonywania pętli, jeśli `largest` to `None`, to jako największą wartość bierzemy pierwszą przetwarzaną liczbę. Możesz zobaczyć w pierwszej iteracji, że wartość `itervar` wynosi 9, ponieważ wartość `largest` to `None`, więc natychmiast ustawiamy `largest` na 9.

Po pierwszej iteracji `largest` nie jest już wartością `None`, więc druga część złożonego wyrażenia logicznego, która sprawdza `itervar > largest` wyzwala się tylko wtedy, gdy widzimy liczbę, która jest większa niż "największa do tej pory". Gdy widzimy nową "jeszcze większą" liczbę, bierzemy tę nową wartość i przypisujemy do `largest`. Możesz zobaczyć na wyjściu programu, że `largest` zwiększa się z 9 do 41, a potem do 74.

Na końcu pętli przeskanowaliśmy wszystkie liczby i zmienna `largest` zawiera teraz największą wartość z listy.

Kod do obliczenia najmniejszej liczby jest bardzo podobny, ale z jedną małą zmianą:

```
smallest = None
print('Przed:', smallest)
for itervar in [9, 41, 12, 3, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print('Pętla:', itervar, smallest)
print('Po:', smallest)
```

Podobnie jak wcześniej, `smallest` oznacza "najmniejszą do tej pory" przed, podczas i po wykonaniu pętli. Gdy pętla zostanie zakończona, `smallest` zawiera najmniejszą wartość na liście.

I tak samo, jak w przypadku zliczania i sumowania, wbudowane funkcje `max()` i `min()` czynią pisanie tego typu pętli zbędnym.

Poniżej znajduje się prosta wersja wbudowanej w Pythona funkcji `min()`:

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

W powyższej wersji usunęliśmy wszystkie polecenia `print()`, tak aby były równoważne z funkcją `min()`, która jest już wbudowana w Pythona.

5.7. Debugowanie

Gdy zaczniesz pisać większe programy, być może będziesz musiał więcej czasu poświęcać na debugowanie. Więcej kodu oznacza większe szanse na popełnienie błędu i więcej miejsc dla błędów by się ukryć.

Jednym ze sposobów na skrócenie czasu debugowania jest "debugowanie przez dzielenie na części". Na przykład, jeśli w Twoim programie jest 100 linii i sprawdzasz je po kolei, zajmie to 100 kroków.

Zamiast tego spróbuj podzielić problem na pół. Spójrz na środek programu lub w jego pobliżu, tak aby sprawdzić jakąś wartość pośrednią. Dodaj funkcję `print()` (lub coś innego, co ma sprawdzalny efekt) i uruchom program.

Jeśli sprawdzenie w środku programu jest nieprawidłowe, problem musi być w pierwszej połowie programu. Natomiast jeśli jest poprawne, to problem jest w drugiej połowie.

Za każdym razem, gdy wykonujesz taką kontrolę swojego programu, zmniejszasz o połowę liczbę linii, które musisz przeszukać. Po sześciu krokach (czyli znacznie mniej niż 100), sprowadza się to do jednej lub dwóch linijek kodu (przynajmniej w teorii).

W praktyce nie zawsze jest jasne, czym jest "środek programu" i nie zawsze można go sprawdzić. Nie ma sensu liczyć linii i znajdować dokładnego środka programu. Zamiast tego pomyśl o miejscach w programie, w których mogą wystąpić błędy, oraz o miejscach, w których łatwo jest to sprawdzić. Następnie wybierz miejsce, w którym Twoim zdaniem szanse na wystąpienie błędu przed lub po punkcie kontrolnym są mniej więcej takie same.

5.8. Słowniczek

akumulator Zmienna używana w pętli do sumowania lub akumulowania wyniku.
dekrementacja Aktualizacja, która zmniejsza wartość zmiennej (najczęściej o jeden).
inicjalizacja Przypisanie nadające zmiennej wartość początkową, która później zostanie zaktualizowana.
inkrementacja Aktualizacja, która zwiększa wartość zmiennej (najczęściej o jeden).
iteracja Powtórne wykonanie zbioru instrukcji przy pomocy funkcji, która sama się wywołuje, lub przy pomocy pętli.
licznik Zmienna używana w pętli do zliczania, ile razy coś się wydarzyło. Inicjalizujemy licznik jako zero, a następnie zwiększamy go za każdym razem, gdy chcemy coś "zliczyć".
pętla nieskończona Pętla, w której warunek kończący nie jest nigdy spełniony lub dla której nie ma warunku kończącego.

5.9. Ćwiczenia

Ćwiczenie 1. Napisz program, który odczytuje liczby tak długo, aż użytkownik wprowadzi "gotowe". Po wpisaniu "gotowe" wypisz sumę, ile wprowadzono liczb oraz średnią z tych liczb. Jeśli użytkownik wprowadzi coś innego niż liczba, to używając `try` i `except` wykryj jego błąd, wypisz komunikat o błędzie oraz przejdź do następnej liczby.

```
Wprowadź liczbę: 4
Wprowadź liczbę: 5
Wprowadź liczbę: złe dane
Nieprawidłowe wejście
Wprowadź liczbę: 7
Wprowadź liczbę: gotowe
16 3 5.333333333333333
```

Ćwiczenie 2. Napisz kolejny program, który będzie prosił o listę liczb tak jak wyżej, ale na końcu zamiast średniej wypisze zarówno największą, jak i najmniejszą wprowadzoną liczbę.