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 6

Napisy

6.1. Napis jest sekwencją

Napis jest *sekwencją* znaków. Dostęp do poszczególnych znaków możesz uzyskać za pomocą operatora w postaci nawiasów kwadratowych:

```
>>> fruit = 'banan'
>>> letter = fruit[1]
```

Druga instrukcja wyodrębnia ze zmiennej `fruit` znak z pozycji o indeksie 1 i przypisuje go do zmiennej `letter`.

Wyrażenie w nawiasach kwadratowych nazywane jest *indeksem*. Indeks wskazuje, który chcesz znak z zadanej sekwencji.

Wynik powyższego pobrania litery z napisu może być nieco zaskakujący:

```
>>> print(letter)
a
```

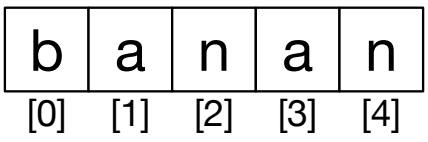
Dla większości ludzi pierwsza litera słowa "banan" to "b", a nie "a". Jednak w Pythonie indeks oznacza *przesunięcie od początku* napisu, a przesunięcie od pierwszej litery wynosi zero.

```
>>> letter = fruit[0]
>>> print(letter)
b
```

Tak więc "b" to 0. ("zerowa") litera słowa "banan", "a" to pierwsza litera, a "n" to druga litera.

Możesz użyć dowolnego wyrażenia, w tym zmiennych i operatorów, jako indeksu, ale wartość indeksu musi być liczbą całkowitą. W przeciwnym razie otrzymasz:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```



Rysunek 6.1. Indeksy w napisach

6.2. Uzyskanie długości napisu przy pomocy len()

len() jest funkcją wbudowaną, która zwraca liczbę znaków w danym napisie:

```
>>> fruit = 'banan'
>>> len(fruit)
5
```

Jeśli chciałbyś otrzymać ostatnią literę, to może Cię kusić, by spróbować czegoś takiego:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

Powodem błędu IndexError jest to, że w słowie "banan" nie ma litery z indeksem 5. Odkąd zaczęliśmy liczyć od zera, pięć liter jest oznaczonych liczbami od 0 do 4. Aby otrzymać ostatni znak, musisz odjąć 1 od length:

```
>>> last = fruit[length-1]
>>> print(last)
n
```

Opcjonalnie możesz użyć ujemnych indeksów, które liczą wstecz od końca napisu. Wyrażenie fruit[-1] daje ostatnią literę, fruit[-2] daje przedostatnią itd.

6.3. Przejście przez napis przy użyciu pętli

Wiele obliczeń polega na przetwarzaniu tekstu znak po znaku. Często zaczynają się one od początku napisu, pobierają po kolei jeden znak, robią coś z nim i kontynuują tak aż do końca. Ten schemat przetwarzania jest nazywany *przejściem*. Jednym ze sposobów na napisanie przejścia jest użycie pętli while:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

Pętla ta przechodzi po napisie i wyświetla w wierszu pojedynczo każdą literę. Warunkiem pętli jest index < len(fruit), więc gdy index jest równy długości napisu, warunek jest fałszywy, a ciało pętli nie jest już wykonywane. Ostatnim dostępnym znakiem jest ten z indeksem len(fruit)-1, który jest ostatnim znakiem w napisie.

Ćwiczenie 1. Napisz pętlę while, która zaczyna się od ostatniego znaku w napisie i działa od końca, do pierwszego znaku, wypisując każdą literę w osobnej linii, w odwrotnej kolejności.

Innym sposobem na napisanie przejścia przez napis jest użycie pętli for:

```
for char in fruit:
    print(char)
```

Za każdym przejściem przez pętlę kolejny znak napisu jest przypisywany do zmiennej char. Pętla jest kontynuowana aż do momentu, gdy nie pozostaną już żadne znaki do przypisania.

6.4. Wycinki napisów

Fragment napisu nazywany jest *wycinkiem* lub *kawałkiem*. Wybranie wycinka jest podobne do wybrania pojedynczego znaku:

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

Operator dwukropka zwraca część napisu od "n-tego" do "m-tego" znaku, włączając w to pierwszy, ale wyłączając z niego ostatni.

Jeśli pominiesz pierwszy indeks (przed dwukropkiem), wycinek zaczyna się od początku napisu. Jeśli pominiesz drugi indeks, wycinek jest pobierany aż do końca napisu:

```
>>> fruit = 'banan'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'an'
```

Jeśli pierwszy indeks jest większy lub równy drugiemu, wynikiem jest *pusty napis*, reprezentowany przez dwa apostrofy:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

Pusty napis nie zawiera żadnych znaków i ma długość 0 (ale poza tym jest tym samym, co każdy inny napis).

Ćwiczenie 2. Zakładając, że `fruit` jest napisem, co oznacza `fruit[:]`?

6.5. Napisy są niezmienne

Jeśli będziemy mieli potrzebę zmiany jakiegoś znaku w napisie, to może być kuszące skorzystanie z operatora przypisania. Na przykład:

```
>>> greeting = 'Witaj świecie!'
>>> greeting[0] = 'V'
TypeError: 'str' object does not support item assignment
```

Obiekt ("object") w tym przypadku to napis, a element ("item") jest znakiem, który próbowałeś do niego przypisać. W tym momencie *obiekt* jest tym samym co wartość, ale definicję tę doszlifujemy później. *Element* jest jedną z wartości w sekwencji.

Powodem błędu jest to, że napisy są *niezmienne*, co oznacza, że nie można zmienić istniejącego już napisu. Jedyne co możesz zrobić, to stworzyć nowy napis, który jest jakąś wersją oryginału:

```
>>> greeting = 'Witaj świecie!'
>>> new_greeting = 'V' + greeting[1:]
>>> print(new_greeting)
Vitaj świecie!
```

Ten przykład konkatenuje (łączy) nową pierwszą literę z wycinkiem zmiennej `greeting`. Nie ma to żadnego wpływu na pierwotny napis.

6.6. Przechodzenie w pętli i zliczanie

Poniższy program zlicza, ile razy litera "a" pojawia się w danym napisie:

```
word = 'banan'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

Zmienna `count` jest inicjalizowana z wartością 0, a następnie zwiększana za każdym razem, gdy natrafimy na "a". Po wyjściu z pętli `count` zawiera wynik: łączną liczbę wystąpień litery "a".

Ćwiczenie 3. Hermetyzacja to styl programowania, w którym szczegóły danej implementacji są ukryte.[1] Dokonaj hermetyzacji powyższego kodu poprzez zamknięcie go w funkcji o nazwie `count()` i uogólnij ten kod tak, by przyjmował jako argumenty napis i literę.

6.7. Operator `in`

Słowo `in` jest operatorem logicznym, który bierze dwa napisy i zwraca `True`, jeśli pierwszy z nich pojawi się jako podciąg w drugim:

```
>>> 'a' in 'banan'
True
>>> 'pestka' in 'banan'
False
```

6.8. Porównywanie napisów

Operatory porównania działają również na napisach. Aby sprawdzić, czy dwa napisy są równe, możemy wykonać:

```
if word == 'banan':
    print('Wszystko okej, to banany.')
```

Pozostałe operacje porównania są przydatne podczas układania słów w porządku alfabetycznym:

```
if word < 'banan':
    print('Twój wyraz, ' + word + ', jest przed słowem banan.')
elif word > 'banan':
    print('Twój wyraz, ' + word + ', jest po słowie banan.')
else:
    print('Okej, to banan.')
```

Python nie radzi sobie z dużymi i małymi literami tak samo jak ludzie. Wszystkie duże litery mają pierwszeństwo przed wszystkimi małymi:

```
Twój wyraz, Grejpfrut, jest przed słowem banan.
```

Powszechnym sposobem na rozwiązanie tego problemu jest konwersja napisów do standardowego formatu, czyli zawierającego wyłącznie małe litery, przed wykonaniem porównania. Pamiętaj o tym, na wypadek gdybyś musiał się bronić przed napastnikiem uzbrojonym w grejpfruta.

[1] W węższym znaczeniu hermetyzacja jest jednym z założeń programowania obiektowego, ale na razie nie wchodźmy za bardzo w szczegóły tego zagadnienia.

6.9. Metody obiektów będących napisami

Napisy są przykładem *obiektów* Pythona. Obiekt zawiera zarówno dane (czyli interesujący nas tekst), jak i *metody*, które w praktyce są funkcjami wbudowanymi w obiekt i dostępnymi dla każdej *instancji* obiektu.

Python posiada funkcję o nazwie `dir()`, która zawiera listę metod dostępnych dla danego obiektu. Funkcja `type()` pokazuje typ obiektu:

```
>>> stuff = 'Witaj świecie'
>>> type(stuff)
<class 'str'>
```

Natomiast funkcja `dir()` pokazuje dostępne dla niego metody. Możesz też użyć `help()`, by uzyskać prostą dokumentację jakiejś metody:

```
>>> dir(stuff)
['capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> str

    Return a capitalized version of S, i.e. make the first character
    have upper case and the rest lower case.
>>>
```

W przypadku metod związanych z napisami, dobrym źródłem dokumentacji jest również https://docs.python.org/library/stdtypes.html#string-methods.

Wywołanie *metody* jest podobne do wywołania funkcji (przyjmuje ona argumenty i zwraca wartość), ale składnia jest nieco inna. Metodę wywołujemy poprzez dołączenie nazwy metody do nazwy zmiennej, używając przy tym kropki jako separatora.

Na przykład metoda `upper()` przyjmuje napis i zwraca nowy napis, w którym wszystkie litery zostały zamienione na duże litery.

Zamiast składni funkcji `upper(word)` używa ona składni metody `word.upper()`.

```
>>> word = 'banan'
>>> new_word = word.upper()
>>> print(new_word)
BANAN
```

Taka forma notacji kropkowej określa nazwę metody (`upper()`) oraz nazwę zmiennej z napisem, na której metoda ta ma zostać zastosowana (`word`). Puste nawiasy wskazują, że metoda ta nie przyjmuje żadnego argumentu.

W tym przypadku powiedzielibyśmy, że *wywołujemy metodę* `upper()` na `word`.

Na przykład dla napisów istnieje metoda o nazwie `find()`, która szuka pozycji jednego napisu w drugim:

```
>>> word = 'banan'
>>> index = word.find('a')
>>> print(index)
```

```
1
```

W powyższym przykładzie wywołujemy metodę `find()` na `word` i przekazujemy szukaną literę jako argument.

Metoda `find()` może znaleźć zarówno pojedyncze znaki, jak i dłuższe sekwencje znaków (podciągi):

```
>>> word.find('an')
1
```

Metoda ta jako drugi argument może przyjąć indeks, od którego powinien zacząć szukać:

```
>>> word.find('an', 2)
3
```

Jednym z częstych zadań jest usunięcie białych znaków (spacji, tabulatorów lub znaków nowej linii) z początku i końca napisu przy użyciu metody `strip()`:

```
>>> line = '  Proszę bardzo  '
>>> line.strip()
'Proszę bardzo'
```

Niektóre metody, takie jak `startswith()`, zwracają wartości logiczne.

```
>>> line = 'Miłego dnia'
>>> line.startswith('Miłego')
True
>>> line.startswith('m')
False
```

Możesz zauważyć, że `startswith()` wymaga dopasowania wielkości liter, więc czasami zanim zrobimy jakiekolwiek sprawdzenie, konwertujemy wszystkie znaki na małe litery przy użyciu metody `lower()`.

```
>>> line = 'Miłego dnia'
>>> line.startswith('m')
False
>>> line.lower()
'miłego dnia'
>>> line.lower().startswith('m')
True
```

W ostatnim przykładzie metoda `lower()` jest wywoływana, a następnie używamy `startswith()` by sprawdzić, czy wynikowy ciąg małych liter zaczyna się od litery "m". Tak długo jak pilnujemy właściwiej kolejności, w jednym wyrażeniu możemy wykonywać wiele wywołań metod.

Ćwiczenie 4. Dla tekstowego typu danych istnieje metoda o nazwie `count()`, która jest podobna do funkcji z poprzedniego ćwiczenia. Przeczytaj dokumentację tej metody pod adresem:

https://docs.python.org/library/stdtypes.html#string-methods

Napisz wywołanie metody, które zliczy, ile razy litera "a" występuje w słowie "banan".

6.10. Parsowanie napisów

Często chcemy zajrzeć do zmiennej przechowującej tekst po to, aby znaleźć w nim pewien podciąg. Na przykład, jeśli przedstawiono nam serię wierszy tekstowych sformatowanych w następujący sposób:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

i chcielibyśmy z każdej linii wyciągnąć tylko drugą połowę adresu e-mail (tj. `uct.ac.za`), to możemy to zrobić za pomocą metody `find()` i operacji wyodrębniania wycinków z napisów.

Najpierw znajdziemy w napisie pozycję znaku `@`. Następnie znajdziemy pozycję pierwszej spacji *po* znaku `@`. Na końcu użyjemy wycinków, by wyodrębnić część napisu, której szukamy.

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ', atpos)
>>> print(sppos)
31
>>> host = data[atpos+1:sppos]
>>> print(host)
uct.ac.za
>>>
```

Używamy tej wersji metody `find()`, która pozwala nam określić w napisie pozycję, od której metoda ta ma rozpocząć przeszukiwanie. Wyodrębniamy znaki od "znaku będącego zaraz za @ aż do *ale nie włączając w to* znaku spacji".

Dokumentacja dotycząca metody `find()` jest dostępna na stronie https://docs.python.org/library/stdtypes.html#string-methods.

6.11. Operator formatowania

Operator formatowania `%` pozwala nam na konstruowanie napisów, zastępując części napisów danymi przechowywanymi w zmiennych. W przypadku zastosowania go do liczb całkowitych `%` jest operatorem modulo. Ale gdy pierwszym operandem jest napis, `%` jest operatorem formatowania.

Pierwszy operand jest *napisem formatującym*, który zawiera jedną lub więcej *sekwencji formatujących*, które z kolei określają, w jaki sposób jest formatowany drugi operand. Wynikiem tej operacji jest nowy napis.

Na przykład sekwencja formatująca `%d` oznacza, że drugi operand powinien być sformatowany jako liczba całkowita ("d" pochodzi od angielskiego słowa *decimal*, czyli "dziesiętny"):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

Wynikiem jest napis `'42'`, którego nie należy mylić z liczbą całkowitą o wartości `42`.

Sekwencja formatowania może pojawić się w dowolnym miejscu napisu, więc możesz osadzić jakąś wartość np. w środku zdania:

```
>>> camels = 42
>>> 'Zauważyłem %d wielbłądy.' % camels
'Zauważyłem 42 wielbłądy.'
```

Jeśli w napisie znajduje się więcej niż jedna sekwencja formatująca, to drugim argumentem musi być krotka[2]. Każda sekwencja formatująca jest dopasowywana do kolejnego elementu krotki.

Poniższy przykład używa `%d` do formatowania liczby całkowitej, `%g` do formatowania liczby zmiennoprzecinkowej (nie pytaj, dlaczego), a `%s` do formatowania napisu[3]:

```
>>> 'Przez %d lata widziałem %g %s.' % (3, 0.1, 'wielbłąda')
'Przez 3 lata widziałem 0.1 wielbłąda.'
```

Liczba elementów w krotce musi być zgodna z liczbą sekwencji formatujących w napisie. Typy elementów również muszą być zgodne z sekwencjami formatującymi:

[2] Krotka to sekwencja wartości oddzielonych przecinkami wewnątrz pary nawiasów okrągłych. Omówimy krotki w rozdziale 10.
[3] "s" od angielskiego "string", czyli napis.

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dolary'
TypeError: %d format: a number is required, not str
```

W pierwszym przykładzie nie ma wystarczającej liczby elementów, w drugim – element ma niewłaściwy typ.

Operator formatowania niesie ze sobą wiele możliwości, ale czasami może być trudny w użyciu. Możesz przeczytać więcej na ten temat na stronie z dokumentacją:

https://docs.python.org/library/stdtypes.html#printf-style-string-formatting

6.12. Debugowanie

Umiejętność, którą powinieneś rozwijać w trakcie programowania, to zawsze zadawanie sobie pytania: "Co tu się może nie udać?" lub alternatywnie: "Jaką szaloną rzecz może zrobić nasz użytkownik, aby wysypać nasz (pozornie) doskonały program?".

Na przykład spójrz na program, którego użyliśmy do zademonstrowania pętli `while` w rozdziale o iteracjach:

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

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

Spójrz, co się stanie, gdy użytkownik wprowadzi pusty wiersz:

```
> no hejka
no hejka
> # nie wyświetlaj tego
> wyświetl to!
wyświetl to!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#':
IndexError: string index out of range
```

Kod działa dobrze, dopóki nie zostanie mu podsunięty pusty wiersz. Nie można wtedy odnaleźć znaku zerowego, więc otrzymujemy komunikat mechanizmu *traceback*. Są na to dwa rozwiązania, które sprawiają, że trzeci wiersz jest "bezpieczny", nawet jeśli jest on pusty.

Jedną z możliwości jest po prostu użycie metody `startswith()`, która zwraca `False`, jeśli napis jest pusty.

```
if line.startswith('#'):
```

Innym sposobem jest bezpieczne napisanie wyrażenia `if` przy użyciu *wzorca strażnika* i upewnienie się, że drugie wyrażenie logiczne jest ewaluowane tylko wtedy, gdy w napisie jest co najmniej jeden znak.

```
if len(line) > 0 and line[0] == '#':
```

6.13. Słowniczek

element Jedna z wartości w sekwencji.
indeks Wartość całkowita używana do zaznaczenia elementu w sekwencji, np. znaku w napisie.
metoda Funkcja, która jest związana z obiektem i jest wywoływana przy użyciu notacji kropkowej.
napis formatujący Napis używany z operatorem formatowania, zawierający sekwencje formatowania.
niezmienność Własność sekwencji, której elementy nie mogą być przypisywane.
obiekt Coś, do czego może odnosić się zmienna. Na razie możesz używać zamiennie słowa "obiekt" i "wartość".
operator formatowania Operator %, który bierze napis formatujący oraz krotkę i generuje nowy napis, zawierający elementy krotki sformatowane zgodnie z opisem podanym w napisie formatującym.
przejście Iteracja po elementach sekwencji, wykonująca jakąś podobną operację na każdym z nich.
pusty napis Napis żadnych znaków i o długości 0, reprezentowany przez dwa apostrofy.
sekwencja Uporządkowany zbiór, czyli zbiór wartości, gdzie każda wartość jest identyfikowana przez indeks będący liczbą całkowitą.
sekwencja formatująca Sekwencja znaków w napisie formatującym, np. `%d`, która określa, jak dana wartość powinna być sformatowana.
wycinek Część napisu określona przez zakres indeksów.
wywołanie metody Instrukcja, która wykonuje funkcję ściśle związaną z danym obiektem i jego instancjami.

6.14. Ćwiczenia

Ćwiczenie 5. Mamy następujący kod Pythona, który przechowuje napis w zmiennej `text`:

```
text = 'X-DSPAM-Confidence: 0.8475'
```

Użyj `find()` i wycinków napisów, tak aby wyodrębnić część tekstu po znaku dwukropka, a następnie użyj funkcji `float()`, żeby przekształcić wyodrębniony fragment na liczbę zmiennoprzecinkową.

Ćwiczenie 6. Przeczytaj dokumentację metod związanych z tekstowym typem danych na stronie https://docs.python.org/library/stdtypes.html#string-methods. Możesz poeksperymentować z niektórymi z nich, aby upewnić się, że rozumiesz, jak one działają. Szczególnie przydatne są `strip()` i `replace()`.

Dokumentacja używa składni, która może być nieco myląca. Na przykład w `find(sub[, start[, end]])` nawiasy kwadratowe wskazują opcjonalne argumenty. Tak więc `sub` jest wymagany, ale `start` jest opcjonalny, a jeśli dodasz `start`, to `end` jest opcjonalny.