# Python dla wszystkich

Eksploracja danych z Python 3

Dr Charles R. Severance

## Podziękowania autorskie

Wsparcie edytorskie: Elliott Hauser, Sue Blumenberg
Projekt okładki: Aimee Andrion

## Historia wydań

- 2020-09-27 Pierwsza wersja robocza polskiej wersji językowej
- 2016-07-05 Pierwsza kompletna wersja Pythona 3.0
- 2015-12-20 Pierwsza wstępna konwersja do Pythona 3.0

## Prawa autorskie

# Przedmowa

## Remiksowanie otwartej książki

To całkiem naturalne, że naukowcy, którym ciągle się mówi "publikuj albo giń", chcą zawsze tworzyć od podstaw coś, co będzie ich własnym świeżym dziełem. Poniższa książka jest eksperymentem, który nie zaczyna się od zera, ale zamiast tego "remiksuje" książkę zatytułowaną *Think Python: How to Think Like a Computer Scientist*[1] autorstwa Allena B. Downey'a, Jeffa Elknera i innych.

W grudniu 2009 roku już piąty semestr z rzędu przygotowywałem się na Uniwersytecie Michigan do nauczania *SI502 - Networked Programming* i zdecydowałem, że nadszedł czas, by napisać podręcznik do Pythona, który będzie skupiał się na eksplorowaniu danych, a nie na rozumieniu algorytmów i abstrakcji. Moim celem w kursie SI502 jest nauczenie ludzi na całe życie umiejętności obsługi danych przy użyciu Pythona. Niewielu z moich studentów planowało zostać profesjonalnymi programistami. Zamiast tego planowali być bibliotekarzami, menadżerami, prawnikami, biologami, ekonomistami itp., którzy chcieliby umiejętnie wykorzystywać technologię w wybranej przez siebie dziedzinie.

Nigdy nie wydawało mi się, że znajdę idealną, zorientowaną na dane, książkę Pythona do mojego kursu, więc postanowiłem właśnie taką napisać. Na szczęście na trzy tygodnie przed tym, jak miałem zamiar zacząć moją nową książkę od zera w czasie przerwy wakacyjnej, dr Atul Prakash podczas spotkania na wydziale pokazał mi książkę *Think Python*, której używał do prowadzenia swojego kursu Pythona w tamtym semestrze. Jest to dobrze napisany tekst z dziedziny informatyki, skupiający się na krótkich, bezpośrednich wyjaśnieniach i łatwości uczenia się.

Ogólna struktura książki została zmieniona, aby jak najszybciej przejść do rozwiązywania problemów związanych z analizą danych i mieć od samego początku szereg bieżących przykładów i ćwiczeń dotyczących analizy.

Rozdziały 2-10 są podobne do tych z książki *Think Python*, ale nastąpiły w nich poważne zmiany. Przykłady i ćwiczenia zorientowane na liczby zostały zastąpione ćwiczeniami zorientowanymi na dane. Tematy są przedstawiane w kolejności potrzebnej do budowania coraz bardziej wyrafinowanych rozwiązań w zakresie analizy danych. Niektóre tematy, takie jak `try` i `except`, są rozwijane i prezentowane jako część rozdziału o instrukcjach warunkowych. Funkcje są traktowane bardzo oględnie dopóki będą potrzebne do obsługi bardziej złożonych programów, a nie są wprowadzane w nauce początkowej jako abstrakcja. Prawie wszystkie funkcje zdefiniowane przez użytkownika zostały usunięte z przykładowego kodu i ćwiczeń za wyjątkiem rozdziału 4. Słowo "rekurencja"[2] w ogóle nie pojawia się w tej książce.

W rozdziałach 1 oraz 11-16, cały materiał jest zupełnie nowy, skupiając się na rzeczywistych zastosowaniach i prostych przykładach Pythona w analizie danych, włączając w to wyrażenia regularne do wyszukiwania i parsowania tekstu, automatyzację zadań na komputerze, pobieranie danych w sieci, przeszukiwanie stron internetowych, programowanie obiektowe, korzystanie z usług sieciowych, parsowanie danych XML i JSON, tworzenie i korzystanie z baz danych przy użyciu SQL oraz wizualizację danych.

[1] Polskie wydanie tej książki to *Myśl w języku Python! Nauka programowania.*
[2] Z wyjątkiem, oczywiście, tej linii.

Ostatecznym celem wszystkich tych zmian jest przejście od informatyki w znaczeniu nauki ścisłej (ang. Computer Science) do informatyki w znaczeniu dyscypliny badającej struktury, zachowania i interakcje rzeczywistych i sztucznych systemów, które przechowują, przetwarzają i przekazują informacje (ang. Informatics). Dodatkowym celem jest włączenie do pierwszych zajęć z technologii informacyjnych tych tematów, które mogą być użyteczne nawet jeśli ktoś zdecyduje, że nie zostanie profesjonalnym programistą.

Studenci, których ta książka zainteresuje i będą chcieli dalej zgłębiać poruszaną tematykę, powinni zajrzeć do książki Allena B. Downey'a *Think Python*. Ponieważ te dwie książki w dużym stopniu nakładają się na siebie, studenci szybko nabędą umiejętności w dodatkowych technicznych aspektach programowania i myślenia algorytmicznego, które zostały omówione w książce *Think Python*. A biorąc pod uwagę, że książki mają podobny styl narracji, czytelnicy powinni być w stanie szybko przejść przez *Think Python* przy minimalnym wysiłku.

Jako osoba posiadająca prawa autorskie do *Think Python*, Allen udzielił mi zgody na zmianę licencji na materiał pozostający w tej książce z licencji GNU Free Documentation License na nowszą licencję Creative Commons Uznanie autorstwa-Na tych samych warunkach. Jest to następstwem ogólnego przesunięcia licencji otwartej dokumentacji z GFDL na CC-BY-SA (np. Wikipedia). Korzystanie z licencji CC-BY-SA podtrzymuje silną tradycję książki w zakresie copyleft, jednocześnie jeszcze bardziej ułatwiając nowym autorom ponowne wykorzystanie tego materiału według własnego uznania.

Uważam, że ta książka jest przykładem na to dlaczego otwarte materiały są tak ważne dla przyszłości edukacji i chcę podziękować Allenowi B. Downeyowi i Cambridge University Press za ich przyszłościową decyzję o udostępnieniu książki w ramach otwartych praw autorskich. Mam nadzieję, że są zadowoleni z wyników moich wysiłków i mam nadzieję, że Ty, czytelniku, jesteś zadowolony z *naszych* zbiorowych wysiłków.

Pragnę podziękować Allenowi B. Downey'owi i Lauren Cowles za pomoc, cierpliwość i wskazówki dotyczące postępowania i rozwiązywania problemów związanych z prawami autorskimi do tej książki.

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
9 września 2013 r.

Charles Severance zajmuje stanowisko Clinical Associate Professor w School of Information na Uniwersytecie Michigan.

## Informacja dotycząca polskiego wydania

Książka "Python dla wszystkich", w odniesieniu do swojego oryginału "Python for Everybody", jest nie tylko jej tłumaczeniem, ale posiada również szereg poprawek, rozwinięć i nowych materiałów.

W rozdziale 7 dodano sekcję o przetwarzaniu plików zawierających polski tekst. Rozdziały 13 i 16 korzystają z usługi Nominatim dostarczanej przez OpenStreetMap zamiast z płatnego GoogleAPI. Indeks został poprawiony, usunięto zbędne odniesienia oraz dodano nowe w celu łatwiejszego wyszukiwania treści. W tekście dodano

przypisy w celu rozjaśnienia polskiej terminologii. W kodach programów nie tłumaczono nazw zmiennych - jest to pewna ogólna konwencja programistyczna, ktora powinna również pomóc w sprawniejszym operowaniu pojęciami programistycznymi np. podczas wyszukiwania w internecie treści na zadany problem.

Na stronie internetowej, będącej uzupełnieniem tej książki, rozwinięto sekcję dotyczącą korzystania z Pythona pod Windowsem przy użyciu PowerShella.

# Spis treści

# Rozdział 1

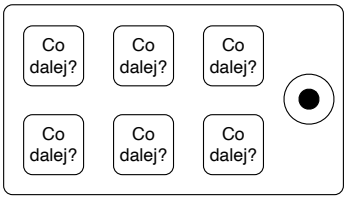
# Dlaczego powinieneś nauczyć się pisać programy?

Pisanie programów (lub programowanie) jest bardzo twórczą i satysfakcjonującą aktywnością. Możesz pisać programy z wielu powodów, od zarabiania na życie, przez rozwiązywanie trudnych problemów analizy danych, po zabawę i pomaganie komuś w rozwiązaniu problemu. Poniższa książka zakłada, że *każdy* powienien wiedzieć jak programować oraz że gdy już dowiesz się jak programować, to zorientujesz się co chcesz zrobić ze swoimi nowo odkrytymi umiejętnościami.

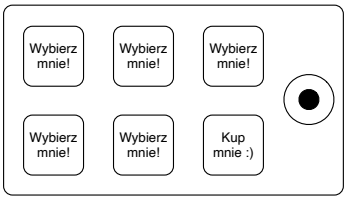
W codziennym życiu jesteśmy otoczeniu przez komputery, począwszy od laptopów, po smartfony. Możemy myśleć o tych komputerach jako o naszych "osobistych asystentach", którzy w naszym imieniu mogą zająć się wieloma rzeczami. Sprzęt we współczesnych komputerach jest zasadniczo zbudowany tak, aby nieustannie zadawać nam pytanie "Co mam teraz zrobić?".

Programiści dodają do sprzętu system operacyjny oraz zbiór aplikacji, dzięki czemu w ten sposób otrzymujemy osobistego asystenta cyfrowego, który okazuje się całkiem pomocny i zdolny do wsparcia nas w wielu różnych sprawach.

Nasze komputery są szybkie i mają ogromne zasoby pamięci – ten fakt mógłby być dla nas bardzo pomocny gdybyśmy tylko znali język do porozumiewania się i wyjaśniania komputerowi co chcemy aby dla nas "teraz zrobił". Gdybyśmy znali taki język, to moglibyśmy powiedzieć komputerowi by wykonał za nas pewne powtarzające się czynności. Co ciekawe, to, co komputery potrafią najlepiej, to często rzeczy, które my, ludzie, uważamy za nudne i otępiające.




Rysunek 1.1: Osobisty asystent cyfrowy




Rysunek 1.2: Programiści mówią do Ciebie

Na przykład, spójrz na pierwsze trzy akapity tego rozdziału i powiedz mi które słowo było najczęście użyte oraz ile razy to słowo zostało użyte. Podczas gdy byłeś w stanie przeczytać i zrozumieć słowa w ciągu kilku sekund, zliczanie ich jest niemalże bolesne, ponieważ nie jest to tego rodzaju problem, dla którego zostały zaprojektowane ludzkie umysły. W przypadku komputera jest na odwrót – czytanie i rozumienie tekstu z kartki papieru jest trudne do wykonania, ale zliczenie słów i wskazanie Tobie ile razy zostało użyte najczęściej występujące słowo jest bardzo łatwe:

```
python words.py
Podaj nazwę pliku: words.txt
w 5
```

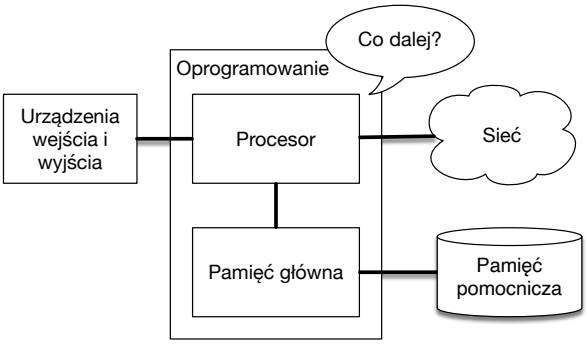
Nasz "osobisty asystent analizy informacji" szybko powiedział nam, że słowo "w" został użyty pięć razy w pierwszych trzech akapitach tego rozdziału.

Właśnie ten fakt, że komputery są dobre w rzeczach, w których ludzie dobrzy nie są, jest powodem, dla którego musisz nauczyć się mówić "językiem komputerowym". Gdy nauczysz się tego nowego języka, możesz przekazywać nieciekawe zadania swojemu partnerowi (komputerowi), pozostawiając sobie więcej czasu na robienie rzeczy, do których jesteś wyjątkowo przystosowany. Wnosisz kreatywność, intuicję i pomysłowość do tego duetu.

## 1.1. Kreatywność i motywacja

Chociaż ta książka nie jest przeznaczona dla profesjonalnych programistów, profesjonalne programowanie może być bardzo satysfakcjonującą pracą, zarówno po kątem finansowym, jak i osobistym. Tworzenie przydatnych, eleganckich i sprytnych programów, z których mogą korzystać inni, to bardzo kreatywne zajęcie. Twój komputer lub smartfon zwykle zawiera wiele różnych programów od wielu różnych grup programistów, które konkurują o Twoją uwagę i zainteresowanie. Robią wszystko, co w ich mocy, aby spełnić Twoje potrzeby i zapewnić podczas ich używania najlepsze *user experience* (z ang. *doświadczenie użytkownika*). W niektórych sytuacjach, gdy wybierasz oprogramowanie, programiści są bezpośrednio wynagradzani z powodu twojego wyboru.

Jeśli myślimy o programach jako o twórczym wyniku pracy grup programistów, to być może poniższy rysunek przedstawia bardziej racjonalną wersję naszego smartfona:




Rysunek 1.3: Architektura sprzętowa

Na razie naszą główną motywacją nie jest zarabianie pieniędzy ani zadowolenie użytkowników końcowych, ale raczej bardziej produktywne przetwarzanie danych i informacji, które napotkamy w naszym życiu. Na początku będziesz zarówno programistą, jak i końcowym użytkownikiem swoich programów. Gdy zdobędziesz umiejętności programisty, a programowanie stanie się dla ciebie bardziej twórcze, twoje myśli mogą skierować się w stronę tworzenia programów dla innych osób.

## 1.2. Architektura sprzętu kommputerowego

Zanim zaczniemy uczyć się języka, którym będziemy wydawać polecenia komputerom aby tworzyły dla nas oprogramowanie, musimy doszkolić się trochę o tym jak budowane są komputery. Gdybyś rozebrał komputer lub smartfon i zajrzał głęboko mu się przyjał, to znalazłbyś następujące części:

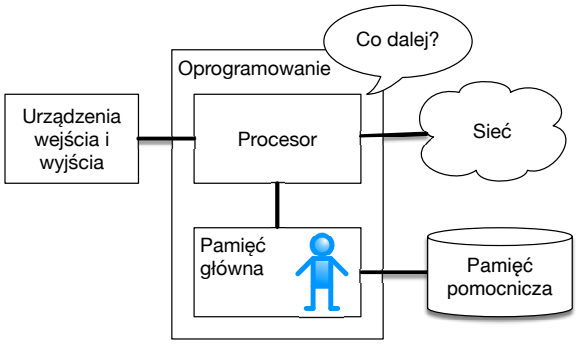
Ogólne definicje tych części są następujące:

- *Procesor* (ang. *central processing unit*, CPU) to część komputera, która została zbudowana tak, aby dosłownie mieć obsesję na punkcie pytania "Co dalej?". Jeśli komputer ma moc 3,0 gigaherców, to oznacza to, że procesor zapyta "Co dalej?" trzy miliardy razy na sekundę. Będziesz musiał nauczyć się w jaki sposób szybko porozumiewać się, tak by nadążać za procesorem.

- *Pamięć główna* jest używana do przechowywania informacji, których procesor potrzebuje w pośpiechu. Pamięć główna jest prawie tak szybka jak procesor. Jednakże informacje przechowywane w pamięci głównej znikają po wyłączeniu komputera.[1]

- *Pamięć pomocnicza* jest również używana do przechowywania informacji, ale jest znacznie wolniejsza niż pamięć główna. Zaletą pamięci dodatkowej jest to, że może przechowywać informacje nawet wtedy, gdy komputer nie jest zasilany. Przykładami pamięci dodatkowej są dyski twarde lub pamięć flash (zwykle znajdująca się w pamięciach USB/pendrive i przenośnych odtwarzaczach muzycznych).

[1] Chodzi tutaj o pamięć o dostępie swobodnym, czyli RAM (z ang. *random-access memory*).




Rysunek 1.4: Gdzie jesteś?

- *Urządzenia wejścia i wyjścia* to po prostu nasz ekran, klawiatura, mysz, mikrofon, głośnik, panel dotykowy itp. Są to wszystkie sposoby interakcji z komputerem.
- Obecnie większość komputerów ma również *połączenie sieciowe*, które umożliwia pobieranie informacji przez sieć. Możemy myśleć o sieci jako o bardzo powolnym miejscu do przechowywania i pobierania danych, które nie zawsze są dostępne. W pewnym sensie sieć jest wolniejszą i czasami zawodną formą *pamięci pomocniczej*.

Chociaż większość szczegółów dotyczących działania tych komponentów najlepiej pozostawić konstruktorom komputerów, dobrze jest mieć pewną terminologię, tak abyśmy mogli rozmawiać o tych różnych częściach komputera podczas pisania naszych programów.

Twoim zadaniem jako programisty jest wykorzystywanie i zarządzanie każdym z tych zasobów w celu rozwiązania problemu i przeanalizowania danych uzyskanych podczas rozwiązynia tego probelmu. Jako programista będziesz głównie "rozmawiać" z procesorem i mówić mu, co ma robić dalej. Czasami powiesz procesorowi aby użył pamięci głównej, pamięci pomocniczej, sieci lub urządzeń wejścia/wyjścia.

Musisz być osobą, która odpowiada na pytanie procesora "Co dalej?". Jednakże byłoby bardzo niewygodne zmniejszenie Ciebie do 5 mm wysokości i włożenie do komputera tylko po to, abyś mógł wydawać polecenia trzy miliardy razy na sekundę. Zamiast tego musisz wcześniej zapisać swoje instrukcje. Zapisane instrukcje nazywamy *programem*, a czynność zapisywania tych instrukcji i doprowadzania ich do poprawnej formy nazywamy *programowaniem*.

## 1.3. Zrozumieć programowanie

W dalszej części książki postaramy się zmienić Cię w osobę biegłą w sztuce programowania. Na końcu zostaniesz *programistą* - być może nie profesjonalnym programistą, ale przynajmniej będziesz posiadał umiejętności spojrzenia na problem analizy danych/informacji oraz opracowania programu do rozwiązania takiego problemu.

W pewnym sensie, aby być programistą potrzebujesz dwóch umiejętności:

- Po pierwsze, musisz znać język programowania (Python) - musisz znać jego słownictwo i gramatykę. Musisz umieć poprawnie pisać słowa w tym nowym języku i umieć konstruować dobrze sformułowane "zdania".
- Po drugie, musisz umieć "opowiadać historię". Pisząc opowiadanie, łączysz słowa i zdania, tak aby przekazać czytelnikowi jakąś ideę lub myśl. Konstruowanie historii wymaga pewnej sztuki, ale umiejętność pisania historii poprawia się poprzez pisanie i uzyskiwanie informacji zwrotnych. W programowaniu nasz program jest "historią", a problem, który próbujesz rozwiązać, to "idea/myśl".

Gdy nauczysz się jednego języka programowania (w tym przypadku Pythona), znacznie łatwiej będzie Ci nauczyć się drugiego języka programowania, takiego jak JavaScript lub C++. Nowy język programowania ma bardzo różne słownictwo i gramatykę, ale umiejętności rozwiązywania problemów będą takie same we wszystkich językach programowania.

Dość szybko nauczysz się "słownictwa" i "zdań" w Pythonie. Napisanie spójnego programu rozwiązującego zupełnie nowy problem zajmie trochę więcej czasu. Uczymy programowania podobnie jak pisania. Zaczynamy czytać i objaśniać programy, potem piszemy proste programy, a potem z czasem piszemy coraz bardziej złożone programy. W pewnym momencie "znajdziesz swoją muzę", sam zobaczysz pewne wzorce i w bardziej naturalny sposób zauważysz w jaki sposób podchodzić do danego problemu i jak napisać program, który go rozwiązuje. A gdy już do tego dojdziesz, programowanie stanie się bardzo przyjemną i twórczą czynnością.

Zaczniemy od słownictwa i struktury programów w języku Python. Bądź cierpliwy, ponieważ proste przykłady przypomną ci moment w życiu gdy pierwszy raz zacząłeś czytać.

## 1.4. Słowa i zdania

W przeciwieństwie do ludzkich języków, słownictwo Pythona jest w rzeczywistości dość skromne. To "słownictwo" nazywamy "słowami zastrzeżonymi". Są to słowa, które mają dla Pythona szczególne znaczenie. Gdy Python widzi te słowa w programie napisanym w języku Python, to mają one jedno i tylko jedno znaczenie dla Pythona. Później, podczas pisania programów, stworzysz własne słowa, które dla ciebie mają jakieś znaczenie - będą to *zmienne*. Będziesz mieć dużą swobodę w wyborze nazw dla swoich zmiennych, ale nie możesz używać żadnych zastrzeżonych słów Pythona jako nazwy zmiennej.

Gdy trenujemy psa, używamy specjalnych słów, takich jak "siad", "zostań" i "aport". Kiedy rozmawiasz z psem i nie używasz żadnych zastrzeżonych słów, to po prostu patrzy na ciebie z pytającym wyrazem twarzy, dopóki nie powiesz zastrzeżonego słowa. Na przykład, jeśli powiesz: "chciałbym aby więcej ludzi chodziło na spacer po to by poprawić ich ogólny stan zdrowia", większość psów prawdopodobnie usłyszy: "bla bla bla *spacer* bla bla bla bla". Dzieje się tak, ponieważ "spacer" jest zarezerwowanym słowem w języku komunikacji z psami. Natomiast wiele wskazuje na to, że język komunikacji między ludźmi a kotami nie ma zastrzeżonych słów[2].

[2] https://xkcd.com/231/

Lista zastrzeżonych słów języka, w którym ludzie komunikują się z Pythonem, zawiera następujące wyrazy:

```
and       del       global      not       with
as        elif      if          or        yield
assert    else      import      pass
break     except    in          raise
class     finally   is          return
continue  for       lambda      try
def       from      nonlocal    while
```

To tyle, a dodatkowo w przeciwieństwie do psa, Python jest już w pełni wyszkolony. Kiedy powiesz "try", Python wykona polecenie "try" bezbłędnie za każdym razem gdy je wypowiesz.

W odpowiednim czasie poznamy powyższe zastrzeżone słówa oraz to kiedy ich używać, ale na razie skupimy się na odpowiedniku słowa "daj głos" w Pythonie (w języku człowiek-pies). Dobrą rzeczą we wskazywaniu Pythonowi, że ma "dać głos", jest to, że możemy mu nawet powiedzieć co ma powiedzieć, przekazując mu wiadomość pomiędzy apostrofami:

```
print('Witaj świecie!')
```

I oto napisaliśmy w Pythonie nasze pierwsze poprawne składniowo zdanie. Nasze zdanie zaczyna się od funkcji *print*, po której następuje ujęty w apostrofy ciąg wybranego przez nas tekstu. Ciągi tekstu w instrukcjach *print* są ujęte w apostrofy lub cudzysłowy. Apostrofy i cudzysłowy robią to samo; większość ludzi używa apostrofów z wyjątkiem przypadków, w których apostrof pojawia się w ciągu tekstu.

## 1.5. Rozmawianie z Pythonem

Teraz, gdy mamy już słowo i proste zdanie, które znamy w Pythonie, musimy wiedzieć w jaki sposób rozpocząć rozmowę z Pythonem, tak aby przetestować nasze nowe umiejętności językowe.

Zanim będziesz mógł rozmawiać z Pythonem, musisz najpierw zainstalować na swoim komputerze oprogramowanie Pythona oraz nauczyć się jak uruchomić Pythona na swoim komputerze. Jest to zbyt wiele szczegółów jak na ten rozdział, więc sugeruję zajrzeć na stronę www.py4e.com, gdzie umieściłem szczegółowe instrukcje i zrzuty ekranu dotyczące konfiguracji i uruchamiania Pythona w systemach komputerów Macintosh i Windows. W pewnym momencie znajdziesz się w terminalu lub oknie wiersza poleceń i wpiszesz *python*, a interpreter Pythona rozpocznie pracę w trybie interaktywnym i pojawi się mniej więcej w następujący sposób:

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:54:25)
[MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Znak zachęty >>> jest sposobem zadawania Tobie pytań przez interpreter języka Python "Co mam teraz zrobić?". Python jest teraz gotowy do rozmowy z tobą. Wszystko co musisz teraz wiedzieć, to jak mówić w języku Pythona.

Powiedzmy na przykład, że nie znałeś nawet najprostszych słów lub zdań w języku Pythona. Możesz użyć standardowego zwrotu, z którego korzystają astronauci gdy lądują na odległej planecie i próbują rozmawiać z jej mieszkańcami:

```
>>> Przybywam w pokoju, zabierz mnie proszę do swojego przywódcy
File "<stdin>", line 1
    Przybywam w pokoju, zabierz mnie proszę do swojego przywódcy
              ^
SyntaxError: invalid syntax
>>>
```

Nie idzie nam za dobrze. Jeśli nie wymyślisz czegoś szybko, mieszkańcy planety prawdopodobnie dźgną cię włóczniami, rzucą na rożen, upieczą na ogniu i zjedzą na obiad.

Na szczęście na czas podróży zabrałeś kopię tej książki, a następnie przewróciłeś kartki na tę stronę i spróbowałeś jeszcze raz:

```
>>> print('Witaj świecie!')
Witaj świecie!
```

Wygląda to o wiele lepiej, więc spróbuj przekazać trochę więcej komunikatów:

```
>>> print('Musisz być legendarnym bogiem, który pochodzi z nieba')
Musisz być legendarnym bogiem, który pochodzi z nieba
>>> print('Czekaliśmy na Ciebie od dawna')
Czekaliśmy na Ciebie od dawna
>>> print('Nasza legenda mówi, że z musztardą będziesz bardzo
↪  smaczny')
Nasza legenda mówi, że z musztardą będziesz bardzo smaczny
>>> print 'Będziemy ucztować wieczorem, o ile nie powiesz
  File "<stdin>", line 1
    print 'Będziemy ucztować wieczorem, o ile nie powiesz
                                                         ^
SyntaxError: Missing parentheses in call to 'print'
>>>
```

Przez jakiś czas rozmowa szła całkiem dobrze, ale potem popełniłeś malutki błąd używając języka Python, a Python ponownie pokazał swoje pazury.

W tym miejscu powinieneś również zdać sobie sprawę, że podczas gdy Python jest niesamowicie złożony i wydajny oraz bardzo wybredny pod względem składni używanej do komunikacji z nim, Python *nie* jest inteligentny. Tak naprawdę po prostu rozmawiasz ze sobą, ale używając odpowiedniej składni.

W pewnym sensie, kiedy używasz programu napisanego przez kogoś innego, to rozmowa odbywa się między tobą a innymi programistami, gdzie Python działa

jako pośrednik. Python to sposób, dzięki któremu twórcy programów mogą wyrazić jak ma przebiegać taka rozmowa. W kilku kolejnych rozdziałach będziesz jednym z tych programistów używających Pythona do rozmów z użytkownikami twojego programu.

Zanim opuścimy naszą pierwszą rozmowę z interpreterem Pythona, prawdopodobnie powinieneś znać właściwy sposób na "pożegnanie się" podczas interakcji z mieszkańcami planety Python:

```
>>> do-zobaczenia
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'do' is not defined
>>> if you don't mind, I need to leave
  File "<stdin>", line 1
    if you don't mind, I need to leave
            ^
SyntaxError: invalid syntax
>>> quit()
```

Możesz zauważyć, że błąd jest różny dla pierwszych dwóch niepoprawnych prób. Drugi błąd jest inny, ponieważ *if* jest słowem zastrzeżonym, a Python zobaczył to słowo i pomyślał, że próbujemy coś powiedzieć, ale składnia zdania była błędna.

Właściwym sposobem "pożegnania się" z Pythonem jest wpisanie *quit()* za interaktywnym znakiem zachęty >>>. Odgadnięcie tego pewnie zajęłoby ci trochę czasu, więc posiadanie książki pod ręką prawdopodobnie okaże się pomocne.

## 1.6. Terminologia: interpreter i kompilator

Python jest językiem *wysokiego poziomu*, który w założeniu ma być dla ludzi stosunkowo prosty do czytania i pisania oraz do odczytywania i przetwarzania przez komputery. Inne języki wysokiego poziomu to Java, C++, PHP, Ruby, Basic, Perl, JavaScript i wiele innych. Sprzęt wewnątrz procesora nie rozumie żadnego z tych języków wysokiego poziomu.

Procesor rozumie język, który nazywamy *językiem maszynowym*. Język maszynowy jest bardzo prosty i, szczerze mówiąc, bardzo męczący w pisaniu, ponieważ jest przedstawiony w postaci zer i jedynek:

```
00101000111010010010101000000001111
11100110000011101010010101101101
...
```

Język maszynowy wydaje się z pozoru dość prosty, biorąc pod uwagę, że istnieją tylko zera i jedynki, ale jego składnia jest jeszcze bardziej złożona i znacznie bardziej skomplikowana niż Python. Tak więc bardzo niewielu programistów kiedykolwiek pisze w języku maszynowym. Zamiast tego tworzymy różne translatory, tak aby umożliwić programistom pisanie w językach wysokiego poziomu takich jak Python

lub JavaScript, a te translatory konwertują programy na język maszynowy w celu ich wykonania przez procesor.

Ponieważ język maszynowy jest powiązany ze sprzętem komputerowym, język maszynowy nie jest *przenośny* w przypadku różnych typów sprzętu. Programy napisane w językach wysokiego poziomu można przenosić między różnymi komputerami, używając innego interpretera na nowej maszynie lub rekompilując kod w celu utworzenia wersji językowej programu dla nowej maszyny.

Translatory języków programowania dzielą się na dwie ogólne kategorie: (1) interpretery i (2) kompilatory.

*Interpreter* odczytuje kod źródłowy programu w postaci napisanej przez programistę, analizuje kod źródłowy i w locie interpretuje instrukcje. Python to interpreter i kiedy uruchamiamy Pythona interaktywnie, to możemy wpisać wiersz (zdanie) w języku Pythona, a Python natychmiast to przetworzy i jest gotowy do wpisania kolejnego wiersza w języku Pythona.

Niektóre wiersze w języku Pythona mówią Pythonowi, że chcesz aby zapamiętał on jakąś wartość na później. Musimy wybrać nazwę dla tej wartości do zapamiętania i możemy użyć tej symbolicznej nazwy do późniejszego pobrania tej wartości. Używamy terminu *zmienna* w odniesieniu do etykiet, których używamy odnosząc się do tych przechowywanych danych.

```
>>> x = 6
>>> print(x)
6
>>> y = x * 7
>>> print(y)
42
>>>
```

W tym przykładzie prosimy Pythona, aby zapamiętał wartość sześć i użył etykiety $x$ abyśmy mogli później pobrać tę wartość. Sprawdzamy za pomocą *print* czy Python faktycznie zapamiętał wartość. Następnie prosimy Pythona o pobranie $x$, pomnożenie przez siedem i umieszczenie nowo obliczonej wartości w $y$. Następnie prosimy Pythona o wyświetlenie aktualnej wartości znajdującej się w $y$.

Mimo że wpisujemy te polecenia w Pythonie po jednym wierszu na raz, Python traktuje je jako uporządkowaną sekwencję instrukcji, przez co późniejsze instrukcje mogą pobierać dane utworzone we wcześniejszych instrukcjach. Napisaliśmy nasz pierwszy prosty akapit z czterema zdaniami w logicznej i zrozumiałej kolejności.

Naturą *interpretera* jest możliwość prowadzenia interaktywnej rozmowy, tak jak pokazano powyżej. *Kompilator* musi otrzymać w pliku cały program, potem uruchamia proces tłumaczenia kodu źródłowego wysokiego poziomu na język maszynowy, a następnie kompilator umieszcza wynikowy język maszynowy w pliku w celu jego późniejszego wykonania.

Jeśli masz system Windows, często te wykonywalne programy maszynowe mają przyrostek ".exe" lub ".dll", które oznaczają odpowiednio "plik wykonywalny" i "bibliotekę łączoną dynamicznie". W systemach Linux i Macintosh nie ma przyrostka, który jednoznacznie określałby plik jako wykonywalny.

Gdybyś miał otworzyć plik wykonywalny w edytorze tekstu, to wyglądałby on jak pisany przez szaleńca i byłby nieczytelny:

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

Nie jest łatwo czytać ani pisać w języku maszynowym, więc dobrze, że mamy *interpretery* i *kompilatory*, które pozwalają nam pisać w językach wysokiego poziomu takich jak Python czy C.

W tym miejscu w naszej dyskusji o kompilatorach i interpreterach powinieneś trochę się zastanowić nad samym interpreterem Pythona. W jakim języku jest on napisany? Czy jest napisany w języku kompilowanym? Gdy wpiszemy "python", to co właściwie się dzieje?

Interpreter Pythona jest napisany w języku wysokiego poziomu o nazwie "C". Możesz spojrzeć na rzeczywisty kod źródłowy interpretera Pythona, przechodząc na stronę www.python.org i sprawdzając kod źródłowy. Tak więc Python sam w sobie jest programem i jest skompilowany do kodu maszynowego. Gdy zainstalowałeś język Python na swoim komputerze (lub zainstalował go sprzedawca), to skopiowałeś do systemu kopię programu Python przetłumaczonego do kodu maszynowego. W systemie Windows wykonywalny kod maszynowy samego Pythona prawdopodobnie znajduje się w pliku o nazwie takiej jak:

```
C:\Python35\python.exe
```

Jest to więcej niż faktycznie musisz wiedzieć by zostać programistą Pythona, ale czasami warto już na początku odpowiedzieć sobie na te dręczące pytania.

## 1.7. Pisanie programu

Wpisywanie poleceń do interpretera Pythona to świetny sposób na eksperymentowanie z funkcjami Pythona, ale nie jest zalecane do rozwiązywania bardziej złożonych problemów.

Gdy chcemy napisać program, to używamy edytora tekstu po to by zapisać instrukcje Pythona do pliku, który nazywa się *skryptem*. Zgodnie z przyjętą konwencją, skrypty w Pythonie mają nazwy kończące się na `.py`.

Aby uruchomić skrypt, musisz podać interpreterowi Pythona nazwę pliku. W oknie wiersza poleceń wpiszesz `python hello.py` w następujący sposób:

```
$ cat hello.py
print('Witaj świecie!')
$ python hello.py
Witaj świecie!
```

Symbol "$" to znak zachęty systemu operacyjnego, a "cat hello.py" wyświetla nam, że plik "hello.py" zawiera jednolinijkowy program w języku Python do wyświetlania ciągu znaków.

Uruchamiamy interpreter języka Python i mówimy mu aby wczytał kod źródłowy z pliku "hello.py" (zamiast interaktywnie pytać nas o kolejne wiersze kodu Pythona).

Zauważysz, że nie było potrzeby umieszczania *quit()* na końcu pliku programu Pythona. Gdy Python wczytuje kod źródłowy z pliku, to wie, że ma zakończyć pracę gdy osiągnie koniec pliku.

## 1.8. Czym jest program?

W najbardziej podstawowej formie definicja *programu* to sekwencja instrukcji Pythona, które została stworzona po to by coś zrobić. Nawet nasz prosty skrypt *hello.py* jest programem. Jest to program jednowierszowy i co prawda nie jest on szczególnie przydatny, ale w najściślejszej definicji jest to program w języku Python.

Najłatwiej jest zrozumieć czym jest program, myśląc o problemie, dla którego można ntworzyć program rozwiązujący ten problem, a następnie patrząc właśnie na ten program, który rozwiązuje ten problem.

Załóżmy, że prowadzisz badania w dziedzinie socjologii obliczeniowej bazując na postach umieszczanych na Facebooku i interesuje Cię najczęściej używane słowo w danej serii postów. Możesz wydrukować strumień postów na Facebooku i przejrzeć tekst w poszukiwaniu najpopularniejszego słowa, ale zajęłoby to dużo czasu i byłoby to bardzo podatne na pomyłki. Sprytniej byłoby napisać program w Pythonie, który szybko i dokładnie poradzi sobie z tym zadaniem, dzięki czemu będziesz mógł spędzić weekend na robieniu czegoś fajnego.

Dla przykładu, spójrz na poniższy tekst o klaunie i samochodzie. Spójrz na tekst i policz jakie słowo jest najczęściej używane oraz ile razy występuje.

```
the clown ran after the car and the car ran into the tent
and the tent fell down on the clown and the car
```

Następnie wyobraź sobie, że wykonujesz to zadanie patrząc na miliony linii tekstu. Szczerze mówiąc, szybsze byłoby nauczenie się języka Python i napisanie programu w języku Python do liczenia słów, niż ręczne analizowanie słów.

Jeszcze lepszą informacją jest to, że wymyśliłem już prosty program do znajdowania najczęściej występującego słowa w pliku tekstowym. Napisałem go, przetestowałem, a teraz oddaję Ci go do użytku, tak abyś mógł zaoszczędzić trochę czasu.

```
name = input('Podaj nazwę pliku: ')
handle = open(name, 'r')
counts = dict()

for line in handle:
    line = line.lower()
```

```
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in list(counts.items()):
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)

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

Nie musisz nawet znać języka Python, aby korzystać z tego programu. Będziesz musiał przejść przez rozdział 10 tej książki aby w pełni zrozumieć niesamowite techniki Pythona, które zostały użyte do stworzenia tego programu. Jesteś użytkownikiem końcowym, więc po prostu korzystasz z programu i zachwycasz się jego sprytem oraz oszczędnością ręcznego wysiłku. Po prostu wpisujesz powyższy kod do pliku o nazwie *words.py* i go uruchamiasz, lub pobierasz kod źródłowy ze strony https://pl.py4e.com/code3/ i uruchamiasz.

Jest to dobry przykład na to jak Python i jego język działają jako pośrednicy między tobą (użytkownikiem końcowym) a mną (programistą). Python to dla nas sposób na wymianę przydatnych sekwencji instrukcji (tj. programów) we wspólnym języku, z którego może korzystać każdy kto zainstaluje Pythona na swoim komputerze. Nikt z nas nie rozmawia *z Pythonem*, a zamiast tego komunikujemy się między sobą *poprzez* Pythona.

## 1.9. Elementy składowe programów

W następnych kilku rozdziałach dowiemy się więcej o słownictwie, strukturze zdań, akapitów i opowieści w Pythonie. Dowiemy się o zaawansowanych możliwościach Pythona oraz o tym jak łączyć te możliwości, tak aby tworzyć przydatne programy.

Istnieje kilka schematów pojęciowych niskiego poziomu, których używamy do tworzenia programów. Konstrukcje te nie są przeznaczone tylko dla programów pisanych w Pythonie - są one częścią każdego języka programowania, od języka maszynowego po języki wysokiego poziomu.

**wejście** Uzyskaj dane ze "świata zewnętrznego". Może to być odczyt danych z pliku lub nawet jakiegoś urządzenia takiego jak mikrofon lub GPS. W naszych początkowych programach dane wejściowe będą pochodzić od użytkownika wpisującego dane na klawiaturze.

**wyjście** Wyświetl wyniki programu na ekranie, zapisze je w pliku lub na przykład przekaż je do urządzenia takiego jak głośnik w celu odtworzenia muzyki lub wygłoszenia tekstu.

**wykonanie sekwencyjne** Wykonaj instrukcje jedna po drugiej w tej kolejności, w której występują w skrypcie.

**wykonanie warunkowe** Sprawdź czy występują określone warunki, a następnie wykonaj lub pomiń sekwencję instrukcji.

**wykonanie wielokrotne** Wykonaj kilka zestawów instrukcji wielokrotnie, zwykle z pewnymi zmianami.

**ponowne użycie** Napisz zestaw instrukcji raz i nadaj im nazwę, a następnie w miarę potrzeb użyj tych instrukcji w całym programie.

Brzmi to zbyt prosto by mogło być prawdziwe, no i oczywiście nigdy nie jest to takie proste. To tak jakby powiedzieć, że chodzenie to po prostu "stawianie jednej stopy przed drugą". "Sztuka" pisania programu polega na wielokrotnym komponowaniu i splataniu ze sobą powyższych podstawowych elementów, a wszystko to w celu wytworzenia czegoś, co jest przydatne dla użytkowników tego programu.

Powyższy program do liczenia słów bezpośrednio używa wszystkich tych wzorców z wyjątkiem jednego.

## 1.10. Co może pójść nie tak?

Jak widzieliśmy w naszych pierwszych rozmowach z Pythonem, musimy bardzo precyzyjnie komunikować się gdy piszemy kod w Pythonie. Najmniejsze odchylenie lub błąd spowoduje, że Python przestanie patrzeć na Twój program.

Początkujący programiści często przyjmują fakt, że Python nie pozostawia miejsca na błędy, jako dowód na to, że Python jest podły, nienawistny i okrutny. Podczas gdy Python wydaje się lubić wszystkie pozostałe osoby, Python zna programistów osobiście i żywi do nich urazę. Z powodu tej urazy Python bierze nasze doskonale napisane programy i odrzuca je jako "nieodpowiednie" tylko po to by nas drażnić.

```
>>> primt 'Witaj świecie!'
  File "<stdin>", line 1
    primt 'Witaj świecie!'
          ^
SyntaxError: invalid syntax
>>> primt ('Witaj świecie!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'primt' is not defined

>>> Nienawidzę Cię Pythonie!
  File "<stdin>", line 1
    Nienawidzę Cię Pythonie!
               ^
SyntaxError: invalid syntax
>>> gdybyś tylko stamtąd wyszedł, to dałbym ci nauczkę
  File "<stdin>", line 1
    gdybyś tylko stamtąd wyszedł, to dałbym ci nauczkę
          ^
SyntaxError: invalid syntax
>>>
```

Niewiele można zyskać kłócąc się z Pythonem. To tylko narzędzie. Nie ma emocji i jest szczęśliwy oraz gotowy by Ci służyć kiedy tylko tego potrzebujesz. Komunikaty o błędach brzmią ostro, ale są one po prostu wezwaniem pomocy przez Pythona. Sprawdził co wpisałeś i po prostu nie może tego zrozumieć.

Python jest bardziej podobny do psa, który kocha Cię bezwarunkowo, zna kilka kluczowych słów, które rozumie, patrzy na Ciebie ze słodkim wyrazem twarzy (>>>) i czeka aż powiesz coś, co zrozumie. Gdy Python mówi "SyntaxError: invalid syntax", to po prostu macha ogonem i mówi: "Wydawało mi się, że coś powiedziałeś, ale po prostu nie rozumiem co miałeś na myśli, ale proszę mów dalej (>>>)".

W miarę jak Twoje programy będą coraz bardziej wyrafinowane, napotkasz trzy ogólne typy błędów:

**Błędy składniowe** Błędy składniowe (ang. *syntax errors*) to pierwsze błędy, które popełnisz, i zarazem najłatwiejsze do naprawienia. Błąd składniowy oznacza, że naruszyłeś "gramatyczne" zasady Pythona. Python robi wszystko co w jego mocy by wskazać wiersz i znak, w którym zauważył, że jest zdezorientowany. Jedynym trudnym problemem błędów składniowych jest to, że czasami błąd, który wymaga naprawy, występuje w programie w rzeczywistości wcześniej niż w miescu, w którym Python *zauważył*, że został zdezorientowany. Tak więc linia i znak, które Python wskazuje w błędzie składniowym, mogą być tylko punktem wyjścia dla Twojego śledztwa.

**Błędy logiczne** Błąd logiczny (ang. *logic error*) występuje wtedy, gdy Twój program ma poprawną składnię, ale występuje błąd w kolejności instrukcji lub być może w tym jak instrukcje odnoszą się do siebie. Dobrym przykładem błędu logicznego może być: "napij się z butelki z wodą, włóż ją do plecaka, idź do biblioteki, a następnie zakręć butelkę".

**Błędy semantyczne** Błąd semantyczny (ang. *semantic error*) występuje gdy opis kroków, które należy wykonać, jest idealny pod względem składniowym i podany we właściwej kolejności, ale w programie jest po prostu pomyłka. Program jest całkowicie poprawny, ale nie robi tego co *zamierzałeś* by zrobił. Prostym przykładem może być podanie osobie wskazówek dojazdu do restauracji i powiedzenie: "... kiedy dojdziesz do skrzyżowania ze stacją benzynową, skręć w lewo i przejedź jeden kilometr, a restauracja to czerwony budynek po lewej stronie". Twój znajomy jest bardzo spóźniony i dzwoni do Ciebie, aby powiedzieć, że są na farmie i chodzą za stodołą, ale nie ma tu śladu restauracji. Następnie pytasz: "Skręciłeś w lewo czy w prawo na stacji benzynowej?". A oni mówią: "Postępowałem dokładnie zgodnie z twoimi wskazówkami, mam je zapisane, jest napisane, żeby skręcić w lewo i jechać jeden kilometr do stacji benzynowej". Następnie mówisz: "Bardzo mi przykro, bo chociaż moje instrukcje były poprawne składniowo, to niestety zawierały mały, ale niewykryty błąd semantyczny".

We wszystkich trzech typach błędów Python po prostu stara się zrobić dokładnie to, o co prosiłeś.

## 1.11. Debugowanie

Gdy Python zwraca błąd, a nawet gdy daje wynik inny niż zamierzony, to rozpoczyna poszukiwanie przyczyny błędu. Debugowanie to proces znajdowania w kodzie

przyczyny błędu. Podczas debugowania programu, a zwłaszcza podczas pracy nad poważnym błędem, należy spróbować czterech rzeczy:

**czytanie** Sprawdź swój kod, przeczytaj go i sprawdź czy zawiera to, co chciałeś przekazać.

**uruchamianie** Poeksperymentuj wprowadzając zmiany i uruchamiając różne wersje. Często gdy wyświetlasz w programie właściwą rzecz we właściwym miejscu, to problem staje się oczywisty, ale czasami trzeba poświęcić na to trochę czasu.

**rozmyślanie** Poświęć trochę czasu na przemyślenia! Jaki to rodzaj błędu: składniowy, wykonania, semantyczny? Jakie informacje można uzyskać z komunikatów o błędach lub z danych wyjściowych programu? Jaki rodzaj błędu może spowodować napotkany problem? Co ostatnio zmieniałeś zanim pojawił się problem?

**wycofanie się** W pewnym momencie najlepszą rzeczą, którą można zrobić, jest wycofanie się, tj. cofnięcie ostatnich zmian aż wrócisz do programu, który działa i który rozumiesz. Następnie możesz rozpocząć rekonstrukcję programu.

Początkujący programiści czasami grzęzną przy jednej z tych czynności i zapominają o innych. Znalezienie trudnego błędu wymaga czytania, uruchamiania, rozmyślania, a czasem wycofania się. Jeśli ugrzęźniesz w jednej z tych czynności, to wypróbuj inne. Każda czynność ma swój własny tryb wyszukiwania błędów.

Na przykład przeczytanie kodu może pomóc gdy problem jest błędem typograficznym, ale nie pomoże gdy problem dotyczy niezrozumienia pewnych konceptów i pojęć. Jeśli nie rozumiesz co robi Twój program, to możesz go przeczytać i ze 100 razy, a i tak nigdy nie zobaczysz błędu, ponieważ błąd tkwi w Twojej głowie.

Przeprowadzanie eksperymentów może pomóc, zwłaszcza jeśli przeprowadzasz małe, proste testy. Ale jeśli przeprowadzasz eksperymenty bez myślenia lub czytania kodu, możesz wpaść w schemat, który nazywam "programowaniem losowym", który jest procesem dokonywania przypadkowych zmian, dopóki program nie zrobi właściwej rzeczy. Nie trzeba dodawać, że programowanie losowe może zająć dużo czasu.

Musisz trochę pomyśleć. Debugowanie jest jak nauki eksperymentalne. Powinieneś mieć przynajmniej jedną hipotezę na temat tego, na czym polega problem. Jeśli są dwie lub więcej możliwości, to spróbuj pomyśleć o teście, który wyeliminowałby jedną z nich.

Przerwa pomaga w myśleniu. Tak samo działa mówienie. Jeśli wyjaśnisz problem komuś innemu (lub nawet sobie), to czasami znajdziesz odpowiedź, zanim skończysz zadawać pytanie.

Ale nawet najlepsze techniki debugowania zawiodą jeśli jest zbyt wiele błędów lub jeśli kod, który próbujesz naprawić, jest zbyt duży i skomplikowany. Czasami najlepszą opcją jest wycofanie się i uproszczanie programu tak długo, aż dojdziesz do czegoś, co działa i co rozumiesz.

Początkujący programiści często niechętnie się wycofują, ponieważ nie mogą znieść usunięcia linii kodu (nawet jeśli jest błędny). Jeśli sprawi to, że poczujesz się lepiej, to skopiuj kod programu do innego pliku zanim zaczniesz go demontować. Następnie możesz wklejać po trochu z powrotem kawałki kod.

## 1.12. Podróż do krainy rozwoju

W miarę postępów przez kolejne części książki nie bój się jeśli za pierwszym razem przedstawiane koncepcje nie będą wydawały się do siebie pasować. Kiedy uczyłeś się mówić, to przez kilka pierwszych lat nie stanowiło problemu, że po prostu wydawałeś słodkie odgłosy chichotania. I nie był to problem, że przejście od prostego słownictwa do prostych zdań zajęło ci sześć miesięcy, a przejście od zdań do akapitów zajęło ci jeszcze 5-6 lat, a kilka lat więcej by móc napisać interesujące, kompletne, samodzielne opowiadanie.

Chcemy abyś nauczył się języka Python znacznie szybciej, więc w następnych kilku rozdziałach uczymy wszystkiego po trochu w tym samym czasie. No ale to jest tak jakby uczyć się nowego języka, którego przyswojenie i zrozumienie wymaga czasu zanim posługiwanie się nim stanie się naturalne. Prowadzi to do pewnego zamieszania, gdy odkrywamy i powracamy do tematów, tak abyś mógł zobaczyć pełny obraz, podczas gdy jednocześnie definiujemy małe fragmenty składające się na ten pełny obraz. Podczas gdy książka jest napisana w sposób liniowy, a jeśli bierzesz udział w kursie, to będzie on przebiegał w sposób liniowy, nie wahaj się być bardzo nieliniowym w stosunku do materiału. Spójrz do kolejnych i poprzednich rozdziałów i czytaj je bez stresu. Przeglądając bardziej zaawansowany materiał bez pełnego zrozumienia szczegółów, możesz lepiej zrozumieć pytanie "dlaczego?" pojawiające się w programowaniu. Przeglądając wcześniejszy materiał, a nawet powtarzając poprzednie ćwiczenia, zdasz sobie sprawę, że faktycznie nauczyłeś się bardzo wielu rzeczy, nawet jeśli materiał, na który się właśnie patrzysz, wydaje się nieco nieprzystępny.

Zwykle gdy uczysz się swojego pierwszego języka programowania, to jest kilka wspaniałych chwil typu "a-ha!", w których możesz spojrzeć zza uderzeń młota i dłuta w jakąś skałę, odejść i zobaczyć, że rzeczywiście budujesz piękną rzeźbę.

Jeśli coś wydaje się szczególnie trudne, to zazwyczaj nie ma sensu zarywać całej nocy i wpatrywać się w ten problem. Zrób sobie przerwę, zdrzemnij się, zjedz przekąskę, wyjaśnij komuś (lub swojemu psu) z czym masz problem, a potem wróć do tego ze świeżym umysłem. Zapewniam Cię, że kiedy już nauczysz się z tej książki koncepcji programowania, to spojrzysz wstecz i zobaczysz, że wszystko było naprawdę łatwe i eleganckie, a przyswojenie tego zajęło Ci troszkę czasu.

## 1.13. Słowniczek

**błąd** Pomyłka w programie.

**błąd semantyczny** Błąd w programie, który zmusza go do zrobienia czegoś zupełnie innego niż zakładał programista.

**funkcja print** Instrukcja, która powoduje, że interpreter Pythona wyświetla daną wartość na ekranie.

**interpretowanie** Uruchomienie programu napisanego w języku wysokiego poziomy poprzez translację w danym momencie jednej linii kodu.

**język niskiego poziomu** Język programowania, który jest zaprojektowany by komputer mógł go łatwo uruchomić; czasem nazywany "kodem maszynowym" lub "językiem asemblera".

**język wysokiego poziomu** Język programowania taki jak Python, który jest zaprojektowany aby był łatwy do czytania i pisania przez ludzi.

**kod maszynowy** Język najniższego poziomu dla oprogramowania, który jest językiem bezpośrednio uruchamianym przez procesor (CPU).

**kod źródłowy** Program napisany w języku wysokiego poziomu.

**kompilacja** Tłumaczenie za jednym zamachem programu napisanego w języku wysokiego poziomu do języka niskiego poziomu w celu jego późniejszego uruchomienia.

**pamięć główna** Przechowuje programy i dane. Pamięć główna traci wszystkie swoje informacje gdy komputer zostanie wyłączony.

**pamięć pomocnicza** Przechowuje programy i dane oraz zachowuje ich informacje nawet gdy komputer zostanie wyłączony. Zasadniczo wolniejsza od pamięci głównej. Przykładami pamięci pomocniczej mogą być dyski twarde oraz pamięć flash w pamięciach USB/pendrive.

**parsowanie** Sprawdzanie programu i analizowanie jego struktury składniowej.

**procesor** Serce każdego komputera. Jest to to, co uruchamia oprogramowanie, które piszemy; czasem występuje pod nazwą "CPU".

**program** Zbiór instrukcji, które określają obliczenia.

**przenoszalność** Cecha programu, która pozawala na jego uruchomienia na więcej niż jednym komputerze.

**rozwiązywanie problemów** Proces formułowania problemu, znajdywania rozwiązania i wyrażania tego rozwiązania.

**semantyka** Znaczenie programu.

**tryb interaktywny** Sposób używania interpretera Pythona poprzez pisanie komend i wyrażeń w wierszu poleceń za znakiem zachęty.

**znak zachęty** Pojawia się gdy program wyświetla wiadomość i wstrzymuje swoje działanie by użytkownik mógł wprowadzić dane wejściowe do programu.

## 1.14. Ćwiczenia

**Ćwiczenie 1: Jaką rolę pełni w komputerze pamięć pomocnicza?**

a) Wykonuje wszystkie obliczenia i logikę programu
b) Pobiera strony z internetu
c) Przechowuje informacje przez dłuższy czas, nawet po wyłączeniu sprzętu
d) Pobiera dane wejściowe od użytkownika

**Ćwiczenie 2: What is a program?**

**Ćwiczenie 3: Jaka jest różnica między kompilatorem a intepreterem?**

**Ćwiczenie 4: Który z poniższych zawiera "kod maszynowy"?**

a) Interpreter Pythona
b) Klawiatura
c) Kod źródłowy Pythona
d) Dokument edytora tekstu

**Ćwiczenie 5: Co jest nie tak w poniższym kodzie:**

```
>>> primt 'Witaj świecie!'
File "<stdin>", line 1
```

```
  primt 'Witaj świecie!'
                  ^
SyntaxError: invalid syntax
>>>
```

**Ćwiczenie 6: Gdzie w komputerze będzie przechowywana zmienna "x" po wykonaniu poniższej linijki kodu Pythona?**

```
x = 123
```

a) W procesorze
b) W pamięci głównej
c) W pamięci pomocniczej
d) W urządzeniach wejścia
e) W urządzeniach wyjścia

**Ćwiczenie 7: Co wyświetli poniższy program:**

```
x = 43
x = x + 1
print(x)
```

a) 43
b) 44
c) x + 1
d) Błąd, ponieważ x = x + 1 łamie reguły matematyki

**Ćwiczenie 8: Wyjaśnij każdy z poniższych elementów na przykładzie ludzkich możliwości: (1) procesor, (2) pamięć główna, (3) pamięć pomocnicza, (4) Urządzenie wejścia, oraz (5) urządzenie wyjścia. Np. "Czym mógłby być procesor w kontekście człowieka?"?**

**Ćwiczenie 9: W jaki sposób naprawiasz "Syntax Error" (błąd składniowy)?**

# Rozdział 2

# Zmienne, wyrażenia i instrukcje

## 2.1. Wartości i typy

*Wartość* jest jedną z podstawowych rzeczy, z którymi program pracuje, takich jak litera czy cyfra. Wartości, które widzieliśmy do tej pory, to 1, 2 i "Witaj świecie!".

Wartości te należą do różnych *typów*: 2 jest liczbą całkowitą, a "Witaj świecie!" jest *ciągiem znaków*, "napisem" lub *łańcuchem znaków* (ang. *string*). Ty (i interpreter) możesz rozpoznać ciągi znaków, ponieważ są one zawarte między apostrofami lub w cudzysłowie.

Instrukcja `print` działa również w przypadku liczb całkowitych. Do uruchomienia interpretera używamy komendy `python`.

```
python
>>> print(4)
4
```

Jeśli nie jesteś pewien jaki typ ma dana wartość, interpreter może Ci to powiedzieć.

```
>>> type('Witaj świecie!')
<class 'str'>
>>> type(17)
<class 'int'>
```

Nic dziwnego, że ciągi znaków należą do typu `str`, a liczby całkowite (ang. *integers*) należą do typu `int`. Mniej oczywiste jest, że liczby z kropką dziesiętną należą do typu `float`, ponieważ liczby te są reprezentowane w formacie nazywanym *liczbą zmiennoprzecinkową*[1] (ang. *floating point*).

[1] Mimo że nazwa wskazuje na używanie przecinka i tak też robimy na co dzień w polskim zapisie, podczas programowania używamy kropki - przyp. tłum.

```
>>> type(3.2)
<class 'float'>
```

A co z takimi wartościami jak ‘17’ i ‘3.2’? Wyglądają one jak liczby, ale tak jak ciągi znaków, są zawarte między apostrofami.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

Są to ciągi znaków.

Gdy wpiszesz dużą liczbę całkowitą, być może ulegniesz pokusie użycia przecinków między grupami trzycyfrowymi, np. jak w przypadku 1,000,000.[2] Nie jest poprawna liczba całkowita w Pythonie, ale jest zapis całkowicie poprawny:

```
>>> print(1,000,000)
1 0 0
```

Cóż, nie tego się spodziewaliśmy! Python interpretuje 1,000,000 jako oddzielony przecinkami ciąg liczb całkowitych, które wypisuje ze spacjami.

Jest to pierwszy przykład omawianego wcześniej błędu semantycznego: kod działa bez generowania komunikatu o błędzie, ale nie robi tego, co “powinien” robić.

## 2.2. Zmienne

Jedną z najważniejszych cech języka programowania jest zdolność do operowania *zmiennymi*. Zmienna jest nazwą, która odnosi się do wartości.

*Instrukcja przypisania* tworzy nowe zmienne i nadaje im wartości:

```
>>> message = 'A teraz coś z zupełnie innej beczki'
>>> n = 17
>>> pi = 3.1415926535897931
```

Powyższy przykład składa się z trzech przypisań. Pierwsze wiersz przypisuje łańcuch znaków do nowej zmiennej o nazwie `message`; drugi przypisuje liczbę całkowitą 17 do `n`; trzeci przypisuje (przybliżoną) wartość $\pi$ do `pi`.

Aby wyświetlić wartość zmiennej, możesz skorzystać z instrukcji print:

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

[2] Jest to zapis częściej stosowany w USA.

Typ zmiennej jest typem wartości, do której się odnosi.

```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

## 2.3. Nazwy zmiennych i słowa kluczowe

Programiści zazwyczaj wybierają dla swoich zmiennych nazwy, które są sensowne i opisują do czego dana zmienna jest używana.

Nazwy zmiennych mogą być dowolnie długie. Mogą zawierać zarówno litery, jak i cyfry, ale nie mogą zaczynać się od cyfry. Użycie dużych liter jest poprawne, ale lepiej jest zacząć nazwy zmiennych od małej litery (później zobaczymy dlaczego).[3]

Znak podkreślenia ( _ ) może pojawić się w nazwie. Jest on często używany w nazwach zawierających wiele słów, takich jak `moje_ime` lub `predkosc_jaskolki_bez_obciazenia`. Nazwy zmiennych mogą zaczynać się od znaku podkreślenia, ale generalnie unikamy tego, chyba że piszemy kod biblioteki docelowo używanej przez inne osoby.

Jeśli nadasz zmiennej niepoprawną nazwę, otrzymasz błąd składniowy:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` jest niepoprawne, ponieważ zaczyna się od cyfry. `more@` jest niepoprawne, ponieważ zawiera niepoprawny znak `@`. Ale co jest złego w `class`?

Okazuje się, że `class` jest jednym ze *słów kluczowych* Pythona. Interpreter używa słów kluczowych do rozpoznania struktury programu, przez co nie można ich używać jako nazw zmiennych.

Python ma zarezerwowanych 35 słów kluczowych:

```
and       del       from      None      True
as        elif      global    nonlocal  try
assert    else      if        not       while
break     except    import    or        with
class     False     in        pass      yield
continue  finally   is        raise     async
def       for       lambda    return    await
```

[3] Co prawda w nazwach zmiennych można też używać np. polskich znaków diakrytycznych, ale lepiej trzymać się przyjętej w środowisku programistów konwencji i ich nie używać.

Być może zechcesz mieć tę listę gdzieś pod ręką. Jeśli interpreter zgłasza uwagi dotyczące jednej z Twoich zmiennych i nie wiesz dlaczego, sprawdź, czy jej nazwa jest na tej liście.

## 2.4. Instrukcje

*Instrukcja* jest jednostką kodu, którą może wykonać interpreter Pythona. Widzieliśmy dwa rodzaje wyrażeń: print jako wyrażenie oraz przypisanie.

Kiedy wpisujesz instrukcję w trybie interaktywnym, interpreter wykonuje ją i wyświetla wynik (jeśli taki istnieje).

Skrypt zazwyczaj zawiera sekwencję instrukcji. Jeżeli istnieje więcej niż jedna instrukcja, wyniki pojawiają się jeden po drugim podczas wykonywania kolejnych instrukcji.

Na przykład, skrypt

```
print(1)
x = 2
print(x)
```

zwraca wynik

```
1
2
```

Instrukcja przypisania nie zwraca żadnych wyników.

## 2.5. Operatory i operandy

*Operatory* są specjalnymi symbolami, które reprezentują obliczenia takie jak dodawanie i mnożenie. Wartości, na których operator jest stosowany, są nazywane *operandami*.

Operatory `+`, `-`, `*`, `/` i `**` wykonują odpowiednio dodawanie, odejmowanie, mnożenie, dzielenie i potęgowanie, jak w poniższych przykładach:

```
20+32
hour-1
hour*60+minute
minute/60
5**2
(5+9)*(15-7)
```

Nastąpiła zmiana w operatorze dzielenia między Pythonem 2.x a Pythonem 3.x. W Pythonie 3.x wynik takiego dzielenia jest wynikiem zmiennoprzecinkowym:

```
>>> minute = 59
>>> minute/60
0.9833333333333333
```

Operator dzielenia w Pythonie 2.0 dzieliłby dwie liczby całkowite i obcinałby wynik do liczby całkowitej:

```
>>> minute = 59
>>> minute/60
0
```

Aby uzyskać taki sam wynik w Pythonie 3.0, należy użyć dzielenia całkowitoliczbowego `//`, zaokrąglającego w dół.

```
>>> minute = 59
>>> minute//60
0
```

W Pythonie 3.0 dzielenie liczb całkowitych działa tak jakbyś wprowadził wyrażenie na kalkulatorze.

## 2.6. Wyrażenia

*Wyrażenie* jest kombinacją wartości, zmiennych i operatorów. Wartość sama w sobie jest uważana za wyrażenie, podobnie jak zmienna, a więc wszystkie poniższe są poprawnymi wyrażeniami (przy założeniu, że zmiennej `x` przypisano wcześniej jakąś wartość):

```
17
x
x + 17
```

Jeśli wpiszesz wyrażenie w trybie interaktywnym, interpreter dokona jego *ewaluacji* (wartościowania) i wyświetli wynik:

```
>>> 1 + 1
2
```

Natomiast w skrypcie wyrażenie samo w sobie nic nie robi! Jest to powszechne źródło dezorientacji wśród początkujących.

**Ćwiczenie 1: Wpisz następujące stwierdzenia w interpreterze Pythona, tak aby zobaczyć co one robią:**

```
5
x = 5
x + 1
```

## 2.7. Kolejność wykonywania działań

Kiedy w danym wyrażeniu pojawia się więcej niż jeden operator, kolejność ewaluacji zależy od *zasad pierwszeństwa*. W przypadku operatorów matematycznych, Python stosuje konwencję matematyczną. Akronim *PEMDAS* jest przydatnym sposobem na zapamiętanie tych reguł:

- Nawiasy okrągłe (ang. *P*arentheses) mają najwyższy priorytet i mogą być użyte do wymuszenia ewaluacji wyrażenia w pożądanej kolejności. Ponieważ wyrażenia w nawiasach są ewaluowane jako pierwsze, `2 * (3-1)` to 4, a `(1+1)**(5-2)` to 8. Możesz również użyć nawiasów aby wyrażenie było łatwiejsze do odczytania, tak jak np. w `(minute * 100) / 60`, nawet jeśli nie zmienia to wyniku.
- Potęgowanie (ang. *E*xponentiation) ma kolejny najwyższy priorytet, więc `2**1+1` to 3, a nie 4, a `3*1**3` to 3, a nie 27.
- Mnożenie (ang. *M*ultiplication) i dzielenie (ang. *D*ivision) mają ten sam priorytet, który jest wyższy niż dodawanie (ang. *A*ddition) i odejmowanie (ang. *S*ubtraction), które również mają ten sam priorytet. Tak więc `2*3-1` to 5, a nie 4, a `6+4/2` to 8, a nie 5.
- Operatory z takim samym pierwszeństwem są ewaluowane od lewej strony do prawej. Tak więc wyrażenie `5-3-1` to 1, a nie 3, ponieważ `5-3` jest wyliczane najpierw, a potem `1` jest odejmowane od 2.

W razie wątpliwości, zawsze umieszczaj nawiasy w swoich wyrażenia, tak aby upewnić się, że obliczenia zostały wykonane w zamierzonej kolejności.

## 2.8. Operator modulo

*Operator modulo* działa na liczbach całkowitych i zwraca resztę po podzieleniu pierwszego operanda przez drugi. W Pythonie operator modulo jest znakiem procentu (`%`). Składnia jest taka sama jak w przypadku innych operatorów:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

Tak więc 7 podzielone przez 3 to 2 z resztą 1.

Operator modulo okazuje się być zaskakująco użyteczny. Na przykład, możesz sprawdzić czy jedna liczba jest podzielna przez drugą: jeśli `x % y` wynos zero, to `x` jest podzielne przez `y`.

Możesz również wyodrębnić ostatnią cyfrę po prawej stronie cyfry z liczby. Na przykład, `x % 10` zwraca ostatnią cyfrę po prawej stronie z `x` (w systemie dziesiętnym). Podobnie, `x % 100` zwraca dwie ostatnie cyfry.

## 2.9. Operacje na ciągach znaków

Operator `+` działa z ciągami znaków, ale nie jest to dodawanie w sensie matematycznym. Zamiast tego wykonuje on *konkatenację*, co oznacza łączenie łańcuchów znaków poprzez złączenie ich końców. Na przykład:

```
>>> first = 10
>>> second = 15
>>> print(first+second)
25
>>> first = '100'
>>> second = '150'
>>> print(first + second)
100150
```

Operator `*` pracuje również z ciągami znkaów, powielając zawartość napisu tyle razy ile wynosi wartość liczby całkowitej. Na przykład:

```
>>> first = 'Test '
>>> second = 3
>>> print(first * second)
Test Test Test
```

## 2.10. Proszenie użytkownika o wprowadzenie danych

Czasami chcielibyśmy ustawić wartość zmiennej poprzez pobranie tej wartości od użytkownika za pomocą jego klawiatury. Python udostępnia wbudowaną funkcję o nazwie `input`, która pobiera dane z klawiatury[4]. Kiedy funkcja ta jest wywoływana, program zatrzymuje się i czeka, aż użytkownik coś wpisze. Gdy użytkownik naciśnie przycisk `Return` lub `Enter`, program wznawia działanie, a `input` zwraca to, co użytkownik wpisał jako ciąg znaków.

```
>>> inp = input()
Jakieś bzdury
>>> print(inp)
Jakieś bzdury
```

Przed otrzymaniem danych wejściowych od użytkownika, dobrze jest wyświetlić komunikat informujący użytkownika o tym, co należy wprowadzić. Możesz przekazać ciąg znaków do `input`, który zostanie wyświetlony użytkownikowi przed wstrzymaniem działania na czas wprowadzania danych:

```
>>> name = input('Jak masz na imię?\n')
Jak masz na imię?
```

[4]W Pythonie 2.0 funkcja ta nazywa się `raw_input`.

```
Chuck
>>> print(name)
Chuck
```

Sekwencja `\n` na końcu komunikatu reprezentuje *nową linię*, która jest specjalnym znakiem, który powoduje przerwanie aktualnej linii. Dlatego dane wprowadzane przez użytkownika znajdują się pod komunikatem.

Jeśli oczekujesz od użytkownika wprowadzenia liczby całkowitej, możesz spróbować przekonwertować zwracaną wartość na `int` używając funkcji `int()`:

```
>>> prompt = 'Jaka jest prędkość lotu jaskółki bez obciążenia?\n'
>>> speed = input(prompt)
Jaka jest prędkość lotu jaskółki bez obciążenia?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

Ale jeśli użytkownik wpisze coś innego niż ciąg cyfr, otrzymasz błąd:

```
>>> speed = input(prompt)
Jaka jest prędkość lotu jaskółki bez obciążenia?
Jakiej jaskółki? Afrykańskiej czy europejskiej?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
```

Zobaczymy później jak poradzić sobie z tego typu błędami.

## 2.11. Komentarze

W miarę jak programy stają się coraz większe i bardziej skomplikowane, stają się również coraz trudniejsze do odczytania. Formalne języki są skomplikowane i często trudno jest spojrzeć na kawałek kodu i zrozumieć co on robi lub dlaczego to robi.

Z tego powodu dobrym pomysłem jest dodawanie do programów notatek wyjaśniających co robi program, pisanych w języku naturalnym. Te notatki nazywane są *komentarzami*, a w Pythonie zaczynają się od symbolu `#`:

```
# oblicza procent godziny, która upłynęła
percentage = (minute * 100) / 60
```

W tym przypadku komentarz pojawia się samodzielnie w linii. Możesz również umieścić komentarz na końcu wiersza:

```
percentage = (minute * 100) / 60      # procent godziny
```

Wszystko od # do końca linii jest ignorowane; nie ma to żadnego wpływu na program.

Komentarze są najbardziej przydatne wtedy, gdy dokumentują nieoczywiste cechy kodu. Rozsądne jest założenie, że osoba czytająca kod może dojść do tego *co* robi kod; o wiele bardziej przydatne jest wyjaśnienie *dlaczego* to robi.

Poniższy komentarz w kodzie jest zbędny i bezużyteczny:

```
v = 5      # przypisz 5 do v
```

Natomiast ten komentarz zawiera przydatne informacje, których nie ma w kodzie:

```
v = 5      # prędkość w metrach na sekundę.
```

Dobre nazwy zmiennych mogą zmniejszyć potrzebę komentarzy, ale długie nazwy mogą sprawić, że złożone wyrażenia będą trudne do odczytania, więc trzeba tutaj iść na pewien kompromis.

## 2.12. Wybór mnemonicznych nazw zmiennych

Tak długo, jak stosujesz się do prostych zasad nazewnictwa zmiennych i unikasz zastrzeżonych słów, masz duży wybór przy nadawaniu nazw swoim zmiennym. Na początku wybór ten czasami może być dezorrientujący zarówno podczas czytania kodu programu, jak i pisania własnych programów. Na przykład, następujące trzy programy są identyczne pod względem tego co robią, ale są bardzo różne gdy je czytasz i starasz się je zrozumieć.

```
a = 35.0
b = 12.50
c = a * b
print(c)
```

```
hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)
```

```
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

Interpreter Pythona widzi wszystkie trzy te programy jako *dokładnie takie same*, ale ludzie widzą i rozumieją te programy zupełnie inaczej. Ludzie najszybciej zrozumieją *zamiary* drugiego programu, ponieważ programista wybrał nazwy zmiennych, które odzwierciedlają jego intencje dotyczące tego, jakie dane będą przechowywane w każdej z nich.

Te mądrze wybrane nazwy zmiennych nazywamy "mnemonicznymi nazwami zmiennych". Słowo *mnemonika* oznacza "zespół sposobów ułatwiających zapamiętywanie nowego materiału"[5]. Wybieramy mnemoniczne nazwy zmiennych aby pomóc nam przypomnieć sobie, po co stworzyliśmy daną zmienną.

Podczas gdy to wszystko brzmi świetnie i bardzo dobrym pomysłem jest używanie mnemonicznych nazw zmiennych, to mnemoniczne nazwy zmiennych mogą przeszkadzać początkującemu programiście w analizowaniu i rozumieniu kodu. Dzieje się tak dlatego, że początkujący programiści nie zapamiętali jeszcze zastrzeżonych słów (jest ich tylko 33) i czasami zmienne o nazwach zbyt opisowych zaczynają wyglądać jak część języka, a nie jak dobrze dobrane nazwy zmiennych.

Spójrzmy na poniższy przykładowy kod Pythona, który przechodzi w pętli po pewnych danych. Wkrótce zajmiemy się pętlami, ale na razie postarajmy się tylko zgadnąć, co oznacza ten kod:

```
for word in words:
    print(word)
```

Co tu się dzieje? Które z tokenów (for, word, in, itp.) są słowami zastrzeżonymi, a które są tylko nazwami zmiennych? Czy Python rozumie zapis "words" na poziomie fundamentalnym? Początkujący programiści mają problem z oddzieleniem tych częsci kodu, które *muszą* być takie same jak w tym przykładzie, od tych, które po prostu są wyborem dokonywanym przez programistę.

Poniższy kod jest odpowiednikiem powyższego kodu:

```
for slice in pizza:
    print(slice)
```

Początkującemu programiście łatwiej jest spojrzeć na ten kod i dowiedzieć się, które części są zastrzeżonymi słowami zdefiniowanymi przez Pythona, a które są po prostu zmiennymi nazwami wybranymi przez programistę. Jest całkiem zrozumiałe, że Python nie ma podstawowego pojęcia o pizzy i jej kawałkach oraz o tym, że pizza składa się z zestawu jednego lub więcej kawałków.

Ale jeśli nasz program naprawdę zajmuje się czytaniem danych i szukaniem słów w danych, `pizza` i `slice` to bardzo niemonmoniczne nazwy zmiennych. Wybranie ich jako nazw zmiennych odwraca uwagę od znaczenia programu.

Dość szybko zapamiętasz najczęstsze zastrzeżone słowa i zaczniesz widzieć jak Cię atakują:

Części kodu, które są zdefiniowane przez Pythona (`for`, `in`, `print` i `:`) są tutaj pogrubione, a zmienne wybrane przez programistę (`word` i `words`) nie są pogrubione. Wiele edytorów tekstu jest świadomych składni Pythona i inaczej pokoloruje zastrzeżone słowa, tak aby dać ci wskazówki jak oddzielić zmienne od zastrzeżonych słów. Po pewnym czasie zaczniesz biegle czytać kod Pythona i będziesz szybko określać co jest zmienną, a co słowem zastrzeżonym.[6]

[5] https://sjp.pwn.pl/sjp/mnemotechnika;2568165.html

[6] Dodatkowo warto zwrócić uwagę, że zgodnie z przyjętą konwencją programistyczną nazwy zmiennych oraz komentarze powinny być w języku angielskim. Podczas realizacji tego kursu nie ma to większego znaczenia, gdyż tutaj programy piszemy dla siebie, jednak warto o tym pamiętać w przyszłości podczas tworzenia bardziej zaawansowanych programów.

## 2.13. Debugowanie

W tym momencie najbardziej prawdopodobnym błędem składniowym jest niepoprawna nazwa zmiennej, np. `class` i `yield`, które są słowami kluczowymi, lub `odd~job` i `US$`, które zawierają niedozwolone znaki.

Jeśli umieścisz spację w nazwie zmiennej, Python pomyśli, że są to dwa operandy bez operatora:

```
>>> bad name = 5
SyntaxError: invalid syntax
```

W przypadku błędów składniowych, komunikaty o błędach nie są zbyt pomocne. Najczęstszym komunikatem jest `SyntaxError: invalid syntax`. Czasem jednak można trafić na bardziej pomocny komunikat o błędzie składniowym

```
>>> month = 09
  File "<stdin>", line 1
    month = 09
            ^
SyntaxError: leading zeros in decimal integer literals are not
↪  permitted;
            use an 0o prefix for octal integers
```

Widzimy tutaj, że zera widoące w liczbach całkowitych są niedozwolone, a Python podpowiada, że może chodzi nam o zapis w systemie ósemkowym, gdzie liczby zaczynają się od `0o`.

Błąd wykonania (ang. *runtime error*), który jest najprawdopodobniejszy do popełnienia przez Ciebie, to "użycie przed definicją", co oznacza próbę użycia zmiennej przed przypisaniem wartości. Może się to zdarzyć wtedy, gdy błędnie wpiszesz nazwę zmiennej:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

W nazwach zmiennych wielkość liter jest rozróżniana, więc `LaTeX` nie jest tym samym co `latex`.

W tym momencie, najbardziej prawdopodobną przyczyną błędu semantycznego jest kolejność operacji. Na przykład, aby obliczyć $1/2\pi$, można pokusić się o napisanie:

```
>>> 1.0 / 2.0 * pi
```

Ale dzielenie wykonuje się pierwsze, więc dostałbyś w rezultacie $\pi/2$, a to nie to samo! Nie ma możliwości by Python wiedział co chciałeś napisać, więc w tym przypadku nie dostaniesz komunikatu o błędzie; po prostu dostaniesz złą odpowiedź.

## 2.14. Słowniczek

**ciąg znaków** Typ danych `str`, który reprezentuje sekwencje znaków; inaczej napis lub łańcuch znaków.

**ewaluacja** Uproszczenie wyrażenia poprzez wykonanie operacji w celu uzyskania pojedynczej wartości; inaczej wartościowanie.

**instrukcja** Część kodu, która reprezentuje polecenie lub akcję. Jak dotąd, polecenia, które widzieliśmy, to przypisania i wyrażenie print.

**kolejność wykonywania działań** Zbiór zasad regulujących kolejność ewaluacji wyrażeń obejmujących wiele operatorów i operandów.

**komentarz** Informacja w programie, która jest przeznaczona dla innych programistów (lub osób czytających kod źródłowy) i nie ma wpływu na wykonanie programu.

**konkatenacja** Połączenie dwóch ciągów znaków.

**liczba całkowita** Typ danych `int`, który reprezentuje liczby całkowite.

**liczba zmiennoprzecinkowa** Typ danych `float`, który reprezentuje liczby z częściami ułamkowymi.

**mnemonika** Pomoc pamięciowa. Często nadajemy zmiennym nazwy mnemoniczne, tak aby pomóc nam zapamiętać co jest przechowywane w zmiennej.

**operand** Jedna z wartości, na której działa operator.

**operator** Specjalny symbol, który reprezentuje proste obliczenie, np. takie jak dodawanie, mnożenie lub konkatenację ciągów.

**operator modulo** Operator oznaczony znakiem procentu (`%`), który działa na liczbach całkowitych i zwraca resztę gdy jedna liczba jest dzielona przez drugą.

**przypisanie** Instrukcja, która przypisuje wartość zmiennej.

**słowo kluczowe** Zastrzeżone słowo, które jest używane przez kompilator/interpreter do przetwarzania programu; nie można używać słów kluczowych np. takich jak `if`, `def` i `while` jako nazw zmiennych.

**typ** Kategoria wartości. Typy, które widzieliśmy do tej pory, to liczby całkowite (typ `int`), liczby zmiennoprzecinkowe (typ `float`) oraz ciągi znaków (typ `str`).

**wartość** Jedna z podstawowych jednostek danych, takich jak liczba lub ciąg znaków, którą operuje program.

**wyrażenie** Kombinacja zmiennych, operatorów i wartości, która reprezentuje pojedynczą wartość wyniku.

**zmienna** Nazwa, która odnosi się do wartości.

## 2.15. Ćwiczenia

**Ćwiczenie 2: Napisz program, który wykorzystuje funkcję `input` do poproszenia użytkownika o jego imię, a następnie przywita go używając jego imienia.**

```
Podaj swoje imię: Chuck
Hello Chuck
```

**Ćwiczenie 3: Napisz program, który wyświetli użytkownikowi pytanie o liczbę godzin pracy i stawkę za godzinę w celu obliczenia wynagrodzenia.**

```
Podaj liczbę godzin: 39
Podaj stawkę godzinową: 28.75
Wynagrodzenie: 1121.25
```

Na razie nie będziemy się martwić o to, by nasze wynagrodzenie miało dokładnie dwie cyfry po przecinku. Jeśli chcesz, możesz pobawić się wbudowaną funkcją Pythona `round`, tak aby prawidłowo zaokrąglić wynagrodznie do dwóch miejsc po przecinku.

**Ćwiczenie 4: Załóżmy, że wykonujemy następujące instrukcje przypisania:**

```
width = 17
height = 12.0
```

Dla każdego z poniższych wyrażeń podaj wartość wyrażenia i oraz typ (wartości wyrażenia).

1. `width//2`
2. `width/2.0`
3. `height/3`
4. `1 + 2 * 5`

Aby sprawdzić swoje odpowiedzi, użyj interpretera Pythona.

**Ćwiczenie 5: Napisz program, który prosi użytkownika o podanie temperatury w skali Celsjusza, przelicza ją na skalę Fahrenheita i wyświetla przeliczoną temperaturę.**

# Rozdział 3

# Wykonanie warunkowe

## 3.1. Wyrażenia logiczne

*Wyrażenie logiczne* to wyrażenie, które jest prawdziwe lub fałszywe. Poniższe przykłady używają operatora `==`, który porównuje dwa operandy i zwraca `True`, jeśli są one równe, lub `False` w przeciwnym wypadku:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` i `False` są specjalnymi wartościami, które należą do klasy `bool`; nie są one ciągami znaków:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Operator `==` jest jednym z *operatorów porównania*; pozostałe to:

```
x != y               # x nie jest równe y
x > y                # x jest większe od y
x < y                # x jest mniejsze od y
x >= y               # x jest większe lub równe y
x <= y               # x jest mniejsze lub równe y
x is y               # x jest tym samym co y
x is not y           # x nie jest tym samym co y
```

Choć te operacje są pewnie Ci znane, symbole Pythona różnią się od symboli matematycznych dla tych samych operacji. Częstym błędem jest użycie pojedynczego znaku równości (`=`) zamiast znaku podwójnej równości (`==`). Pamiętaj, że `=` jest operatorem przypisania, a `==` jest operatorem porównania. Nie ma czegoś takiego jak `=<` lub `=>`.

## 3.2. Operatory logiczne

Istnieją trzy *operatory logiczne*: `and`, `or` i `not`. Semantyka (znaczenie) tych operatorów jest podobna do ich znaczenia w języku angielskim. Na przykład,

`x > 0 and x < 10`

jest prawdziwe tylko wtedy, gdy `x` jest większe niż 0 *i* mniejsze niż 10.

`n%2 == 0 or n%3 == 0` jest prawdziwe, jeśli *którykolwiek* z tych warunków jest prawdziwy, to znaczy, jeśli liczba jest podzielna przez 2 *lub* 3.

Operator `not` neguje wyrażenie logiczne, więc `not (x > y)` jest prawdziwe, jeśli `x > y` jest fałszywe, tzn. jeśli `x` jest mniejsze lub równe `y`.

Ściślej mówiąc, operandy operatorów logicznych powinny być wyrażeniami logicznymi, ale Python nie jest tutaj bardzo restrykcyjny. Każda dodatnia liczba całkowita jest interpretowana jako "prawdziwa".

```
>>> 17 and True
True
```

Ta elastyczność może być użyteczna, ale istnieją pewne niuanse, które mogą być mylące. Być może będziesz chciał ich unikać, dopóki nie będziesz pewien, że wiesz, co robisz.

## 3.3. Instrukcja warunkowa

Aby napisać użyteczny program, prawie zawsze potrzebujemy możliwości sprawdzenia pewnych warunków i odpowiedniej zmiany zachowania programu. *Instrukcje warunkowe* dają nam tę możliwość. Najprostszą formą jest instrukcja `if`:

```
if x > 0 :
    print('x jest dodatnia')
```

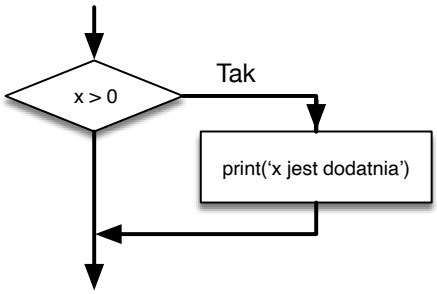
Wyrażenie logiczne po instrukcji `if` jest nazywane *warunkiem*. Kończymy wyrażenie `if` znakiem dwukropka (:), a linia (linie) po wyrażeniu `if` są wcięte.

Jeśli warunek logiczny jest prawdziwy, to wykonywane jest wcięcie wyrażenia. Jeśli warunek logiczny jest fałszywy, to wcięcie wyrażenia jest pomijane.

Instrukcje `if` mają taką samą strukturę jak definicje funkcji lub pętle `for`[1]. Instrukcja składa się z linii nagłówka, która kończy się znakiem dwukropka (:), po którym następuje wcięty blok (ciało instrukcji `if`). Takie instrukcje nazywane są *instrukcjami złożonymi*, ponieważ rozciągają się na więcej niż jedną linię.

Nie ma ograniczenia co do liczby instrukcji, które mogą pojawić się w ciele instrukcji `if`, ale musi być co najmniej jedno. Od czasu do czasu warto mieć ciało `if` bez instrukcji (zazwyczaj jako miejsce na kod, którego jeszcze nie napisałeś). W takim przypadku możesz użyć `pass`, który nie robi nic.

[1] Dowiemy się o funkcjach w rozdziale 4, a o pętlach w rozdziale 5.




Rysunek 3.1: Logika instrukcji if

```
if x < 0 :
    pass          # trzeba obsłużyć wartości ujemne!
```

Jeśli wpiszesz instrukcję `if` w interpreterze Pythona, znak zachęty zmieni się z jodełki na trzy kropki, tak aby wskazać, że jesteś w środku bloku instrukcji:

```
>>> x = 3
>>> if x < 10:
...     print('malutko')
...
malutko
>>>
```

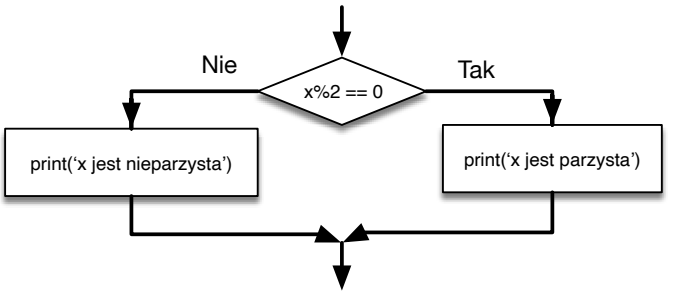
Kiedy używasz interpretera Pythona, musisz zostawić pustą linię na końcu bloku, bo w przeciwnym razie Python zwróci błąd:

```
>>> x = 3
>>> if x < 10:
...     print('malutko')
... print('zrobione')
  File "<stdin>", line 3
    print('zrobione')
        ^
SyntaxError: invalid syntax
```

Pusta linia na końcu bloku instrukcji nie jest konieczna podczas pisania i wykonywania *skryptu*, ale może poprawić czytelność Twojego kodu.

## 3.4. Wykonanie alternatywnego bloku kodu

Drugą formą instrukcji `if` jest *wykonanie alternatywne*, w którym istnieją dwie możliwości i to warunek określa, która z nich zostanie wykonana. Składnia wygląda następująco:




Rysunek 3.2: Logika if-else

```
if x%2 == 0 :
    print('x jest parzysta')
else :
    print('x jest nieparzysta')
```

Jeśli reszta z dzielenia `x` przez 2 jest równa 0, to wiemy, że zmienna `x` jest parzysta, a program wyświetla stosowny komunikat. Jeśli warunek jest fałszywy, to wykonywany jest drugi zestaw poleceń.

Ponieważ warunek musi być albo prawdziwy, albo fałszywy, dokładnie jedna z alternatyw zostanie wykonana. Te alternatywy nazywane są *gałęziami*, ponieważ są gałęziami w przepływie wykonania programu.

## 3.5. Warunki powiązane

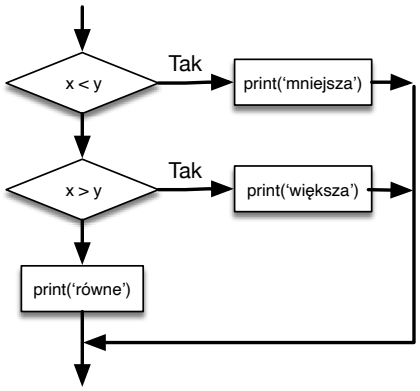
Czasami istnieje więcej niż dwie możliwości i potrzebujemy więcej niż dwóch gałęzi. Jednym ze sposobów na wyrażenie takich obliczeń jest *warunek powiązany*:

```
if x < y:
    print('x jest mniejsza niż y')
elif x > y:
    print('x jest większa niż y')
else:
    print('x i y są równe')
```

`elif` jest skrótem od "else if". Ponownie, dokładnie jedna gałąź zostanie wykonana.

Nie ma ograniczenia co do liczby instrukcji `elif`. Jeśli istnieje klauzula `else`, to musi ona być na końcu (ale samo jej pojawienie się nie jest wymagane).

```
if choice == 'a':
    print('Zła odpowiedź')
elif choice == 'b':
    print('Dobra odpoiwedź')
elif choice == 'c':
    print('Blisko, ale źle')
```




Rysunek 3.3: Logika if-elif

Każdy warunek jest kolejno sprawdzany. Jeśli pierwszy jest fałszywy, sprawdzany jest następny itd. Jeśli jeden z nich jest prawdziwy, to wykonywana jest odpowiadająca mu gałąź, a instrukcja jest kończona. Nawet jeśli więcej niż jeden warunek jest prawdziwy, to tylko pierwszy z tych prawdziwych wykonuje swoją gałąź kodu.

## 3.6. Warunki zagnieżdżone

Jeden warunek może być również zagnieżdżony w innym. Mogliśmy tak napisać przykład z trzema gałęziami:

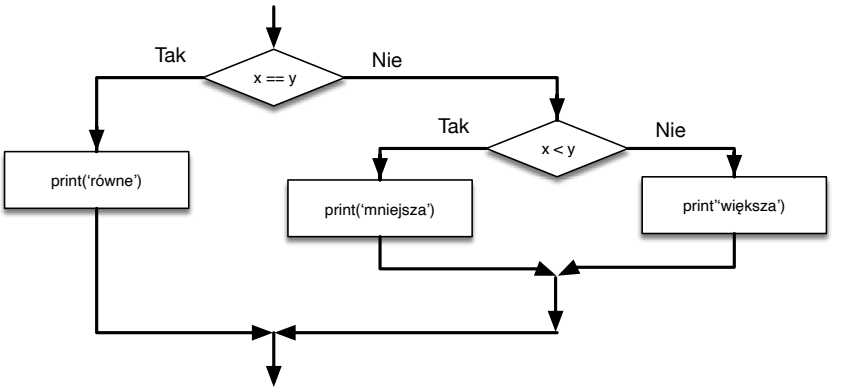
```
if x == y:
    print('x i y są równe')
else:
    if x < y:
        print('x jest mniejsza niż y')
    else:
        print('x jest większa niż y')
```

Warunek zewnętrzny zawiera dwie gałęzie. Pierwsza gałąź zawiera prostą instrukcję. Druga gałąź zawiera kolejną instrukcję `if`, która ma dwie własne gałęzie. Te dwie gałęzie to proste instrukcje, chociaż mogłyby być również instrukcjami warunkowymi.

Chociaż wcięcie tych instrukcji sprawia, że ich struktura jest widoczna, to jednak *zagnieżdżone instrukcje warunkowe* bardzo szybko stają się trudne do odczytania. Ogólnie rzecz biorąc, jeśli możesz, to ich unikaj.

Operatory logiczne często są sposobem na uproszczenie zagnieżdżonych instrukcji warunkowych. Na przykład, możemy przepisać następujący kod za pomocą jednego warunku:

```
if 0 < x:
    if x < 10:
        print('x jest jednocyfrową dodatnią liczbą.')
```




Rysunek 3.4: Zagnieżdżone instrukcje if

Instrukcja `print` jest wykonywana tylko wtedy, gdy spełnimy oba warunki, więc możemy uzyskać ten sam efekt z operatorem `and`:

```
if 0 < x and x < 10:
    print('x jest jednocyfrową dodatnią liczbą.')
```

## 3.7. Łapanie wyjątków przy użyciu try i except

Wcześniej widzieliśmy fragment kodu, w którym używaliśmy funkcji `input` i `int` do odczytania i przetworzenia liczby całkowitej wprowadzonej przez użytkownika. Widzieliśmy też, jak takie podejście może być zdradzieckie:

```
>>> prompt = "Jaka jest prędkość lotu jaskółki bez obciążenia?\n"
>>> speed = input(prompt)
Jaka jest prędkość lotu jaskółki bez obciążenia?
Jakiej jaskółki? Afrykańskiej czy europejskiej?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
>>>
```

Kiedy wykonamy te polecenia w interpreterze Pythona, otrzymujemy od niego prośbę o wprowadzenie danych, interpreter pomyśli "ups!" i przejdzie do naszej następnej instrukcji.

Jeżeli jednak umieścisz ten kod w skrypcie Pythona i wystąpi błąd, to Twój skrypt natychmiast się zatrzyma w miejscu pojawienia się błędu, wyświetlając przy tym informacje z mechanizmu *traceback*, który pozwala zobaczyć kolejne wywołania funkcji, które ostatecznie doprowadziły do wystąpienia błędu. Po wystąpieniu błędu skrypt nie wykona kolejnych instrukcji.

Oto przykładowy program do konwersji temperatury Fahrenheita na temperaturę Celsjusza:

```
inp = input('Podaj temperaturę w skali Fahrenheita: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)

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

Jeśli uruchomimy ten kod i podamy mu błędne dane wejściowe, to program po prostu zakończy się niepowodzeniem z niemiłym komunikatem o błędzie:

```
python fahren.py
Podaj temperaturę w skali Fahrenheita: 72
22.22222222222222

python fahren.py
Podaj temperaturę w skali Fahrenheita: fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

W Pythonie jest wbudowana warunkowa struktura wykonania poleceń, która obsługuje tego typu oczekiwane i nieoczekiwane błędy, zwana "try / except". Idea `try` i `except` polega na tym, że wiesz, iż pewna sekwencja instrukcji może sprawić problem i chcesz dodać kilka poleceń do wykonania w przypadku wystąpienia błędu. Te dodatkowe polecenia (blok "except") są ignorowane gdy nie ma błędu.

Możesz myśleć o `try` i `except` w Pythonie jako o "polisie ubezpieczeniowej" na sekwencję poleceń.

Możemy przepisać nasz konwerter temperatury w następujący sposób:

```
inp = input('Podaj temperaturę w skali Fahrenheita: ')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('Musisz wprowadzić liczbę')

# Kod źródłowy: https://pl.py4e.com/code3/fahren2.py
```

Python zaczyna od wykonania sekwencji poleceń w bloku `try`. Jeśli wszystko pójdzie dobrze, pomija blok `except` i przechodzi dalej. Jeśli wystąpi wyjątek w bloku `try`, Python wyskakuje z bloku `try` i wykonuje sekwencję instrukcji w bloku `except`.

```
python fahren2.py
Podaj temperaturę w skali Fahrenheita: 72
22.22222222222222
```

```
python fahren2.py
Podaj temperaturę w skali Fahrenheita: fred
Musisz wprowadzić liczbę
```

Obsługa wyjątku z instrukcją `try` nazywana jest *łapaniem* wyjątku. W tym przykładzie, klauzula `except` wypisuje komunikat o błędzie. Ogólnie rzecz biorąc, złapanie wyjątku daje Ci szansę naprawienia problemu, spróbowania ponownie, albo przynajmniej zgrabnego zakończenia programu.

## 3.8. Minimalna ewaluacja wyrażeń logicznych

Gdy Python przetwarza wyrażenie logiczne takie jak `x >= 2 and (x/y) > 2`, ewaluuje je od lewej do prawej strony. Ze względu na definicję `and`, jeśli `x` jest mniejsze niż 2, wyrażenie `x >= 2` zostanie ewaluowane do `False`, a więc całe wyrażenie zostanie ewaluowane do `False` niezależnie od tego, czy `(x/y) > 2` zostanie ewaluowane do `True` czy `False`.

Gdy Python wykryje, że nic już nie zyska ewaluując pozostałą część wyrażenia logicznego, przerywa jego ewaluację i nie wykonuje obliczeń w pozostałej części wyrażenia logicznego. Gdy ocena wyrażenia logicznego zostaje zatrzymana, ponieważ ogólna wartość jest już znana, nazywa się to *minimalną ewaluacją*.

Specyfika działania minimalnej ewaluacji ma konsekwencję w sprytnej technice zwanej *wzorcem strażnika*. Przeanalizujmy następującą sekwencję kodu w interpreterze Pythona:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

Trzecie obliczenie nie powiodło się, ponieważ Python ewaluował `(x/y)` i `y` było zerem, co spowodowało błąd wykonania. Ale pierwszy i drugi przykład *nie* zakończyły się błędami, ponieważ pierwsza część tych wyrażeń `x >= 2` została ewaluowana do `False`, więc warunek `(x/y)` nigdy nie został sprawdzony właśnie z powodu reguły minimalnej ewaluacji, przez co nie było tutaj żadnego błędu.

Możemy w następujący sposób skonstruować takie wyrażenie logiczne, aby strategicznie umieścić *strażnika* tuż przed ewaluacją potencjalnie powodującą błąd:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

W pierwszym wyrażeniu logicznym, `x >= 2` wynosi `False`, więc ewaluacja kończy się na `and`. W drugim wyrażeniu logicznym, `x >= 2` wynosi `True`, ale `y != 0` wynosi `False`, więc nigdy nie dochodzimy do próby ewaluacji `(x/y)`.

W trzecim wyrażeniu logicznym, `y != 0` jest *po* obliczeniu `(x/y)`, więc wyrażenie kończy się błędem.

W drugim wyrażeniu mówimy, że `y != 0` działa jak *strażnik*, tak by zapewnić, że wykonamy `(x/y)` tylko wtedy, gdy `y` nie jest zerem.

## 3.9. Debugowanie

Mechanizm *traceback* w Pythonie pojawia się w przypadku wystąpienia błędu; zawiera on wiele informacji, ale czasem może być przytłaczający. Najbardziej użyteczne są zazwyczaj informacje o tym:

- jaki to był błąd,
- gdzie ten błąd wystąpił.

Błędy składniowe są zazwyczaj łatwe do znalezienia, ale jest kilka podstępnych przypadków. Błędy związane z tzw. białymi znakami[2] mogą być trudne, ponieważ np. spacje i tabulacje są niewidoczne i jesteśmy przyzwyczajeni do ich ignorowania.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

W powyższym przykładzie problem polega na tym, że druga linia jest wcięta przez jedną spację. Ale komunikat o błędzie wskazuje na `y`, co jest mylące. Ogólnie

[2]Należą do nich spacje, znaki tabulacji, znaki końca linii oraz dowolne inne znaki niemające kształtu na ekranie.

rzecz biorąc, komunikaty o błędach wskazują, gdzie problem został wykryty, ale rzeczywisty błąd może być w kodzie gdzieś wcześniej, czasami w poprzedniej linii.

Ogólnie rzecz biorąc, komunikaty o błędach informują o tym, gdzie wykryto problem, ale często nie tam, gdzie został on popełniony.

## 3.10. Słowniczek

**ciało instrukcji** Sekwencja instrukcji zawarta w złożonej instrukcji.
**gałąź** Jedna z alternatywnych sekwencji instrukcji występujących w instrukcji warunkowej.
**instrukcja warunkowa** Instrukcja, które kontroluje przepływ wykonywania programu w zależności od pewnego warunku.
**instrukcja złożona** Instrukcja składająca się z nagłówka i ciała. Nagłówek kończy się dwukropkiem (:). Ciało jest wcięte w stosunku do nagłówka.
**minimalna ewaluacja** Kiedy Python jest w trakcie ewaluacji złożonego wyrażenia logicznego i przerywa dalszą ewaluację, ponieważ Python zna końcową wartość wyrażenia bez potrzeby ewaluacji pozostałej reszty wyrażenia.
**operator logiczny** Jeden z operatorów, który łączy wyrażenia logiczne: `and`, `or` lub `not`.
**operator porównania** Jeden z następujących operatorów, który porównuje swoje operandy: `==`, `!=`, `>`, `<`, `>=` i `<=`.
**traceback** Lista funkcji, które są wykonywane; pojawia się, gdy wystąpi wyjątek.
**warunek** Wyrażenie logiczne w instrukcji warunkowej określające która gałąź jest wykonywana.
**warunki powiązane** Wyrażenie warunkowe z serią alternatywnych gałęzi.
**warunki zagnieżdżone** Instrukcja warunkowa, które pojawia się w jednej z gałęzi innej instrukcji warunkowej.
**wyrażenie logiczne** Wyrażenie o wartości `True` lub `False`.
**wzorzec strażnika** Miejsce, w którym konstruujemy wyrażenie logiczne z dodatkowymi porównaniami, tak aby skorzystać z działania minimalnej ewaluacji.

## 3.11. Ćwiczenia

**Ćwiczenie 1: Przepisz ponownie swój program obliczający wynagrodzenie, tak aby dać pracownikowi 1,5 raza większą stawkę godzinową za czas przepracowany powyżej 40 godzin.**

```
Podaj liczbę godzin: 45
Podaj stawkę godzinową: 10
Wynagrodzenie: 475.0
```

**Ćwiczenie 2: Przepisz ponownie swój program płacowy używając `try` i `except`, tak aby elegancko obsługiwał on nie-numeryczne dane wejściowe, wyświetlając w takim przypadku wiadomość i kończąc swoje działanie. Poniżej znajdują się wyniki dwóch uruchomień programu:**

```
Podaj liczbę godzin: 20
Podaj stawkę godzinową: dziewięć
Błąd, podaj wartość numeryczną

Podaj liczbę godzin: czterdzieści
Błąd, podaj wartość numeryczną
```

**Ćwiczenie 3: Napisz program, który poprosi użytkownika o wartość pomiędzy 0.0 a 1.0. Jeśli wartość jest poza zakresem, wypisz komunikat o błędzie. Jeśli wartość jest między 0.0 a 1.0, wypisz ocenę, korzystając z poniższej tabeli:**

```
Wartość   Ocena
>= 0.9     5,0
>= 0.8     4,5
>= 0.7     4,0
>= 0.6     3,5
>= 0.5     3,0
 < 0.5     2,0

Podaj wartość: 0.95
5,0

Podaj wartość: doskonała
Niepoprawna wartość

Podaj wartość: 10.0
Niepoprawna wartość

Podaj wartość: 0.75
4,0

Podaj wartość: 0.5
3,0

Podaj wartość: 0.46
2,0
```

Uruchom program wielokrotnie, jak pokazano powyżej, tak aby przetestować różne wartości.

# Rozdział 4

# Funkcje

## 4.1. Wywoływanie funkcji

W kontekście programowania, *funkcja* to nazwana sekwencja instrukcji, która wykonuje pewne obliczenia. Kiedy definiujesz funkcję, określasz jej nazwę i kolejność instrukcji. Później możesz "wywołać" funkcję po jej nazwie. Widzieliśmy już jeden przykład *wywołania funkcji*:

```
>>> type(32)
<class 'int'>
```

Nazwa funkcji to `type`. Wyrażenie w nawiasach nazywane jest *argumentem* funkcji. Argument jest wartością lub zmienną, którą przekazujemy do funkcji jako dane wejściowe funkcji. Wynikiem, dla funkcji `type`, jest typ przekazanego argumentu.

Zwykle mówi się, że funkcja "przyjmuje" argument i "zwraca" wynik. Wynik jest nazywany *wartością zwracaną*.

## 4.2. Funkcje wbudowane

Python zapewnia szereg ważnych funkcji wbudowanych, z których możemy korzystać bez konieczności podawania definicji tych funkcji. Twórcy Pythona napisali zestaw funkcji do rozwiązywania popularnych problemów i włączyli je do Pythona, tak abyśmy mogli z nich korzystać.

Funkcje `max` i `min` dają nam odpowiednio największe i najmniejsze wartości występujące w liście:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

Funkcja `max` mówi nam o "największym znaku" w ciągu znaków (który okazuje się być literą "w"), a funkcja `min` pokazuje nam najmniejszy znak (który okazuje się być spacją).

Inną bardzo często używaną funkcją jest funkcja `len`, która mówi nam, ile elementów znajduje się w przekazanym argumencie. Jeśli argument `len` jest ciągiem znaków, to zwraca liczbę znaków w tym ciągu.

```
>>> len('Hello world')
11
>>>
```

Funkcje te nie ograniczają się do analizowania wyłącznie łańcuchów znaków. Mogą one działać na dowolnym zestawie wartości, jak zobaczymy w późniejszych rozdziałach.

Nazwy wbudowanych funkcji należy traktować jako słowa zastrzeżone (tzn. unikać używania "max" jako nazwy zmiennej).

## 4.3. Funkcje konwersji typu

Python oferuje również funkcje wbudowane, które konwertują wartości z jednego typu na drugi. Funkcja `int` przyjmuje dowolną wartość i konwertuje ją na liczbę całkowitą, jeśli może, lub awanturuje się przeciwnym przypadku:

```
>>> int('32')
32
>>> int('Halo')
ValueError: invalid literal for int() with base 10: 'Halo'
```

`int` może zamieniać wartości zmiennoprzecinkowe na liczby całkowite, ale nie zaokrągla ich (ucina część ułamkową):

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` konwertuje liczby całkowite i ciągi znaków na liczby zmiennoprzecinkowe:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Natomiast `str` konwertuje swój argument na ciąg znaków:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

## 4.4. Funkcje matematyczne

Python posiada moduł `math`, który dostarcza większość znanych funkcji matematycznych. Zanim będziemy mogli skorzystać z tego modułu, musimy go zaimportować:

```
>>> import math
```

Powyższa instrukcja tworzy *obiekt modułu* o nazwie math. Jeśli spróbujesz wydrukować na ekranie obiekt modułu, otrzymasz kilka informacji na jego temat:

```
>>> print(math)
<module 'math' (built-in)>
```

Obiekt modułu zawiera funkcje i zmienne zdefiniowane w tym module. Aby uzyskać dostęp do jednej z tych funkcji, musisz podać nazwę modułu i nazwę funkcji, oddzielone kropką. Format ten nazywa się *notacją kropkową*.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

Pierwszy przykład oblicza logarytm przy podstawie 10 ze stosunku sygnału (`signal_power`) do szumu (`noise_power`) przemnożony przez 10. Moduł matematyczny udostępnia również funkcję o nazwie `log`, która oblicza logarytm naturalny.

Drugi przykład znajduje sinus z wartości w zmiennej `radians`. Nazwa zmiennej jest podpowiedzią, że `sin` i inne funkcje trygonometryczne (`cos`, `tan`, itp.) przyjmują argumenty w radianach. Aby przeliczyć wartość ze stopni na radiany, należy ją podzielić przez 360 i pomnożyć przez $2\pi$:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
```

Wyrażenie `math.pi` pobiera zmienną `pi` z modułu matematycznego. Wartość tej zmiennej jest przybliżeniem $\pi$ z dokładnością do około 15 cyfr.

Jeśli znasz trygonometrię, to możesz sprawdzić poprzedni wynik, porównując go z pierwiastkiem kwadratowym z dwóch podzielonym przez dwa:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

## 4.5. Liczby losowe

Biorąc pod uwagę te same dane wejściowe, większość programów komputerowych generuje za każdym razem te same dane wyjściowe, więc w takich przypadkach mówi się, że te programy są *deterministyczne*. Determinizm jest zazwyczaj dobrą rzeczą, ponieważ oczekujemy, że to samo obliczenie da ten sam wynik. W przypadku niektórych aplikacji chcemy jednak, by komputer był nieprzewidywalny. Gry są oczywistym przykładem, ale jest ich więcej.

Uczynienie programu prawdziwie niedeterministycznym okazuje się wcale nie takie proste, ale są sposoby, by przynajmniej wydawał się niedeterministycznym. Jednym z nich jest użycie *algorytmów*, które generują liczby *pseudolosowe*. Liczby pseudolosowe nie są tak naprawdę losowe, ponieważ są generowane przez deterministyczne obliczenia, ale po prostu patrząc na te liczby, nie da się ich odróżnić od losowych.

Moduł `random` oferuje funkcje, które generują liczby pseudolosowe (które od tej chwili będę po prostu nazywał "losowymi").

Funkcja `random` zwraca losową liczbę pomiędzy 0.0 a 1.0 (włączając w to 0.0, ale nie 1.0). Za każdym razem, gdy wywołujesz `random`, otrzymujesz następną liczbę. Aby zobaczyć krótki przykład, uruchom poniższą pętlę:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

Powyższy program tworzy następującą listę 10 losowych liczb z zakresu od 0.0 do 1.0 (z wyłączeniem 1.0).

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

**Ćwiczenie 1: Uruchom program na swoim systemie i zobacz jakie dostajesz liczby. Uruchom program więcej niż raz i zobacz jakie tym razem dostajesz liczby.**

Funkcja `random` jest tylko jedną z wielu funkcji, które obsługują liczby losowe. Funkcja `randint` pobiera parametry `low` oraz `high` i zwraca liczbę całkowitą pomiędzy `low` i `high` (włączając w to obydwie te liczby).

```
>>> random.randint(5, 10)
5
```

```
>>> random.randint(5, 10)
9
```

Aby wybrać losowo element z jakiejś sekwencji, możesz użyć `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

Moduł `random` zapewnia również funkcje do generowania wartości losowych z rozkładów ciągłych, w tym rozkładu Gaussa, wykładniczego, gamma i kilku innych.

## 4.6. Dodawanie nowych funkcji

Do tej pory korzystaliśmy tylko z funkcji dostarczanych wraz z Pythonem, ale możliwe jest również dodawanie nowych funkcji. *Definicja funkcji* określa nazwę nowej funkcji i kolejność poleceń, które wykonują się podczas jej wywołania. Kiedy już zdefiniujemy jakąś funkcję, będziemy mogli używać jej ile wlezie w całym programie.

Oto krótki przykład:

```
def print_lyrics():
    print("Jestem sobie drwal i równy chłop.")
    print('Pracuję w dzień i śpię całą noc.')
```

`def` to słowo kluczowe, które wskazuje, że jest to definicja funkcji. Nazwa funkcji to `print_lyrics`. Zasady dotyczące nazw funkcji są takie same jak dla nazw zmiennych: litery, cyfry i niektóre znaki interpunkcyjne są dozwolone, ale pierwszy znak nie może być liczbą. Nie możesz używać słowa kluczowego jako nazwy funkcji i powinieneś unikać posiadania zmiennej i funkcji o tej samej nazwie.

Puste nawiasy po nazwie oznaczają, że ta funkcja nie przyjmuje żadnych argumentów. Później zbudujemy funkcje, które przyjmują argumenty jako swoje dane wejściowe.

Pierwszy wiersz definicji funkcji nazywa się *nagłówkiem* funkcji; reszta nazywa się *ciałem* funkcji. Nagłówek musi się kończyć dwukropkiem, a ciało musi być wcięte. Zgodnie z konwencją, wcięcie jest zawsze czterema spacjami. Ciało może zawierać dowolną liczbę instrukcji.

Jeżeli wpiszesz definicję funkcji w trybie interaktywnym, interpreter wypisze trzy kropki (...) aby dać Ci znać, że definicja nie jest kompletna:

```
>>> def print_lyrics():
...     print("Jestem sobie drwal i równy chłop.")
...     print('Pracuję w dzień i śpię całą noc.')
...
```

Aby zakończyć funkcję, musisz wpisać pustą linię (w skrypcie nie jest to konieczne).

Zdefiniowanie funkcji tworzy zmienną o tej samej nazwie.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

Wartość `print_lyrics` to *obiekt funkcji* o typie "function".

Składnia wywoływania nowej funkcji jest taka sama jak dla funkcji wbudowanych:

```
>>> print_lyrics()
Jestem sobie drwal i równy chłop.
Pracuję w dzień i śpię całą noc.
```

Po zdefiniowaniu funkcji, możemy jej używać wewnątrz innej funkcji. Na przykład, aby powtórzyć poprzednią zwrotkę, możemy napisać funkcję o nazwie `repeat_lyrics`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

Następnie wywołujemy `repeat_lyrics`:

```
>>> repeat_lyrics()
Jestem sobie drwal i równy chłop.
Pracuję w dzień i śpię całą noc.
Jestem sobie drwal i równy chłop.
Pracuję w dzień i śpię całą noc.
```

No ale tak naprawdę to nie tak idzie ta piosenka.

## 4.7. Definiowanie i używanie

Składając razem fragmenty kodu z poprzedniej sekcji, cały program wygląda następująco:

```
def print_lyrics():
    print("Jestem sobie drwal i równy chłop.")
    print('Pracuję w dzień i śpię całą noc.')


def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

```
repeat_lyrics()

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

Program ten zawiera dwie definicje funkcji: `print_lyrics` i `repeat_lyrics`. Definicje funkcji wykonywane są tak samo jak inne polecenia, ale efektem jest tworzenie obiektów funkcji. Polecenia wewnątrz funkcji są wykonywane dopiero po jej wywołaniu, a definicja funkcji nie generuje danych wyjściowych.

Jak pewnie się spodziewasz, musisz stworzyć funkcję, zanim będziesz mógł ją wykonać. Innymi słowy, definicja funkcji musi zostać wykonana przed jej pierwszym wywołaniem.

**Ćwiczenie 2: Przesuń ostatnią linię programu na samą górę, tak aby wywołanie funkcji pojawiło się przed definicjami. Uruchom program i zobacz jaki otrzymasz komunikat o błędzie.**

**Ćwiczenie 3: Przesuń wywołanie funkcji na sam dół i przenieś definicję `print_lyrics` po definicji `repeat_lyrics`. Co się stanie, gdy uruchomisz taki program?**

## 4.8. Przepływ sterowania

Aby zapewnić, że funkcja zostanie zdefiniowana przed jej pierwszym użyciem, musisz znać kolejność, w jakiej wykonywane są instrukcje - tę kolejność nazywa się *przepływem sterowania*.

Wykonanie poleceń rozpoczyna się zawsze od pierwszej instrukcji programu. Instrukcje są wykonywane pojedynczo, w kolejności od góry do dołu.

*Definicje* funkcji nie zmieniają przepływu sterowania programu, ale pamiętaj, że instrukcje wewnątrz funkcji nie są wykonywane, dopóki funkcja nie zostanie wywołana.

Wywołanie funkcji można traktować w przepływie sterowania jak objazd. Zamiast przejść do następnego polecenia, przepływ sterowania przeskakuje do ciała funkcji, wykonuje tam wszystkie polecenia, a następnie wraca do miejsca, w którym zostało on przerwany.

Brzmi to wystarczająco prosto, dopóki nie przypomnisz sobie, że jedna funkcja może wywołać inną. Będąc w środku jednej funkcji, program może być zmuszony do wykonania poleceń w innej funkcji. Ale podczas wykonywania tej nowej funkcji, program może być zmuszony do wykonania jeszcze jednej funkcji!

Na szczęście Python jest dobry w śledzeniu gdzie się znajduje, więc za każdym razem, gdy funkcja się kończy, program wznawia działanie tam gdzie przerwał działanie w funkcji, która ją wywołała. Kiedy dociera do końca programu, to program się kończy.

Jaki jest morał tej strasznej opowieści? Kiedy czytasz kod programu, nie zawsze chcesz go czytać od góry do dołu. Czasami większy sens ma podążanie za jego przepływem sterowania.

## 4.9. Parametry i argumenty

Niektóre z wbudowanych funkcji, które widzieliśmy, wymagają argumentów. Na przykład, gdy wywołujesz `math.sin`, podajesz liczbę jako argument. Niektóre funkcje przyjmują więcej niż jeden argument: `math.pow` przyjmuje dwa, tj. podstawę i wykładnik.

Wewnątrz funkcji argumenty są przypisane do zmiennych nazywanych *parametrami*. Oto przykład zdefiniowanej przez użytkownika funkcji, która przyjmuje argument:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

Funkcja ta przypisuje argument do parametru o nazwie `bruce`. Gdy funkcja zostanie wywołana, wypisuje wartość parametru (czymkolwiek on jest) dwukrotnie.

Funkcja ta działa z dowolną wartością, która może być wypisana.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

Te same zasady składania wyrażeń lub instrukcji, które odnoszą się do funkcji wbudowanych, dotyczą także funkcji zdefiniowanych przez użytkownika, więc możemy użyć dowolnego rodzaju wyrażenia jako argumentu w `print_twice`:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

Argument jest ewaluowany przed wywołaniem funkcji, więc w powyższych przykładach wyrażenia `'Spam '*4` i `math.cos(math.pi)` są ewaluowane tylko raz.

Możesz również użyć zmiennej jako argumentu:

```
>>> michael = 'Eryk Pół-Ćma.'
>>> print_twice(michael)
Eryk Pół-Ćma.
Eryk Pół-Ćma.
```

Nazwa zmiennej, którą przekazujemy jako argument (`michael`) nie ma nic wspólnego z nazwą parametru (`bruce`). Nie ma znaczenia jak na tę wartość wołano w domu (w miejscu wywoływania); tutaj w `print_twice` wszystkich nazywamy `bruce`.

## 4.10. Funkcje owocne i funkcje niezwracające wartości

Niektóre z używanych przez nas funkcji to np. funkcje matematyczne, które zwracają wyniki; z braku lepszej nazwy, nazywam je *funkcjami owocnymi* (ang. *fruitful functions*). Inne funkcje, takie jak `print_twice`, wykonują jakąś akcję, ale nie zwracają wartości. Nazywa się je *funkcjami niezwracającymi wartości*[1] (ang. *void functions*).

Kiedy wołasz owocną funkcję, prawie zawsze chcesz zrobić coś ze zwróconym wynikiem. Na przykład, możesz przypisać go do zmiennej lub użyć go jako części wyrażenia:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

Kiedy wywołujesz funkcję w trybie interaktywnym, Python wyświetla wynik:

```
>>> math.sqrt(5)
2.23606797749979
```

Ale w skrypcie, jeśli wywołasz owocną funkcję i nie zapiszesz wyniku funkcji w zmiennej, to zwracana wartość zniknie we mgle!

```
math.sqrt(5)
```

Powyższy skrypt oblicza pierwiastek kwadratowy z 5, ale ponieważ nie zapisuje wyniku w zmiennej ani nie wyświetla zwróconego wyniku, nie jest on zbyt użyteczny.

Funkcje niezwracające wartości mogą wyświetlać coś na ekranie lub mieć jakiś inny efekt. W związku z tym, że nie zwracają żadnej wartości, jeśli spróbujesz przypisać wynik takiej funkcji do zmiennej, otrzymasz specjalną wartość o nazwie `None`.

```
>>> result = print_twice('Ping')
Ping
Ping
>>> print(result)
None
```

Wartość `None` nie jest taka sama jak ciąg znaków “None”. Jest to specjalna wartość, która ma swój własny typ:

[1] W niektórych językach programowania tego typu funkcje nazywa się procedurami.

```
>>> print(type(None))
<class 'NoneType'>
```

Aby zwrócić wynik z funkcji, używamy w niej instrukcji `return`. Na przykład, możemy napisać bardzo prostą funkcję o nazwie `addtwo`, która dodaje dwie liczby i zwraca wynik dodawania.

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)

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

Gdy powyższy skrypt zostanie wykonany, instrukcja `print` wypisze "8", ponieważ funkcja `addtwo` została wywołana z argumentami 3 i 5. Wewnątrz funkcji, parametry `a` i `b` miały wartości odpowiednio 3 i 5. Funkcja obliczyła sumę tych dwóch liczb i umieściła ją w zmiennej lokalnej o nazwie `added`. Następnie użyła instrukcji `return`, by wysłać wyliczoną wartość z powrotem do kodu wywołującego funkcję, gdzie zwrócona wartość została przypisana do zmiennej `x` i wypisany.

## 4.11. Dlaczego potrzebujemy funkcji?

Na początku może nie być jasne dlaczego warto podzielić program na funkcje. Jest kilka powodów:

- Stworzenie nowej funkcji daje Ci możliwość nazwania grupy poleceń, co czyni Twój program łatwiejszym do odczytania, zrozumienia i debugowania.
- Funkcje mogą sprawić, że program będzie mniejszy poprzez wyeliminowanie powtarzającego się kodu. Później, jeśli dokonasz zmiany kodu, będziesz musiał zrobić to tylko w jednym miejscu.
- Dzielenie długiego programu na funkcje pozwala na debugowanie po kolei jego części, a następnie złożenie ich w działającą całość.
- Dobrze zaprojektowane funkcje są często przydatne w wielu programach. Kiedy już napiszesz i zdebugujesz jedną z nich, możesz ją ponownie użyć w innym programie.

W pozostałej części książki często będziemy używać definicji funkcji, tak by wyjaśnić pewne pojęcie. Częścią umiejętności tworzenia i korzystania z funkcji jest posiadanie funkcji, która właściwie uchwyci cel taki jak "znajdź najmniejszą wartość na liście wartości". Później pokażemy Ci kod, który znajduje najmniejszą wartość na liście wartości i przedstawimy go jako funkcję o nazwie `min`, która przyjmuje listę wartości jako swój argument i zwraca najmniejszą wartość występującą na tej liście.

## 4.12. Debugowanie

Jeśli używasz edytora tekstu do pisania swoich skryptów, możesz mieć problemy ze spacjami i tabulacjami. Najlepszym sposobem na uniknięcie tych problemów jest użycie wyłącznie spacji (bez tabulacji). Większość edytorów tekstowych, które znają specyfikę Pythona, robi to domyślnie, ale niektóre nie.

Tabulacje i spacje są zazwyczaj niewidoczne, co utrudnia ich debugowanie, więc postaraj się znaleźć edytor, który zarządza wcięciami dla Ciebie.

Nie zapomnij również zapisać swojego programu, zanim go uruchomisz. Niektóre środowiska programistyczne robią to automatycznie, ale niektóre nie. W takim przypadku program, który oglądasz w edytorze tekstu, nie jest taki sam, jak program, który uruchamiasz.

Debugowanie może trwać bardzo długo jeśli będziesz ciągle uruchamiać ten sam nieprawidłowy program!

Debugging can take a long time if you keep running the same incorrect program over and over!

Upewnij się, że kod, na który patrzysz, jest tym samym kodem, który uruchamiasz. Jeśli nie jesteś pewien, umieść coś w rodzaju `print("hello")` na początku programu i uruchom go ponownie. Jeśli nie widzisz `hello`, to nie uruchamiasz właściwego programu![2]

## 4.13. Słowniczek

**algorytm** Ogólny proces rozwiązywania pewnej kategorii problemów.
**argument** Wartość przekazywana do funkcji w momencie jej wywołania. Wartość ta jest przypisana do odpowiadającego jej parametru w funkcji.
**ciało funkcji** Sekwencja instrukcji wewnątrz definicji funkcji.
**definicja funkcji** Instrukcja, która tworzy nową funkcję, określająca jej nazwę, parametry i polecenia, które wykonuje.
**deterministyczny** Dotyczy programu, który robi to samo za każdym razem, gdy dostarczy mu się te same dane wejściowe.
**funkcja** Nazwany ciąg instrukcji wykonujący jakąś użyteczną operację. Funkcje mogą, ale nie muszą, przyjmować argumenty i mogą, ale nie muszą, zwracać wyniki.
**funkcja niezwracająca wartości** Funkcja, które nie posiada zwracanej wartości.

**funkcja owocna** Funkcja zwracająca wartość.
**instrukcja import** Instrukcja, która wczytuje plik modułu i tworzy obiekt modułu.
**obiekt modułu** Wartość utworzona przez instrukcję `import`, która zapewnia dostęp do danych i kodu zdefiniowanego w module.
**nagłówek funkcji** Pierwszy wiersz definicji funkcji.

[2] Ta technika debugowania nosi nazwę *caveman debugging* (z ang. debugowanie metodą jaskiniowca) lub *print debugging* i nie jest zalecana podczas profesjonalnego programowania, jednak dobrze sprawdza się podczas pierwszych kroków w nauce programowania.

**notacja kropkowa** Składnia do wywołania funkcji w innym module poprzez określenie nazwy modułu, po której następuje kropka i nazwa funkcji.
**obiekt funkcji** Wartość utworzona przez definicję funkcji. Nazwa funkcji jest zmienną, która odnosi się do obiektu funkcji.
**parametr** Nazwa używana wewnątrz funkcji do odwoływania się do wartości przekazanej jako argument.
**przepływ sterowania** Kolejność, w której polecenia są wykonywane podczas uruchamiania programu.
**pseudolosowość** Odnosi się do ciągu liczb, które wydają się być losowe, ale są generowane przez deterministyczny program.
**składanie** Użycie wyrażenia jako części większego wyrażenia lub instrukcji jako części większej instrukcji.
**wartość zwracana** Wynik działania funkcji. Jeżeli wywołanie funkcji zostanie użyte jako wyrażenie, to wartość zwracana jest wartością tego wyrażenia.
**wywołanie funkcji** Polecenie, które uruchamia funkcję. Składa się z nazwy funkcji, po której następuje lista argumentów.

## 4.14. Ćwiczenia

**Ćwiczenie 4: Jaki jest cel słowa kluczowego "def" w Pythonie?**

a) Jest to slang, który oznacza, że "następujący kod jest naprawdę fajny"
b) Oznacza początek funkcji
c) Oznacza, że następujące wcięcie sekcji kodu ma być przechowywane na później
d) Odpowiedzi b) i c) są poprawne
e) Żadne z powyższych

**Ćwiczenie 5: Co wypisze następujący program Pythona?**

```
def fred():
   print("Zap")

def jane():
   print("ABC")

jane()
fred()
jane()
```

a) Zap ABC jane fred jane
b) Zap ABC Zap
c) ABC Zap jane
d) ABC Zap ABC
e) Zap Zap Zap

**Ćwiczenie 6: Przepisz ponownie swoje obliczenie wynagrodzenia z dodatkiem za nadgodziny i stwórz funkcję o nazwie `computepay`, która przyjmuje dwa parametry (`hours` i `rate`).**

```
Podaj liczbę godzin: 45
```

```
Podaj stawkę godzinową: 10
Wynagrodzenie: 475.0
```

**Ćwiczenie 7: Napisz ponownie program z poprzedniego rozdziału do wyliczania ocen za pomocą funkcji o nazwie `computegrade`, która przyjmuje jako parametr wartość i zwraca ocenę jako ciąg znaków.**

**Exercise 7: Rewrite the grade program from the previous chapter using a function called `computegrade` that takes a score as its parameter and returns a grade as a string.**

```
Wartość   Ocena
>= 0.9     5,0
>= 0.8     4,5
>= 0.7     4,0
>= 0.6     3,5
>= 0.5     3,0
 < 0.5     2,0


Podaj wartość: 0.95
5,0


Podaj wartość: doskonała
Niepoprawna wartość


Podaj wartość: 10.0
Niepoprawna wartość


Podaj wartość: 0.75
4,0


Podaj wartość: 0.5
3,0


Podaj wartość: 0.46
2,0
```

Uruchom program kilkukrotnie, tak aby przetestować różne wartości.

# Rozdział 5

# Iteracja

## 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 `x`:

```
>>> x = x + 1
NameError: name 'x' 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 są często używane do automatyzacji powtarzalnych zadań. Powtarzanie identycznych lub podobnych zadań bez popełniania błędów jest czymś, co komputery robią dobrze, a ludzie źle. Ponieważ iteracja często występuje w programach, Python oferuje kilka elementów, które ułatwiają z nimi 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!')
```

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 słowo `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, 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 która kontroluje kiedy pętla się kończy, nazywamy *zmienną sterującą*. Jeśli nie ma zmiennej sterującej, pętla będzie powtarzać się 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.

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

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 lub przypomnisz sobie gdzie znajduje się przycisk wyłączania komputera. Program ten będzie działał w nieskończoność lub aż do wyczerpania baterii, ponieważ wyrażenie logiczne na górze pętli jest zawsze prawdziwe z uwagi na fakt, że wyrażenie jest stałą wartością `True`.

Podczas gdy jest to patologiczna nieskończona pętla, możemy nadal używać tego schematu do budowania użytecznych pętli tak długo, jak starannie dodamy kod do ciała pętli, tak aby wyraźnie wyjść z niej używając polecenia `break` wtedy, gdy osiągniemy warunek wyjścia.

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

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

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

Warunkiem pętli jest `True`, co jest zawsze prawdą, więc pętla działa wielokrotnie, tak długo aż trafi w 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 poprzez kombinację 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. W tym przypadku możesz użyć instrukcji `continue`, tak aby przejść do następnej iteracji bez kończenia ciała pętli w bieżącej iteracji.

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://pl.py4e.com/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 hasza, 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 instrukcję `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 przeprocesowania 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`, podczas gdy 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[2] trzech ciągów znaków, a pętla `for` przechodzi przez listę i wykonuje ciało raz dla każdego z trzech ciągów znaków 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 myślisz 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 pętlę `for`, 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 ciągi znaków 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:

[2] 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 liczbę pozycji na liście, moglibyśmy napisać następującą pętlę `for`:

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

Powyższy program demonstruje inny 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 z 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.

Kolejna podobna pętla, która oblicza sumę liczb z danej listy, jest następująca:

```
total = 0
for itervar in [3, 41, 12, 9, 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, dodajemy rzeczywistą liczbę (3, 41, 12, itd.) do sumy bieżącej podczas każdej iteracji pętli. Jeśli myślisz 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, podczas pętli `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*.

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

### 5.6.2. Pętle typu maksimum i minimum

Aby znaleźć największą wartość na liście lub sekwencji, tworzymy następującą pętlę:

```
largest = None
print('Przed:', largest)
for itervar in [3, 41, 12, 9, 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: 3 3
Pętla: 41 41
Pętla: 12 41
Pętla: 9 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`. `None` jest specjalną stałą wartością, którą możemy przechowywać w zmiennej, tak by oznaczyć ją jako "pustą".

Przed rozpoczęciem pętli, największą wartością, jaką do tej pory widzieliśmy, jest `None`, ponieważ nie widzieliśmy jeszcze żadnych wartości. Podczas wykonywania pętli, jeśli `largest` to `None` to bierzemy pierwszą wartość, którą do tej pory widzieliśmy, jako największą. Możesz zobaczyć w pierwszej iteracji, że wartość `itervar` wynosi 3, ponieważ `largest` to `None`, więc natychmiast ustawiamy `largest` na 3.

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 wartość, która jest większa niż "największa do tej pory". Gdy widzimy nową "jeszcze większą" wartość, bierzemy tę nową wartość za `largest`. Możesz zobaczyć na wyjściu programu, że `largest` zwiększa się z 3 do 41, a potem do 74.

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

Aby obliczyć najmniejszą liczbę, kod jest bardzo podobny, ale z jedną małą zmianą:

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

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

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

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

Kiedy zaczniesz pisać większe programy, być może będziesz spędzać więcej czasu na debugowaniu. 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 instrukcję `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 i o miejscach, w których łatwo jest to sprawdzić. Następnie wybierz miejsce, w którym Twoim zdaniem szanse na występienie 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, które nadaje wartość początkową zmiennej, 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" wydrukuj sumę, ile wprowadzono liczb oraz średnią z tych liczb. Jeśli użytkownik wprowadzi coś innego niż liczba, wykryj jego błąd używając `try` i `except`, 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.**

# Rozdział 6

# Ciągi znaków

## 6.1. Ciąg znaków jest sekwencją

Ciąg znaków jest *sekwencją* znaków. Możesz uzyskać dostęp do poszczególnych znaków za pomocą operatora nawiasów:

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

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

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

Ale prawdopodobnie nie otrzymasz tego, czego oczekujesz:

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

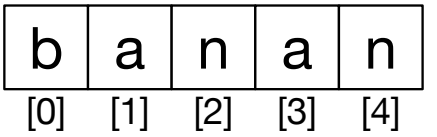
Dla większości ludzi pierwsza litera słowa "banan" to "b", a nie "a". Jednak w Pythonie indeks oznacza przesunięcie od początku ciągu znaków, a przesunięcie od pierwszej litery wynosi zero.

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

Tak więc "b" to 0-wa ("zerowa") litera słowa "banan", "a" to 1-sza litera, a "n" to 2-ga 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 ciągach znaków

## 6.2. Uzyskanie długości ciągu znaków przy użyciu `len`

`len` jest funkcją wbudowaną, która zwraca liczbę znaków w danym ciągu znaków:

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

Aby otrzymać ostatnią literę ciągu znaków, 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 numerami 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 ciągu znaków. Wyrażenie `fruit[-1]` daje ostatnią literę, `fruit[-2]` daje przedostatnią itd.

## 6.3. Przejście przez ciąg znaków przy użyciu pętli

Wiele obliczeń polega na przetwarzaniu tekstu znak po znaku. Często zaczynają się one od początku ciągu znaków, 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 ciągu znaków i wyświetla w wierszu pojedynczo każdą literę. Warunkiem pętli jest `index < len(fruit)`, więc gdy `index` jest równy długości łańcucha, 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 ciągu znaków.

**Ćwiczenie 1: Napisz pętlę `while`, która zaczyna się od ostatniego znaku w ciągu znaków i działa w kierunku przeciwnym do pierwszego znaku, wypisując każdą literę w osobnej linii, ale tym razem w odwrotnej kolejności.**

Innym sposobem na napisanie przejścia jest użycie pętli `for`:

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

Za każdym razem idąc przez pętlę, kolejny znak w łańcuchu jest przypisywany do zmiennej `char`. Pętla jest kontynuowana aż do momentu, gdy nie pozostaną już żadne znaki do przypisania.

## 6.4. Wycinki z ciągu znaków

Segment ciągu znaków nazywany jest *wycinkiem* lub *kawałkiem*. Wybranie wycinka jest podobne do wybrania znaku:

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

Operator dwukropka zwraca część ciągu znaków 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 ciągu znaków. Jeśli pominiesz drugi indeks, wycinek jest pobierany aż do końca ciągu znaków:

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

Jeśli pierwszy indeks jest większy lub równy drugiemu, wynikiem jest *pusty ciąg znaków*, reprezentowany przez dwa apostrofy (zamiast apostrofu możemy użyć cudzysłowu):

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

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

**Część 2: Zakładając, że `fruit` jest ciągiem znaków, co oznacza `fruit[:]`?**

## 6.5. Ciągi znaków są niezmienne

Kuszące jest skorzystanie z operatora przypisania z zamiarem zmiany znaku w ciągu znaków. Na przykład:

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

Obiekt ("object") w tym przypadku to ciąg znaków, a element ("item") jest znakiem, do który próbowałeś coś przypisać. Na tę chwilę *obiekt* jest tym samym co wartość, ale definicję tę doszlifujemy później. *Element* jest jedną z wartości w sekwencji.

Powodem błędu jest to, że ciągi znaków są *niezmienne*, co oznacza, że nie można zmienić istniejącego ciągu znaków. Jedyne co możesz zrobić, to stworzyć nowy łańcuch, 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 ciąg znaków.

## 6.6. Przechodzenie w pętli i zliczanie

Poniższy program zlicza ile razy litera "a" pojawia się w ciągu znaków:

```
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 występień 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 ciąg znaków lub literę.**

## 6.7. Operator in

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

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

## 6.8. Porównywanie ciagów znaków

Operatory porównania działają również na ciągach znaków. Aby sprawdzić, czy dwa ciągi są równe:

```
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 ciągów znaków do standardowego formatu, tj. 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 założeń programowania obiektowego, ale na tę chwilę nie wchodźmy za bardzo w szczegóły tego zagadnienia.

## 6.9. Metody obiektów będących ciągami znaków

Ciągi znaków są przykładem *obiektów* Pythona. Obiekt zawiera zarówno dane (zasadniczy ciąg znaków), 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, a funkcja `dir` pokazuje dostępne dla niego metody.

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

Podczas gdy funkcja `dir` wyświetla listę metod, a Ty możesz użyć `help` by uzyskać prostą dokumentację jakiejś metody, w przypadku metod ciągów znaków lepszym źródłem dokumentacji jest 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` bierze ciąg znaków i zwraca nowy ciąg znaków, 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 ciągu znaków, do którego metoda 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 ciągów znaków istnieje metoda o nazwie `find`, która szuka pozycji jednego ciągu 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 podciągi, jak i znaki:

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

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 ciągu znaków 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, bierzemy linię i konwertujemy wszystko 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 jesteśmy ostrożni z kolejnością, w jednym wyrażeniu możemy wykonywać wiele wywołań metod.

**Ćwiczenie 4: Istnieje metoda dla ciągu znaków 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 ciągów znaków

Często chcemy zajrzeć do ciągu znaków i znaleźć pewie podciąg. Na przykład, jeśli przedstawiono nam serię linii 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` operacji wyciągania wycinków z ciągów znaków.

Najpierw znajdziemy w ciągu znaków pozycję znaku `@`. Następnie znajdziemy pozycję pierwszej spacji *po* znaku `@`. Na końcu użyjemy wycinków by wyodrębnić część ciągu znaków, 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 pozycję w łańcuchu, od której chcemy by `find` zaczął szukać. 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 ciągów znaków, zastępując części ciągów danymi przechowywanymi w zmiennych. W przypadku zastosowania go do liczb całkowitych, `%` jest operatorem modulo. Ale gdy pierwszym operandem jest ciąg znaków, `%` jest operatorem formatowania.

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

Na przykład, sekwencja formatująca `%d` oznacza, że drugi operand powinien być sformatowany jako liczba całkowita ("d" oznacza po angielsku "decimal, czyli"dziesiętny"):

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

Wynikiem jest ciąg znaków '42', którego nie należy mylić z liczbą całkowitą o wartości 42.

Sekwencja formatowania może pojawić się w dowolnym miejscu łańcucha, więc możesz osadzić jakąś wartość w zdaniu:

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

Jeśli w ciągu znaków znajduje się więcej niż jedna sekwencja formatująca, to drugim argumentem musi być krotka[2]. Każda sekwencja formatują 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 ciągu znaków[3]:

```
>>> 'W ciągu %d lat zauważyłem %g %s.' % (3, 0.1, 'wielbłąda')
'W ciągu 3 lat zauważyłem 0.1 wielbłąda.'
```

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

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

[2] Krotka to sekwencja wartości oddzielonych przecinkami wewnątrz pary nawiasów. Omówimy krotki w rozdziale 10.

[3] "s" od angielskiego "string", czyli ciąg znaków.

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:

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 może się 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://pl.py4e.com/code3/copytildone2.py
```

Spójrz co się stanie gdy użytkownik wprowadzi w 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 zerowgo, 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 ciąg znaków 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 ciągu znaków jest co najmniej jeden znak.

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

## 6.13. Słowniczek

**ciąg formatujący** Ciąg znaków, używany z operatorem formatowania, zawierający sekwencje formatowania.

**element** Jedna z wartości w sekwencji.

**indeks** Wartość całkowita używana do zaznaczenia elementu w sekwencji, np. znaku w ciagu znaków.

**metoda** Funkcja, która jest związana z obiektem i jest wywoływana przy użyciu notacji kropkowej.

**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 ciąg formatujący oraz krotkę i generuje nowy ciąg znaków zawierający elementy krotki sformatowane zgodnie z opisem podanym w ciągu formatującym.

**przejście** Iteracja po elementach sekwencji, wykonująca jakąś podobną operację na każdym z nich.

**pusty ciąg znaków** Ciąg bez żadnych znaków i o długości 0, reprezentowany przez dwa apostrofy.

**sekwencja** Uporządkowany zbiór, tj. zbiór wartości, gdzie każda wartość jest sekwencja przez indeks będący liczbą całkowitą.

**sekwencja formatująca** Sekwencja znaków w ciągu formatującym, np. `%d`, która określa, jak dana wartość powinna być sformatowana.

**wycinek** Część ciągu znaków określonego przez zakres indeksów.

**wywołanie metody** Instrukcja, która wykonuje funkcję ściśle związaną z danym obiektem i jego instancjami.

## 6.14. Ćwiczenia

**Ćwiczenia 5: Weź następujący kod Pythona, który przechowuje ciąg znaków:**

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

**Użyj `find` i wycinków ciągów znaków, tak aby wyodrębnić część ciągu po znaku dwukropka, a następnie użyj funkcji `float`, tak aby przekształcić wyodrębniony ciąg znaków w liczbę zmiennoprzecinkową.**

**Ćwiczenia 6: Przeczytaj dokumentację metod związanych z ciągami znaków na stronie https://docs.python.org/library/stdtypes.html#string-**

**methods. Możesz poeksperymentować z niektórymi z nich, tak 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.**

# Rozdział 7

# Pliki

## 7.1. Pamięć nieulotna

Do tej pory uczyliśmy się pisać programy i komunikować *procesorowi* nasze zamiary za pomocą instrukcji warunkowych, funkcji i iteracji. Nauczyliśmy się jak tworzyć i wykorzystywać struktury danych w *pamięci głównej*. Procesor i pamięć główna są miejscem, w którym działa i uruchamia się nasze oprogramowanie. To właśnie tam odbywa się całe "myślenie".

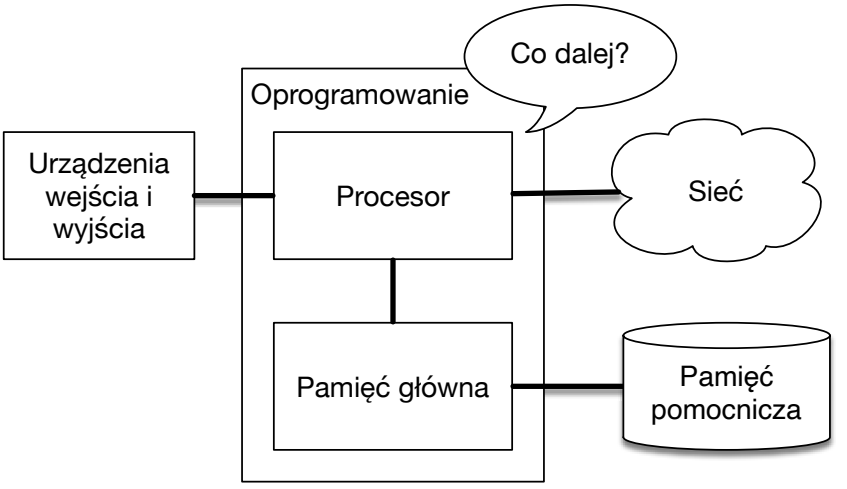
Ale jeśli przypomnisz sobie naszą dyskusję na temat architektury sprzętowej, to po wyłączeniu zasilania, wszystko, co jest zapisane na procesorze lub w pamięci głównej, zostanie usunięte. Tak więc do tej pory nasze programy były tylko przejściowymi, zabawowymi ćwiczeniami do nauki Pythona.

W tym rozdziale rozpoczynamy pracę z *pamięcią pomocniczą* (lub plikami). Pamięć pomocnicza nie jest czyszczona po wyłączeniu zasilania. W przypadku pendrive'a, dane, które zapisujemy z naszych programów, mogą zostać usunięte z systemu i przeniesione do innego systemu.

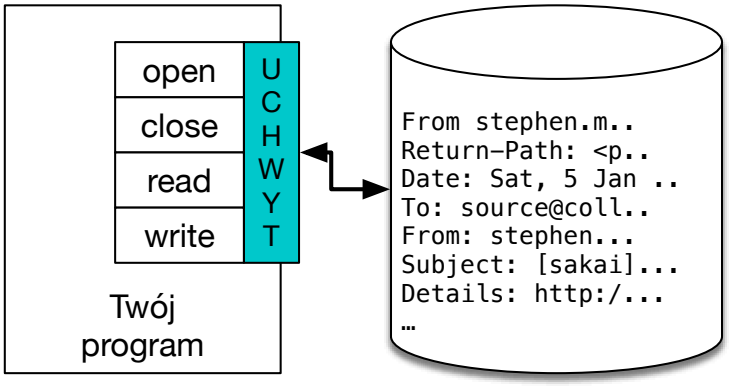
Skupimy się przede wszystkim na odczycie i zapisie plików tekstowych, takich jak te, które tworzymy w edytorze tekstu. Później zobaczymy jak pracować z plikami bazodanowymi, które są plikami binarnymi, przeznaczonymi specjalnie do odczytu i zapisu przez oprogramowanie bazodanowe.

## 7.2. Otwieranie plików

Kiedy chcemy odczytać lub zapisać plik (np. na dysku twardym), musimy najpierw *otworzyć* plik. Otwarcie pliku komunikuje się z Twoim systemem operacyjnym, który wie, gdzie przechowywane są dane dla każdego pliku. Gdy otwierasz plik, prosisz system operacyjny o znalezienie go po nazwie i o upewnienie się, że ten plik istnieje. W poniższym przykładzie otwieramy plik *mbox.txt*, który powinien być przechowywany w tym samym katalogu, w którym znajdujesz się po uruchomieniu Pythona. Możesz pobrać ten plik z pl.py4e.com/code3/mbox.txt.




Rysunek 7.1: Pamięć pomocnicza




Rysunek 7.2: A File Handle

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
```

Jeśli funkcja `open` się powiedzie, system operacyjny zwróci nam *uchwyt pliku*. Uchwyt pliku nie jest rzeczywistymi danymi zawartymi w tym pliku, lecz "uchwytem", który możemy użyć do odczytania danych. Otrzymujesz uchwyt, jeśli żądany plik istnieje i masz odpowiednie uprawnienia do jego odczytania.

Jeśli plik nie istnieje, `open` się nie powiedzie, a Ty nie będziesz mógł uzyskać dostępu do jego zawartości:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'
```

Później użyjemy `try` i `except`, tak aby zgrabniej poradzić sobie z sytuacją, w której próbujemy otworzyć nieistniejący plik.

## 7.3. Pliki tekstowe i linie

Plik tekstowy może być uważany za sekwencję linii, podobnie jak ciąg znaków w Pythona może być uważany za sekwencję liter, liczb i symboli. Na przykład, poniżej znajduje się fragment pliku tekstowego, który rejestruje aktywność pocztową różnych osób w zespole rozwijającym projekt otwartego oprogramowania:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```

Cały plik interakcji mailowych jest dostępny pod adresem:

pl.py4e.com/code3/mbox.txt

a skrócona wersja pliku jest dostępna pod adresem:

pl.py4e.com/code3/mbox-short.txt

Pliki te są w standardowym formacie pliku zawierającego wiele wiadomości pocztowych. Wiersze rozpoczynające się od "From" oddzielają wiadomości, a wiersze rozpoczynające się od "From:" są częścią wiadomości. Więcej informacji na temat formatu mbox można znaleźć na stronie https://pl.wikipedia.org/wiki/Mbox.

Aby podzielić plik na linie, istnieje specjalny znak, który reprezentuje "koniec linii", zwany nomen omen znakiem *końca linii*.

W ciągach znaków Pythona znak *końca linii* reprezentujemy jako lewy ukośnik-n. Nawet jeśli wygląda to jak dwa znaki, to w rzeczywistości jest to pojedynczy znak. W poniższym przykładzie, gdy patrzymy na zmienną "stuff", wpisując ją w interpreterze, pokazuje nam ona `\n` w ciągu znaków, ale gdy używamy `print` do wyświetlenia ciągu znaków, widzimy go rozbitego na dwie linie przez znak końca linii.

```
>>> stuff = 'Witaj\nświecie!'
>>> stuff
'Witaj\nświecie!'
>>> print(stuff)
Witaj
świecie!
>>> stuff = 'X\nY'
>>> print(stuff)
X
Y
```

```
>>> len(stuff)
3
```

Widzimy też, że długość ciągu znaków `X\nY` to *trzy* znaki, ponieważ znak końca linii jest pojedynczym znakiem.

Kiedy więc patrzymy na linie w pliku, musimy sobie *wyobrazić*, że na końcu każdej linii znajduje się specjalny niewidoczny znak zwany znakiem końca linii.

Tak więc znak końca linii dzieli znaki w pliku na linie.

## 7.4. Czytanie plików

Podczas gdy *uchwyt pliku* nie zawiera danych pliku, to całkiem łatwo jest skonstruować pętlę `for` do jego odczytu i zliczyć linie występujące w pliku:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    count = count + 1
print('Liczba linii:', count)

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

Możemy użyć uchwytu pliku jako sekwencji w naszej pętli `for`. Nasza pętla `for` po prostu zlicza linie w pliku i wyświetla tę wartość. Przetłumczanie treści pętli `for` na polski jest z grubsza następujące: "dla każdej linii w pliku reprezentowanej przez uchwyt pliku, dodaj jeden do zmiennej `count`".

Powodem, dla którego funkcja `open` nie odczytuje całego pliku jest to, że plik może być dość duży i zawierać wiele gigabajtów danych. Instrukcja `open` zajmuje zawsze tyle samo czasu, niezależnie od wielkości pliku. Pętla `for` w rzeczywistości powoduje, że dane są odczytywane z pliku.

Gdy plik jest w ten sposób odczytywany przy użyciu pętli `for`, Python zajmuje się podziałem danych z pliku na osobne linie przy użyciu znaku końca linii. Python w każdej iteracji pętli `for` czyta każdą linię od początku aż do wystąpienia znaku końca linii i dołącza go jako ostatni znak w zmiennej `line`.

Ponieważ pętla `for` odczytuje z danych jeden wiersz na raz, może ona efektywnie odczytywać i zliczać wiersze w bardzo dużych plikach, bez obawy, że w pamięci głównej skończy się miejsce na dane. Powyższy program przy użyciu bardzo małej ilości pamięci może zliczać linie w plikach o dowolnym rozmiarze, ponieważ każda linia jest odczytywana, zliczana, a następnie porzucana.

Jeśli wiesz, że plik jest stosunkowo mały w porównaniu z rozmiarem Twojej pamięci głównej, możesz odczytać cały plik za jednym zamachem, używając metody `read` na uchwycie pliku.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
```

```
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```

W powyższym przykładzie, cała zawartość (wszystkie 94 626 znaków) pliku *mbox-short.txt* jest wczytywana bezpośrednio do zmiennej `inp`. Do wyświetlenia pierwszych 20 znaków danych z ciągu znaków przechowywanych w `inp` używamy operacji wycinana podciągów.

Gdy plik zostanie odczytany w ten sposób, to wszystkie znaki, łącznie ze wszystkimi liniami i znakami końca linii, są jednym dużym ciągiem znaków znajdującym się w zmiennej `inp`. Dobrym pomysłem jest przechowywanie wyniku funkcji `read` jako zmiennej, ponieważ każde wywołanie `read` wyczerpuje zasób:

```
>>> fhand = open('mbox-short.txt')
>>> print(len(fhand.read()))
94626
>>> print(len(fhand.read()))
0
```

Pamiętaj, że ta forma funkcji `open` powinna być używana tylko wtedy, gdy dane pliku zmieszczą się spokojnie w pamięci głównej komputera. Jeśli plik jest zbyt duży, by zmieścić się w pamięci głównej, powinieneś napisać swój program tak, aby odczytać plik w kawałkach używając pętli `for` lub `while`.

## 7.5. Przeszukiwanie pliku

Kiedy przeszukujesz dane w pliku, bardzo powszechnym wzorcem postępowania jest odczytywanie danych pliku, ignorując przy tym większość linii i przetwarzając tylko te, które spełniają określony warunek - patrząc na to ogólniej, nazywamy to *schematem filtrowania*. Możemy połączyć przebieg odczytywania pliku z metodami ciągów znaków, tak aby zbudować proste mechanizmy wyszukiwania.

Na przykład, jeśli chcielibyśmy odczytać plik i wypisać tylko te linie, które rozpoczęły się od "From:", moglibyśmy użyć metody ciągów znaków *startswith*, tak aby wybrać tylko te linie, które mają pożądany początek:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    if line.startswith('From:'):
        print(line)

# Kod źródłowy: https://pl.py4e.com/code3/search1.py
```

Gdy powyższy program zostanie uruchomiony, otrzymamy następujący wynik:

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

Wynik wygląda świetnie, ponieważ jedyne linie, które widzimy, to te, które zaczynają się od "From:", ale dlaczego widzimy dodatkowe puste linie? Jest to spowodowane wspomnianym wcześniej niewidocznym *znakiem końca linii*. Każda z linii kończy się wspomnianym znakiem, więc wyrażenie `print` wypisuje ciąg znaków zapisany w zmiennej *line*, który zawiera znak końca linii, a następnie `print` dodaje *kolejny* znak końca linii, co daje efekt podwójnego odstępu, który widzimy.

Moglibyśmy użyć wycinka linii, tak by wypisać wszystkie znaki oprócz ostatniego, ale prostszym podejściem jest użycie metody *rstrip*, która w następujący sposób usuwa białą spację z prawej strony ciągu znaków:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:'):
        print(line)

# Kod źródłowy: https://pl.py4e.com/code3/search2.py
```

Po uruchomieniu tego programu otrzymamy następujący wynik:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

W miarę jak Twoje programy do przetwarzania plików stają się coraz bardziej skomplikowane, możesz chcieć uporządkować swoje pętle wyszukiwania za pomocą `continue`. Podstawową ideą pętli wyszukiwania jest to, że szukasz "interesujących" linii i skutecznie pomijasz "nieciekawe" linie. A potem, gdy znajdziemy interesującą linię, coś z nią robimy.

Pętlę możemy uporządkować tak, aby w następujący sposób trzymała się schematu pomijania "nieciekawych" linii:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
```

```
    # Pomiń 'nieciekawe' linie
    if not line.startswith('From:'):
        continue
    # Przetwarzaj 'interesujące' linie
    print(line)

# Kod źródłowy: https://pl.py4e.com/code3/search3.py
```

Wynik programu jest taki sam. Mówiąc po ludzku, "nieciekawe" są te linie, które nie rozpoczynają się od "From:", więc pomijamy je używając `continue`. W przypadku "interesujących" linii (tzn. tych, które zaczynają się od "From:") wykonujemy przetwarzanie danych, czyli w wypisujemy ich zawartość na ekran.

Możemy użyć metody ciągów znaków `find`, tak aby zasymulować wyszukiwanie w edytorze tekstu, który znajduje linie, w których poszukiwany tekst znajduje się gdziekolwiek w linii. Ponieważ `find` szuka wystąpienia ciągu znaków w innym ciągu i albo zwraca jego pozycję, albo -1 jeśli nie zostanie on znaleziony, możemy napisać następującą pętlę, tak by pokazać te linie, które zawierają ciąg znaków "@uct.ac.za" (tzn. pochodzą z Uniwersytetu w Kapsztadzie w RPA):

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1: continue
    print(line)

# Kod źródłowy: https://pl.py4e.com/code3/search4.py
```

Otrzymujemy następujący wynik:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

Używamy tutaj skróconej formy instrukcji `if`, gdzie umieściliśmy `continue` w tej samej linii co `if`. Ta skrócona forma `if` funkcjonuje tak samo, jak gdyby `continue` znajdowało się w następnej linii i było wcięte.

## 7.6. Pozwolenie użytkownikowi na wybranie nazwy pliku

Naprawdę nie chcemy musieć edytować naszego kodu Pythona za każdym razem, gdy chcemy przetwarzać inny plik. Bardziej użyteczne byłoby poproszenie użytkownika

o wprowadzenie nazwy pliky za każdym razem, gdy program zostanie uruchomiony, tak aby mógł on korzystać z naszego programu na różnych plikach, bez konieczności zmiany kodu Pythona.

Jest to dość proste do zrobienia poprzez odczytanie nazwy pliku od użytkownika za pomocą `input` w następujący sposób:

```
fname = input('Podaj nazwę pliku: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('Mamy', count, 'linii z tematem wiadomości w pliku', fname)

# Kod źródłowy: https://pl.py4e.com/code3/search6.py
```

Odczytujemy nazwę pliku wprowadzoną przez użytkownika, umieszczamy ją w zmiennej o nazwie `fname` i otwieramy ten plik. Teraz możemy uruchomić program wielokrotnie na różnych plikach.

```
python search6.py
Podaj nazwę pliku: mbox.txt
Mamy 1797 linii z tematem wiadomości w pliku mbox.txt

python search6.py
Podaj nazwę pliku: mbox-short.txt
Mamy 27 linii z tematem wiadomości w pliku mbox-short.txt
```

Zanim przejdziemy do następnej sekcji, spójrzmy na powyższy program i zadajmy sobie pytanie: "Co tu może się nie udać?" lub "Co mógłby zrobić nasz miły użytkownik żeby nasz mały program zakończył się błędem z komunikatem traceback, co sprawiłoby, że wyglądalibyśmy kiepsko w oczach naszych użytkowników?".

## 7.7. Używanie `try`, `except` i `open`

Mówiłem Ci, żebyś nie podglądał. To Twoja ostatnia szansa.

Co jeśli nasz użytkownik wpisze coś, co nie jest nazwą pliku?

```
python search6.py
Podaj nazwę pliku: missing.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'

python search6.py
Podaj nazwę pliku: szurum-burum
Traceback (most recent call last):
```

```
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'szurum-burum'
```

Nie śmiej się. Użytkownicy w końcu zrobią wszystko, co w ich mocy, by wysypać Twoje programy, albo celowo, albo w złych zamiarach. W rzeczywistości ważną częścią każdego zespołu zajmującego się tworzeniem oprogramowania jest osoba lub grupa o nazwie *Quality Assurance* (w skrócie QA; zespół *zapewniania jakości*), której zadaniem jest robienie jak najbardziej szalonych rzeczy w celu wysypania oprogramowania stworzonego przez programistę.

Zespół QA jest odpowiedzialny za znalezienie wad w programach, zanim dostarczymy je użytkownikom końcowym, którzy mogą je zakupić lub zapłacić nam za napisanie programu. Tak więc zespół QA jest najlepszym przyjacielem programisty.

Więc teraz, gdy widzimy wadę w programie, możemy ją elegancko naprawić, używając struktury `try/except`. Musimy przyjąć, że `open` może się nie powieść, więc dodamy kod naprawczy gdy `open` się zakończy się niepowodzeniem:

```
fname = input('Podaj nazwę pliku: ')
try:
    fhand = open(fname)
except:
    print('Nie można otworzyć pliku:', fname)
    exit()
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('Mamy', count, 'linii z tematem wiadomości w pliku', fname)

# Kod źródłowy: https://pl.py4e.com/code3/search7.py
```

Funkcja `exit` kończy program. Jest to funkcja, którą wywołujemy, ale która nigdy nie wraca do miejsca wywołania. Teraz, gdy nasz użytkownik (lub zespół QA) wpisze jakieś głupie lub złe nazwy plików, "łapiemy" je i z elegancko przywracamy normalne działanie:

```
python search7.py
Podaj nazwę pliku: mbox.txt
Mamy 1797 linii z tematem wiadomości w pliku mbox.txt

python search7.py
Podaj nazwę pliku: szurum-burum
Nie można otworzyć pliku: szurum-burum
```

Ochrona wywołania `open` jest dobrym przykładem prawidłowego użycia `try` i `except` w programie Pythona. W języku angielskim używamy terminu "pythonowy" (ang. *pythonic*) wtedy, gdy robimy coś "w sposób Pythona". Możemy powiedzieć, że powyższy przykład jest "pythonowym" sposobem na otwarcie pliku.

Kiedy będziesz bardziej biegły w Pythonie, będziesz mógł brać udział w błyskotliwej wymianie zdań z innymi programistami Pythona, tak aby zdecydować, które z dwóch równoważnych rozwiązań problemu jest "bardziej pythonowe". Cel bycia "bardziej pythonowym" oddaje myśl, że programowanie jest częściowo inżynierią i częściowo sztuką. Nie zawsze jesteśmy zainteresowani tym, aby coś po prostu działało; chcemy również, aby nasze rozwiązanie było eleganckie i doceniane jako eleganckie przez równych nam programistów.

## 7.8. Pisanie do plików

Aby móc zapisywać dane do pliku, musisz otworzyć go w trybie "w", podanym jako drugi argument:

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>
```

Jeśli plik już istnieje, otwarcie go w trybie do zapisu usuwa stare dane i tworzy nowy pusty, więc bądź ostrożny! Jeśli plik nie istnieje, tworzony jest nowy plik.

Metoda `write` obiektu uchwytu pliku umieszcza dane w pliku, zwracając liczbę zapisanych znaków. Domyślnym trybem zapisu jest tryb tekstowy, przeznaczony do zapisu (i odczytu) ciągów znaków.

```
>>> line1 = "Oto galaz eukaliptusa,\n"
>>> fout.write(line1)
23
```

Obiekt pliku śledzi gdzie się obecnie znajduje, więc jeśli ponownie wywołasz metodę `write`, to nowe dane zostaną umieszczone na końcu pliku.

Podczas zapisu do pliku musimy być pewni, że ogarniamy znaki końca linii - tutaj wyraźnie wstawiamy znak końca linii, gdy tę linię chcemy zakończyć. Instrukcja `print` automatycznie dołącza znak nowej linii, ale metoda `write` nie dodaje jej automatycznie.

```
>>> line2 = 'ojczyzny naszej emblemat.\n'
>>> fout.write(line2)
26
```

Kiedy skończysz pisać, musisz zamknąć plik, tak aby upewnić się, że ostatni bit danych jest fizycznie zapisany na dysku i aby nie został utracony w przypadku wyłączenia zasilania komputera.

```
>>> fout.close()
```

Możemy zamknąć również pliki, które otwieramy do odczytu, tutaj ale możemy być trochę niechlujni gdy otwieramy tylko kilka plików, ponieważ Python zapewnia, że po zakończeniu programu wszystkie otwarte pliki są zamknięte. Kiedy zapisujemy dane do plików, chcemy je wyraźnie zamknąć, tak aby niczego nie pozostawić przypadkowi.

## 7.9. Kodowanie plików

Przechowując dane tekstowe w plikach należy określić ile bitów potrzeba na zapisanie jednego znaku. Przez ostatnie lata zostało wypracowanych wiele systemów kodowania, np. klasyczny siedmiobitowy ASCII[1], obecnie będący standardem UTF-8[2], popularny w naszej części geograficznej ISO-8859-2[3] (znany również jako Latin-2) oraz CP-1250 (znany również jako Windows-1250[4]). Często informacja o kodowaniu pliku nie jest dostępna, a niektóre programy przetwarzające dane z góry zakładają jakieś kodowanie (np. UTF-8). W takich sytuacjach przetwarzanie pliku wejściowego w niepoprawnym kodowaniu da nam złe wyniki. W konsekwencji, pewną trudność może sprawić obsługa znaków diakrytycznych podczas odczytu i zapisu do pliku.

Dla przykładu użyjemy pliku `polski.txt`, który jest do ściągnięcia z poniższej strony:

pl.py4e.com/code3/polski.txt

Standardowo możemy otworzyć plik bez podawania żadnych argumentów:

```
>>> fhand = open('polski.txt')
```

W zależności od systemu operacyjnego możemy zauważyć, że wydrukowanie informacji o zmiennej `fhand` nieco się różni w sekcji "encoding". Np. w systemie Windows możemy otrzymać:

```
>>> print(fhand)
<_io.TextIOWrapper name='polski.txt' mode='r' encoding='cp1252'>
```

Z kolei w systemie Linux możemy otrzymać:

```
>>> print(fhand)
<_io.TextIOWrapper name='polski.txt' mode='r' encoding='UTF-8'>
```

Jeśli w takiej sytuacji spróbujemy odczytać i wyświetlić zawartość pliku, to w przypadku Windowsa otrzymamy tzw. krzaki:

```
>>> print(fhand.read())
Nikt nie spodziewa siÄ HiszpaÅ„skiej Inkwizycji.
WÅ>rÃ³d naszych metod sÄ... tak rÃ³Å¼ne elementy
jak strach, zaskoczenie... bezwzglÄdna skutecznoÅ>Ä‡
i niemalÅ¼e fanatyczne oddanie papieÅ¼owi...
oraz piÄkne czerwone mundurki.
```

W przypadku Linuxa otrzymamy prawidłowy tekst:

[1] https://pl.wikipedia.org/wiki/ASCII
[2] https://pl.wikipedia.org/wiki/UTF-8
[3] https://pl.wikipedia.org/wiki/ISO_8859-2
[4] https://pl.wikipedia.org/wiki/CP-1250

```
>>> print(fhand.read())
Nikt nie spodziewa się Hiszpańskiej Inkwizycji.
Wśród naszych metod są tak różne elementy
jak strach, zaskoczenie... bezwzględna skuteczność
i niemalże fanatyczne oddanie papieżowi...
oraz piękne czerwone mundurki.
```

Plik `polski.txt` jest zapisany w kodowaniu UTF-8. To, w jakim kodowaniu Python domyślnie otworzy plik, zależy od systemu operacyjnego. Na szczęście funkcja `open` obsługuje parametr o nazwie `encoding`, w którym możemy podać w jakim kodowaniu ma zostać otwarty plik. W przypadku Windowsa moglibyśmy otworzyć plik w następujący sposób:

```
>>> fhand = open('polski.txt', encoding='UTF-8')
>>> print(fhand)
<_io.TextIOWrapper name='polski.txt' mode='r' encoding='UTF-8'>
>>> print(fhand.read())
Nikt nie spodziewa się Hiszpańskiej Inkwizycji.
Wśród naszych metod są tak różne elementy
jak strach, zaskoczenie... bezwzględna skuteczność
i niemalże fanatyczne oddanie papieżowi...
oraz piękne czerwone mundurki.
```

Próbując zapisać tekst "żółć" pod Windowsem w domyślnym kodowaniu, prawdopodobnie trafimy na błąd kodowania:

```
>>> fhand_pl1 = open('proba1.txt', 'w')
>>> print(fhand_pl1)
<_io.TextIOWrapper name='proba1.txt' mode='w' encoding='cp1252'>
>>> fhand_pl1.write("żółć")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python3\lib\encodings\cp1252.py", line 19, in encode
    return
    ↪  codecs.charmap_encode(input,self.errors,encoding_table)[0]
UnicodeEncodeError: 'charmap' codec can't encode character '\u017c'
↪  in position 0: character maps to <undefined>
>>> fhand_pl1.close()
```

Na szczęście możemy użyć parametry `encoding`:

```
>>> fhand_pl2 = open('proba2.txt', 'w', encoding='UTF-8')
>>> fhand_pl2.write("żółć")
4
>>> fhand_pl2.close()
```

Problemy z kodowaniem to szeroki temat i początkującemu programiście może to napsuć wiele krwi. Istnieje wiele sposobów na automatyczne wykrywanie kodowania i konwersji z jednego na drugie. Jednak na potrzeby tego kursu wystarczy, że w

razie wątpliwości otworzysz plik np. w programistycznym edytorze tekstu Atom lub edytorze tekstu Notepad++ - informacja o kodowaniu pliku będzie w prawym dolnym ekranie.

Obecnie zasadniczo przyjętym standardem kodowania plików jest UTF-8.

## 7.10. Debugowanie

Kiedy odczytujesz dane lub zapisujesz dane do plików, możesz mieć problemy z białymi znakami. Tego typu błędy mogą być trudne do debugowania, ponieważ spacje, tabulacje i znaki końca linii są zazwyczaj niewidoczne:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

Wbudowana funkcja `repr` może pomóc. Przyjmuje ona dowolny obiekt jako argument i zwraca jego reprezentację w postaci ciągu znaków. Jeśli obiektem jest właśnie ciąg znaków, to białe znaki są przedstawiane w postaci z ukośnikiem lewym:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

Może to być pomocne przy debugowaniu.

Innym problemem, na który możesz się natknąć, jest to, że różne systemy używają różnych znaków, by oznaczyć koniec linii. Niektóre systemy (np. Linux i macOS) używają znaku końca linii, reprezentowanego przez `\n`. Kolejne używają znaku powrotu karetki, reprezentowanego przez `\r`. Jeszcze inne (np. Windows) używają obu tych znaków, tj. `\r\n`. Jeśli przenosisz pliki między różnymi systemami, te niespójności mogą powodować problemy.[5]

Więcej o tym problemie możesz poczytać na stronie https://pl.wikipedia.org/wiki/Koniec_linii. Dla większości systemów istnieją aplikacje do konwersji z jednego formatu na drugi. Możesz je znaleźć na stronie https://en.wikipedia.org/wiki/Newline w sekcji "Conversion between newline formats". Oczywiście możesz też samodzielnie napisać taki konwerter.

## 7.11. Słowniczek

**krzaki** Niepoprawnie wyświetlane znaki tekstowe.

**łapanie wyjątku** Uniemożliwienie wyjątkowi zakończenia programu przy użyciu instrukcji `try` i `except`.

[5] Jeśli korzystamy np. z edytora Atom, to po otwarciu interesującego nas pliku, informację o użytym sposobie zapisu oznaczenia końca linii możemy znaleźć na dolnej belce w prawym dolnym rogu aplikacji. LF (ang. *line feed*) oznacza `\n`, a CR (ang. *carriage return*) oznacza `\r`.

**plik tekstowy** Sekwencja znaków przechowywana na nośniku danych o trwałej pamięci, np. na dysku twardym.

**pythonowy** Sposób rozwiązania problemu, który działa elegancko w Pythonie. "Użycie try i except jest *pythonowym* sposobem na przywrócenie działania programu w przypadku nieistniejącego pliku".

**Quality Assurance** Osoba lub zespół skoncentrowany na zapewnieniu ogólnej jakości produktu oprogramowania. QA jest często zaangażowana w testowanie produktu i identyfikowanie problemów zanim produkt zostanie wydany.

**znak końca linii** Specjalny znak używany w plikach i ciągach znaków do wskazywania końca linii.

## 7.12. Ćwiczenia

**Ćwiczenie 1: Napisz program odczytujący plik i wypisz zawartość pliku (wiersz po wierszu), ale dużymi literami. Uruchomienie programu będzie wyglądało następująco:**

```
python shout.py
Podaj nazwę pliku: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN  5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
     BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
     SAT, 05 JAN 2008 09:14:16 -0500
```

**Plik możesz pobrać z** pl.py4e.com/code3/mbox-short.txt

**Ćwiczenie 2: Napisz program proszący o podanie nazwy pliku, a następnie przeszukaj plik w celu znalezienia linii podobnej do poniższej:**

```
X-DSPAM-Confidence: 0.8475
```

**Gdy trafisz na linię, która zaczyna się od "X-DSPAM-Confidence:", podziel linię tak, by wyodrębnić z niej liczbę zmiennoprzecinkową. Zliczaj te linie i sumuj wartości oznaczające pewność, że mamy do czynienia ze spamem. Kiedy dotrzesz do końca pliku, wydrukuj średnią wartość pewność co do spamu.**

```
Podaj nazwę pliku: mbox.txt
Średni poziom pewności spamu: 0.894128046745

Podaj nazwę pliku: mbox-short.txt
Średni poziom pewności spamu: 0.750718518519
```

**Przetestuj swój program na plikach *mbox.txt* i *mbox-short.txt*.**

**Ćwiczenie 3: Czasami, gdy programiści się nudzą lub chcą się trochę zabawić, dodają do swojego programu nieszkodliwy tzw. *Easter Egg*[6]**

[6] https://pl.wikipedia.org/wiki/Easter_egg

**(dosł. z ang. *jajko wielkanocne*). Zmodyfikuj swój program, który pyta użytkownika o nazwę pliku tak, by wypisał zabawną wiadomość, gdy użytkownik wpisze w nim nazwę pliku "trele morele". Program powinien zachowywać się normalnie dla wszystkich innych plików, które istnieją i nie istnieją. Oto przykładowe wykonanie programu:**

```
python egg.py
Podaj nazwę pliku: mbox.txt
Mamy 1797 linii z tematem wiadomości w pliku mbox.txt

python egg.py
Podaj nazwę pliku: missing.tyxt
Nie można otworzyć pliku: missing.tyxt

python egg.py
Podaj nazwę pliku: trele morele
TRELE MORELE - co za bzdury!
```

**Nie zachęcamy Cię do umieszczania Easter Eggów w Twoich programach; to tylko ćwiczenie.**

# Rozdział 8

# Listy

## 8.1. Lista jest sekwencją

Podobnie jak ciąg znaków, *lista* jest sekwencją wartości. W ciągu znaku, wartości są znakami, natomiast w liście mogą być dowolnego typu. Wartości występujące w liście są nazywane *elementami*, z rzadka *pozycjami*.

Istnieje kilka sposobów na stworzenie nowej listy; najprostszym jest umieszczenie elementów w nawiasach kwadratowych ("[" i "]"):

```
[10, 20, 30, 40]
['chrupiąca żabka', 'pęcherz barana', 'paw skowronka']
```

Pierwszym przykładem jest lista składająca się z czterech liczb całkowitych. Drugi przykład to lista trzech ciągów znaków. Elementy listy nie muszą być tego samego typu. Poniższa lista zawiera ciąg znaków, liczbę zmiennoprzecinkową, liczbę całkowitą i (uwaga!) inną listę:

```
['spam', 2.0, 5, [10, 20]]
```

Gdy jedna lista znajduje się w innej liście, to mówimy, że jest *zagnieżdżona*.

Lista, która nie zawiera żadnych elementów, nazywana jest listą pustą. Możesz utworzyć taką listę używając pustych nawiasów kwadratowych, tj. `[]`.

Jak pewnie się spodziewałeś, możesz przypisać wartości listy do zmiennych:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

## 8.2. Listy są zmienne

Składnia dostępu do elementów listy jest taka sama jak w przypadku dostępu do znaków w napisach: używa się operatora nawiasów kwadratowych. Wyrażenie wewnątrz nawiasów określa indeks. Pamiętaj, że indeksy zaczynają się od 0:

```
>>> print(cheeses[0])
Cheddar
```

W odróżnieniu od ciągów znaków, listy są zmienne, ponieważ możesz zmienić kolejność elementów na liście lub ponownie przypisać jakiś element do listy. Kiedy operator nawiasów pojawia się po lewej stronie przypisania, identyfikuje on pozycję listy, do której zostanie coś przypisane.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

Pierwszym elementem `numbers`, który kiedyś wynosił 123, jest teraz 5.

Możesz myśleć o liście jako o relacji między indeksami i elementami. Ta relacja nazywana jest *mapowaniem*; każdy indeks "mapuje" do jednego z elementów.

Indeksy listy działają tak samo jak indeksy ciągów znaków:

- Każde wyrażenie będące liczbą całkowitą może być użyte jako indeks.
- Jeśli spróbujesz odczytać lub zapisać pozycję, która nie istnieje, otrzymasz `IndexError`.
- Jeśli indeks ma wartość ujemną, to oblicza się go wstecz od końca listy.

Operator `in` działa również na listach.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

## 8.3. Poruszanie się po listach

Najczęstszym sposobem na przejście po elementach listy jest pętla `for`. Składnia jest taka sama jak dla ciągów znaków:

```
for cheese in cheeses:
    print(cheese)
```

Jest to dobre rozwiązanie jeśli musisz tylko odczytać elementy z listy. Ale jeśli chcesz coś w niej dopisać lub zaktualizować, to potrzebujesz indeksów. W takiej sytuacji najczęściej wykorzystuje się połączenie funkcji `range` i `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

Powyższa pętla przechodzi przez listę i aktualizuje każdy element. `len` zwraca liczbę elementów na liście. `range` zwraca listę indeksów od 0 do $n - 1$, gdzie $n$ jest długością listy. Przy każdej iteracji pętli, `i` kolejno uzyskuje indeks elementu. Instrukcja przypisania w ciele pętli używa `i` do odczytania starej wartości elementu i przypisania nowej wartości.

Pętla `for` przechodząca przez pustą listę nigdy nie wykonuje ciała pętli:

```
empty = []
for x in empty:
    print('Nikt tego nigdy nie zobaczy.')
```

Chociaż lista może zawierać inną listę, to zagnieżdżona lista nadal liczy się jako pojedynczy element. Długość poniższej listy wynosi cztery:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

## 8.4. Operacje na listach

Operator `+` konkatenuje (łączy) listy:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

Z kolei operator `*` powtarza listę określoną liczbę razy:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

W pierwszym przykładzie lista jest powtórzona cztery razy. Drugi przykład powtarza listę trzy razy.

## 8.5. Wycinki list

Operator wycinania działa również na listach:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Jeśli pominiesz pierwszy indeks, wycinek zaczyna się od początku listy. Jeśli pominiesz drugi indeks, wycinek idzie aż do końca. Zatem jeśli pominiesz oba, wycinek jest kopią całej listy.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Ponieważ listy są zmienne, często przydatne jest wykonanie kopii przed wykonaniem innych operacji, które wywracają ich zawartość do góry nogami.

Operator wycinania zastosowany po lewej stronie instrukcji przypisania może aktualizować wiele elementów:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

## 8.6. Metody obiektów będących listami

Python udostępnia metody, które działają na listach. Na przykład, `append` dodaje nowy element na końcu listy:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

`extend` działa podobnie, z tym że przyjmuje jako argument listę elementów do dopisania:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

W powyższym przykładzie lsita `t2` pozostaje niezmieniona.

`sort` układa elementy listy od najmniejszego do największego:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

Większość metod listowych nie zwraca żadnej konkretnej wartości; modyfikują one listę i zwracają `None`. Tego typu modyfikacja nazywana jest *modyfikacją w miejscu*. Jeśli przypadkowo napiszesz `t = t.sort()`, zapewne będziesz zawiedziony zwróconym wynikiem.

## 8.7. Usuwanie elementów

Istnieje kilka sposobów na usunięcie elementów z listy. Jeśli znasz indeks elementu, który chcesz usunąć, to możesz użyć metody `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

`pop` modyfikuje listę i zwraca element, który został usunięty. Jeśli nie podasz indeksu, to usunie on i zwróci ostatni element listy.

Jeśli nie potrzebujesz usuniętej wartości, możesz użyć operatora `del`:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

Jeśli znasz element, który chcesz usunąć (ale nie jego indeks), możesz użyć `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

Wartością zwracaną przez `remove` jest `None`.

Aby usunąć więcej niż jeden element, możesz użyć `del` z indeksami podanymi w postaci wycinka:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

Jak zwykle, wycinek wybiera wszystkie elementy aż do drugiego indeksu, ale wyłączając z tego ostatnią pozycję.

## 8.8. Listy i funkcje

Istnieje szereg wbudowanych funkcji, które mogą być używane na listach i które pozwalają na szybkie przeglądanie listy bez konieczności pisania własnych pętli:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25
```

Funkcja `sum()` działa tylko wtedy, gdy elementy listy są liczbami. Pozostałe funkcje (`max()`, `len()` itp.) działają z listami ciągów znaków i innych typów, które mogą być porównywane.

Moglibyśmy przepisać nasz wcześniejszy program, który teraz obliczyłby średnią z liczb podanych przez użytkownika za pomocą listy.

Poniżej mamy program, który liczy średnią bez listy:

```
total = 0
count = 0
while (True):
    inp = input('Wprowadź liczbę: ')
    if inp == 'gotowe': break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print('Średnia:', average)

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

W powyższym programie mamy zmienne `count` i `total`, tak aby podczas kolejnych próśb o wprowadzenie danych pamiętać ile już wprowadzono liczb oraz by przechowywać sumę bieżącą wprowadzonych liczb.

W alternatywnym podejściu, możemy po prostu zapamiętać każdą wprowadzoną liczbę i pod koniec użyć wbudowanych funkcji do obliczenia sumy i zliczenia elementów.

```
numlist = list()
while (True):
    inp = input('Wprowadź liczbę: ')
    if inp == 'gotowe': break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print('Średnia:', average)

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

Przed rozpoczęciem pętli tworzymy pustą listę, a następnie za każdym razem, gdy otrzymamy nową liczbę, dołączamy ją do listy. Na końcu programu po prostu obliczamy sumę liczb występujących na liście i dzielimy ją przez liczbę elementów listy, tak aby uzyskać średnią.

## 8.9. Listy i ciągi znaków

Ciąg znaków jest sekwencją znaków, a lista jest sekwencją wartości, ale lista znaków nie jest tym samym co ciąg znaków. Aby przekonwertować ciąg znaków na listę znaków, możesz użyć funkcji `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

Ponieważ `list` jest nazwą wbudowanej funkcji, powinieneś unikać używania jej jako nazwy zmiennej. Unikam również litery “l”, ponieważ w wyglądzie jest zbyt podobna do liczby “1”. Właśnie dlatego używam “t”.

Funkcja `list` rozbija ciąg znaków na pojedyncze litery. Jeśli chcesz rozbić ciąg znaków na słowa, to możesz użyć metody `split`:

```
>>> s = 'usycham z tęsknoty za fiordami'
>>> t = s.split()
>>> print(t)
['usycham', 'z', 'tęsknoty', 'za', 'fiordami']
>>> print(t[2])
tęsknoty
```

Gdy już użyjesz `split` aby rozbić ciąg znaków na listę słów, możesz użyć operatora indeksu (nawias kwadratowy) aby przyjrzeć się konkretnemu słowu na liście.

Możesz wywołać `split` z opcjonalnym argumentem zwanym *separatorem*, który określa jakie znaki mają być użyte do określania granic słów. Poniższy przykład używa myślnika jako separatora:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

Funkcja `join` jest odwrotnością `split`. Przyjmuje ona listę ciągów znaków i konkatenuje jej elementy. `join` jest metodą obiektów będących ciągami znaków, więc musisz wywołać ją na separatorze i przekazać listę jako argument:

```
>>> t = ['usycham', 'z', 'tęsknoty', 'za', 'fiordami']
>>> delimiter = ' '
>>> delimiter.join(t)
'usycham z tęsknoty za fiordami'
```

W tym przypadku separator jest znakiem spacji, więc `join` umieszcza spację między słowami. Aby połączyć ciągi znaków bez spacji, możesz użyć pustego ciągu znaków, `""`, jako separator.

## 8.10. Parsowanie linii

Zwykle gdy czytamy z pliku, chcemy zrobić coś innego niż tylko wyświetlić całą linię. Często chcemy znaleźć "interesujące linie", a następnie *rozdzielić* linię, tak aby później znaleźć jakąś interesującą *część* tej linii. Co by było gdybyśmy chcieli wyświetlić dzień tygodnia z tych linii, które zaczynają się od "From"?

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

Metoda `split` jest bardzo skuteczna gdy ma do czynienia z tego rodzaju problemem. Możemy napisać mały program, który szuka linii zaczynających się od "From", rozdzielić te linie przy pomocy `split`, a następnie wyświetlić trzecie słowo występujące w danej linii:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From '): continue
    words = line.split()
    print(words[2])

# Kod źródłowy: https://pl.py4e.com/code3/search5.py
```


Rysunek 8.1: Zmienne i obiekty

Program generuje następujące wyjście:

```
Sat
Fri
Fri
Fri
...
```

Później poznamy coraz bardziej wyrafinowane techniki wybierania linii do dalszego przetwarzania oraz dowiemy się jak je przeanalizować, tak aby znaleźć dokładnie ten fragment informacji, którego szukamy.

## 8.11. Obiekty i wartości

Jeśli wykonamy poniższe instrukcje przypisania:

```
a = 'banan'
b = 'banan'
```

to wiemy, że zarówno `a`, jak i `b` odnoszą się do ciągu znaków, ale nie wiemy, czy odnoszą się one do *tego samego* ciągu znaków. Istnieją dwa możliwe stany:

W jednym przypadku, `a` i `b` odnoszą się do dwóch różnych obiektów, które mają tę samą wartość. W drugim przypadku, odnoszą się one do tego samego obiektu.

Aby sprawdzić, czy dwie zmienne odnoszą się do tego samego obiektu, możesz użyć operatora `is`.

```
>>> a = 'banan'
>>> b = 'banan'
>>> a is b
True
```

W powyższym przykładzie, Python stworzył tylko jeden obiekt ciagu znaków, a obie zmienne `a` i `b` odnoszą się właśnie do tego obiektu.

Ale kiedy tworzysz dwie listy, otrzymujesz dwa obiekty:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

W tym przypadku powiedzielibyśmy, że te dwie listy są *równoważne*, ponieważ mają te same elementy, ale nie są *identyczne*, ponieważ nie są tym samym obiektem. Jeżeli dwa obiekty są identyczne, to są one również równoważne, ale jeżeli są równoważne, to niekoniecznie są identyczne.

Do tej pory używaliśmy zamiennie słów "obiekt" i "wartość", ale powinniśmy mówić, że obiekt ma wartość. Jeśli wykonasz `a = [1,2,3]`, to `a` odnosi się do obiektu listy, którego wartością jest konkretna sekwencja elementów. Jeśli inna lista ma te same elementy, to powiedzielibyśmy, że ma taką samą wartość.

## 8.12. Aliasy

Jeśli `a` odnosi się do obiektu, a Ty przypisujesz `b = a`, to obie zmienne odnoszą się do tego samego obiektu:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

Powiązanie zmiennej z obiektem nazywane jest *referencją*. W powyższym przykładzie istnieją dwie referencje do tego samego obiektu.

Obiekt z więcej niż jedną referencją ma więcej niż jedną nazwę, więc mówimy, że obiekt jest *aliasowany*.

Jeżeli aliasowany obiekt jest zmienny, to zmiany dokonane przy użyciu jednego aliasu wpływają na drugi:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Chociaż takie zachowanie może być użyteczne, jest ono skłonność do generowania błędów. Ogólnie rzecz biorąc, jeśli pracujesz z obiektami zmiennymi, to bezpieczniej jest unikać tworzenia ich aliasów.

W przypadku obiektów niezmiennych, takich jak ciągi znaków, aliasowanie nie jest aż tak dużym problemem. W poniższym przykładzie:

```
a = 'banan'
b = 'banan'
```

prawie nigdy nie ma różnicy czy `a` i `b` odnoszą się do tego samego ciągu znaków, czy też nie.

## 8.13. Argumenty będące listami

Kiedy przekazujesz listę do funkcji, funkcja ta otrzymuje referencję do listy. Jeżeli funkcja modyfikuje parametr będący listą, to z poziomu wywoływania funkcji zauważymy tę zmianę. Na przykład, `delete_head` usuwa pierwszy element z listy:

```
def delete_head(t):
    del t[0]
```

Oto jak możemy ją użyć:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```

Parametr `t` i zmienna `letters` są aliasami dla tego samego obiektu.

Ważne jest, by odróżniać operacje, które modyfikują listy, od operacji, które tworzą nowe listy. Na przykład, metoda `append` modyfikuje listę, ale operator `+` tworzy nową listę:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
None

>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3]
>>> t2 is t3
False
```

Ta różnica jest istotna gdy piszesz definicję funkcji modyfikującej listy. Na przykład, poniższa funkcja *nie usuwa* nagłówka listy:

```
def bad_delete_head(t):
    t = t[1:]               # ŹLE!
```

Operator wycinania tworzy nową listę, a przypisanie sprawia, że zmienna `t` odnosi się teraz do tej nowej listy, ale żadna z tych operacji nie ma żadnego wpływu na listę, która została przekazana do funkcji jako argument.

Alternatywnie można napisać funkcję, która tworzy i zwraca nową listę. Na przykład, `tail` zwraca wszystko oprócz pierwszego elementu listy:

```
def tail(t):
    return t[1:]
```

Funkcja ta pozostawia pierwotną listę bez zmian. Poniżej pokazano jak można użyć tej funkcji:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

**Ćwiczenie 1: Napisz funkcję o nazwie `chop`, która przyjmuje listę i modyfikuje ją, usuwając pierwszy i ostatni element, a następnie zwraca `None`. Następnie napisz funkcję o nazwie `middle`, która przyjmuje listę i zwraca nową listę, która zawiera wszystkie oprócz pierwszego i ostatniego elementu.**

## 8.14. Debugowanie

Nieostrożne korzystanie z list (i innych zmiennych obiektów) może prowadzić do długich godzin debugowania. Oto kilka typowych pułapek wraz ze sposobami na ich uniknięcie:

1. Nie zapominaj, że większość metod listowych modyfikuje argument i zwraca `None`. Jest to odmienne zachowanie w stosunku do tego, co widzieliśmy w metodach obiektów będących ciągami znaków, które zwracają nowy ciąg znaków, a oryginał zostawiają w spokoju.

   Jeśli jesteś przyzwyczajony do pisania takiego kodu dla ciągów znaków jak ten:

   ```
   word = word.strip()
   ```

   to kuszące jest pisanie kodu dla list w poniższy sposób:

   ```
   t = t.sort()              # ŹLE!
   ```

   Metoda `sort` zwraca `None`, więc następna operacja, którą wykonasz z `t`, może się nie udać.

   Przed użyciem metod i operatorów dla list, powinieneś dokładnie przeczytać dokumentację, a następnie przetestować ją w trybie interaktywnym. Metody i operatory, które listy współdzielą z innymi sekwencjami (np. ciągami znaków) są udokumentowane pod adresem:

   docs.python.org/library/stdtypes.html#common-sequence-operations

   Metody i operatory, którzy stosuje się tylko do zmiennych sekwencji, są udokumentowane na stronie internetowej:

   docs.python.org/library/stdtypes.html#mutable-sequence-types

2. Wybierz idiom i trzymaj się go.

   Częścią problemu z listami jest to, że jest zbyt wiele sposobów na wykonanie jakiejś rzeczy. Na przykład, aby usunąć element z listy, możesz użyć `pop`, `remove`, `del`, a nawet użyć przypisania poprzez operację wycinania.

Aby dodać element, możesz użyć metody `append` lub operatora `+`. Ale nie zapomnij, że poniższe sposoby są właściwe:

```
t.append(x)
t = t + [x]
```

Ale te są niepoprawne:

```
t.append([x])          # ŹLE!
t = t.append(x)        # ŹLE!
t + [x]                # ŹLE!
t = t + x              # ŹLE!
```

Wypróbuj każdy z tych przykładów w trybie interaktywnym, tak aby upewnić się, że rozumiesz, co one robią. Zauważ, że tylko ostatni z nich powoduje błąd w czasie wykonania; pozostałe trzy są dozwolone, ale robią coś niepoprawnego.

3. Rób kopie by unikać aliasowania.

   Jeśli chcesz użyć metody takiej jak `sort`, która modyfikuje argument, ale musisz także zachować oryginalną listę, to możesz uprzednio zrobić kopię tej listy.

   ```
   orig = t[:]
   t.sort()
   ```

   W powyższym przykładzie możesz również użyć wbudowanej funkcji `sorted`, która zwraca nową, posortowaną listę, a oryginał pozostawia nienaruszony. Jednak w tym przypadku powinieneś unikać używania nazwy `sorted` jako nazwy dla zmiennej!

4. Listy, `split` i pliki

   Kiedy odczytujemy i analizujemy pliki, jest wiele okazji by trafić na takie dane wejściowe, które mogą wysypać nasz program, więc dobrym pomysłem jest powrót do wzorca *strażnika* wtedy, gdy przychodzi nam do pisania programów, które czytają z pliku i szukają "igły w stogu siana".

   Wróćmy do naszego programu, który szuka dnia tygodnia w wierszach naszego pliku:

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

   Ponieważ rozbijamy linijkę na słowa, możemy zrezygnować z użycia `startswith` i po prostu spojrzeć na pierwsze słowo linii, tak aby określić, czy w ogóle jesteśmy zainteresowani tym wierszem. Możemy użyć `continue` do pominięcia linii, które nie mają słowa "From" jako pierwszego słowa, tak jak podano poniżej:

   ```
   fhand = open('mbox-short.txt')
   for line in fhand:
       words = line.split()
       if words[0] != 'From' : continue
       print(words[2])
   ```

Wygląda to znacznie prościej i nawet nie musimy używać `rstrip` do usunięcia znaku końca linii. Ale czy to jest faktycznie lepsze?

```
python search8.py
Sat
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

Niby działa i widzimy dzień tygodnia dla pierwszej linii (Sat), ale potem program się wysypuje i widzimy błąd razem ze zrzutem z mechanizmu traceback. Co poszło nie tak? Jakie pomyłki w danych spowodowały, że nasz elegancki, pomysłowy i bardzo pythonowy program zawiódł?

Mógłbyś się na ten kod długo gapić i zachodzić w głowę lub poprosić kogoś o pomoc, ale szybsze i mądrzejsze podejście polega na dodaniu instrukcji `print`. Najlepszym miejscem na dodanie wyświetlenia danych jest tuż przed wierszem, w którym program się wysypał i wwyświetlenie tych danych, które wydają się być przyczyną niepowodzenia.

Dzięki takiemu podejściu do problemu możemy wygenerować wiele linii na wyjściu, ale przynajmniej od razu będziemy mieli jakąś wskazówkę co do problemu. Tak więc dodajemy wyświetlenie zmiennej `words` tuż przed piątym wierszem. Dodajemy nawet przedrostek "Debug:" do linii, dzięki czemu możemy oddzielić nasze zwykłe wyjście od wyjścia związanego z debugowaniem.

```
for line in fhand:
    words = line.split()
    print('Debug:', words)
    if words[0] != 'From' : continue
    print(words[2])
```

Kiedy uruchomimy program, przez ekran przewinie się duża ilość danych wyjściowych, ale na końcu zobaczymy nasze debugowane dane wyjściowe i zrzut z mechanizmu traceback, więc wiemy, co się wydarzyło tuż przed pojawieniem się błędu.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

Dla każdego wiersza wypisujemy listę słów, które otrzymujemy z metody `split` podczas dzielenia linii na słowa. Gdy program się wysypie, lista słów jest pusta `[]`. Jeśli otworzymy plik w edytorze tekstu i spojrzymy na niego, to zobaczymy następujący widok:

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan  5 09:14:16 2008
```

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

Błąd pojawia się wtedy, gdy nasz program natknie się na pustą linię! Oczywiście w pustym wierszu znajduje się "zero słów". Dlaczego nie pomyśleliśmy o tym, gdy pisaliśmy kod? Kiedy nasz kod szuka pierwszego słowa (`word[0]`) by sprawdzić czy pasuje do "From", otrzymujemy błąd "index out of range".

Jest to oczywiście idealne miejsce na dodanie kodu ze *strażnikiem*, tak aby uniknąć sprawdzania pierwszego słowa jeśli go nie ma. Istnieje wiele sposobów na ochronę takiego kodu; zdecydujemy się sprawdzić liczbę słów, które mamy, zanim spojrzymy na pierwsze słowo:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    words = line.split()
    # print('Debug:', words)
    if len(words) == 0 : continue
    if words[0] != 'From' : continue
    print(words[2])
```

Najpierw zakomentowaliśmy instrukcję o debugowaniu zamiast ją od razu usuwać, na wypadek gdyby nasza modyfikacja zawiodła i musieliśmy debugować kod ponownie. Następnie dodaliśmy instrukcję strażnika, który sprawdza, czy mamy zero słów, a jeśli tak, to używamy `continue` by przejść do następnej linii w pliku.

Możemy myśleć o tych dwóch instrukcjach `continue` jako o pomocy w filtrowaniu zestawu linii, które są dla nas "interesujące" i które chcemy jeszcze trochę przetworzyć. Linia, która nie ma słów jest dla nas "nieciekawa", więc przechodzimy do następnej linii. Linia, która nie ma "From" jako pierwszego słowa, jest dla nas nieciekawa, więc ją pomijamy.

Zmodyfikowany działa z powodzeniem, więc być może jest poprawny. Nasza instrukcja strażnika zapewnia, że `word[0]` nigdy nie zawiedzie, ale może to nie być wystarczające. Podczas programowania musimy zawsze myśleć: "Co może się nie udać?".

**Ćwiczenie 2: Dowiedz się, która linia powyższego programu nie jest jeszcze właściwie zabezpieczona. Zobacz, czy potrafisz skonstruować taki plik tekstowy, który spowoduje awarię programu, a następnie zmodyfikować program tak, by linia była odpowiednio zabezpieczona. Przetestuj poprawiony program by upewnić się, że obsługuje Twój nowy plik tekstowy.**

**Ćwiczenie 3: Napisz ponownie kod ze strażnikiem z powyższego przykładu, ale bez dwóch instrukcji `if`. Zamiast tego użyj pojedynczej instrukcji `if` ze złożonym wyrażeniem logicznym używającym operatora logicznego.**

## 8.15. Słowniczek

**aliasowanie** Okoliczność, w której dwie lub więcej zmiennych odnosi się do tego samego obiektu.
**element** Jedna z wartości na liście (lub innej sekwencji); nazywana także pozycją.

**identiczny** Będący tym samym obiektem (co oznacza też równoważność).
**indeks** Wartość będąca liczbą całkowitą, która wskazuje element na liście.
**lista** Sekwencja wartości.
**lista zagnieżdżona** Lista, która jest elementem innej listy.
**obiekt** Coś, do czego może odnosić się zmienna. Obiekt ma typ i wartość.
**przejście po liście** Sekwencyjny dostęp do każdego elementu listy.
**referencja** Związek między zmienną a jej wartością.
**równoważny** Posiadający tę samą wartość.
**separator** Znak lub ciąg znaków używany do wskazania, gdzie ciąg znaków powinien zostać podzielony.

## 8.16. Ćwiczenia

**Ćwiczenie 4: Pobierz kopię pliku pl.py4e.com/code3/romeo.txt. Napisz program, który otwiera plik *romeo.txt* i czyta go linia po linii. Dla każdej linii podziel ją na listę słów za pomocą funkcji `split`. Dla każdego słowa sprawdź, czy znajduje się ono już na liście. Jeśli słowa nie ma na liście, to dodaj je do listy. Gdy program zakończy pracę, posortuj i wypisz wynikowe słowa w kolejności alfabetycznej.**

```
Podaj nazwę pliku: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

**Ćwiczenie 5: Napisz program, który czyta dane ze skrzynki pocztowej, a gdy znajdzie wiersz zaczynający się od "From", niech podzieli go na słowa, korzystając z funkcji `split`. Interesuje nas, kto wysłał wiadomość - informacja ta jest drugim słowem w wierszu zawierającym "From".**

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

**Musisz przeparsować linię zawierjącą "From" i wyświetlić drugie słowo dla każdej takiej linii. Następnie musisz zliczyć liczbę wierszy "From" (nie "From:") i na końcu wyświetlić tę liczbę. Poniżej znajduje się przykładowe poprawne wyjście (część linii wyjścia została usunięta):

```
python fromcount.py
Podaj nazwę pliku: mbox-short.txt
stephen.marquard@uct.ac.za
```

```
louis@media.berkeley.edu
zqian@umich.edu

[...część linii wyjścia usunięta...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
Mamy 27 linii w pliku, w których From jest pierwszym wyrazem
```

**Ćwiczenie 6: Napisz ponownie program, który poprosi użytkownika o listę liczbę i wypisze na końcu największą i najmniejszą z nich wtedy, gdy użytkownik wpisze "gotowe". Napisz program w ten sposób, że zapisze on wprowadzone przez użytkownika liczby na liście i użyje funkcji `max()` oraz `min()` do znalezienia największej i najmniejszej liczby po zakończeniu pętli.**

```
Wprowadź liczbę: 6
Wprowadź liczbę: 2
Wprowadź liczbę: 9
Wprowadź liczbę: 3
Wprowadź liczbę: 5
Wprowadź liczbę: gotowe
Największa: 9.0
Najmniejsza: 2.0
```

# Rozdział 9

# Słowniki

*Słownik* jest podobny do listy, ale bardziej ogólny. W liście indeksy muszą być liczbami całkowitymi; w słowniku indeksy mogą być (prawie) dowolnego typu.

Możesz myśleć o słowniku jako o mapowaniu pomiędzy zestawem indeksów (które są nazywane *kluczami*) a zestawem wartości. Każdy klucz mapuje się do jakiejś wartości. Skojarzenie klucza i wartości nazywane jest parą *klucz-wartość* lub czasem *elementem*.

Jako przykład zbudujemy słownik, który mapuje słowa angielskie na hiszpańskie, więc klucze i wartości będą ciągami znaków.

Funkcja `dict` tworzy nowy słownik bez żadnych elementów. Ponieważ `dict` jest nazwą wbudowanej funkcji, powinieneś unikać używania jej jako nazwy zmiennej.

```
>>> eng2sp = dict()
>>> print(eng2sp)
{}
```

Nawiasy klamrowe, `{}`, reprezentują pusty słownik. Aby dodać elementy do słownika, możesz użyć nawiasów kwadratowych:

```
>>> eng2sp['one'] = 'uno'
```

Powyższa linia tworzy element, który mapuje klucz `'one'` do wartości "uno". Jeśli ponownie wyświetlimy słownik, to zobaczymy parę klucz-wartość z dwukropkiem pomiędzy kluczem a wartością:

```
>>> print(eng2sp)
{'one': 'uno'}
```

Widoczny powyżej format wyjściowy jest również formatem wejściowym. Na przykład, możesz utworzyć nowy słownik z trzema elementami. Ale jeśli wydrukujesz `eng2sp`, możesz być zaskoczony:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> print(eng2sp)
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

Kolejność par klucz-wartość nie jest taka sama. W rzeczywistości, jeśli wpiszesz ten sam przykład na swoim komputerze, możesz uzyskać inny wynik. Ogólnie rzecz biorąc, kolejność pozycji w słowniku jest nieprzewidywalna.

Jednak nie jest to problem, ponieważ elementy słownika nigdy nie są indeksowane za pomocą liczb całkowitych. Zamiast tego używasz kluczy do szukania odpowiadających im wartości:

```
>>> print(eng2sp['two'])
'dos'
```

Klucz `'two'` zawsze mapuje się do wartości "dos", więc kolejność elementów nie ma znaczenia.

Jeśli klucza nie ma w słowniku, otrzymasz wyjątek:

```
>>> print(eng2sp['four'])
KeyError: 'four'
```

Funkcja `len` działa również na słownikach i zwraca liczbę par klucz-wartość:

```
>>> len(eng2sp)
3
```

Operator `in` też działa na słownikach i informuje Cię, czy coś pojawia się w słowniku jako *klucz* (wystąpienie jako wartość nie jest wystarczające).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

Aby sprawdzić, czy coś pojawia się w słowniku jako wartość, możesz użyć metody `values`, która zwraca wartości jako typ, który może być skonwertowany na listę, a następnie użyć operatora `in`:

```
>>> vals = list(eng2sp.values())
>>> 'uno' in vals
True
```

Operator `in` używa różnych algorytmów dla list i słowników. W przypadku list, używa algorytmu wyszukiwania liniowego. Gdy lista staje się dłuższa, czas wyszukiwania wydłuża się bezpośrednio proporcjonalnie do jej długości. W przypadku słowników Python używa algorytmu zwanego *tablicą mieszającą* lub *tablicą z haszowaniem*, który ma niezwykłą właściwość: operator `in` zajmuje mniej więcej tyle

samo czasu, niezależnie od tego, ile pozycji jest w słowniku. Nie będę tłumaczył, dlaczego funkcje haszujące są tak magiczne, ale możesz przeczytać o tym więcej na stronie pl.wikipedia.org/wiki/Tablica_mieszająca.

**Ćwiczenie 1: Pobierz kopię pliku** pl.py4e.com/code3/words.txt

**Napisz program, który odczytuje słowa z *words.txt* i przechowuje je jako klucze w słowniku. Nie ma znaczenia, jakie będą wartości w słowniku. Następnie możesz użyć operatora `in` jako szybkiego sposobu na sprawdzenie, czy dany wyraz znajduje się w słowniku.**

## 9.1. Słownik jako zbiór liczników

Załóżmy, że otrzymałeś ciąg znaków zawierający angielski wyraz i chcesz policzyć ile razy każda litera się w nim pojawiła. Możesz to zrobić na kilka sposobów:

1. Mógłbyś utworzyć 26 zmiennych, po jednej dla każdej litery alfabetu. Następnie mógłbyś przejść po napisie i dla każdego znaku zwiększyć odpowiedni licznik, prawdopodobnie używając połączonych wyrażeń warunkowych.
2. Mógłbyś utworzyć listę z 26 elementami. Następnie mógłbyś przekonwertować każdy znak na liczbę (używając wbudowanej funkcji `ord`), użyć liczby jako indeksu do listy i zwiększyć odpowiedni licznik.
3. Mógłbyś utworzyć słownik, w którym znaki byłyby kluczami, a liczniki odpowiednimi wartościami. Za pierwszym razem, gdy natrafisz na znak, dodasz element do słownika. Następnie zwiększałbyś wartość istniejącej wartości.

Każda z tych opcji wykonuje to samo obliczenie, ale każda z nich implementuje to w inny sposób.

*Implementacja* to sposób przeprowadzenia obliczeń; niektóre implementacje są lepsze od innych. Na przykład, zaletą implementacji poprzez słownik jest to, że nie musimy z góry wiedzieć, które litery pojawiają się w ciągu znaków i musimy tylko zrobić miejsce na te, które faktycznie się pojawiają.

Oto jak może wyglądać kod:

```
word = 'brontosaurus'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
print(d)
```

W praktyce obliczamy *histogram*, który jest statystycznym terminem określającym zestaw liczników (lub częstości).

Pętla `for` przechodzi po ciągu znaków. Za każdym razem gdy przechodzimy przez pętlę, jeśli znak zawarty w `c` nie występuje w słowniku, tworzymy nową pozycję z kluczem `c` i wartością początkową 1 (ponieważ widzieliśmy tę literę raz). Jeśli `c` znajduje się już w słowniku, zwiększamy wartość `d[c]`.

Oto wynik programu:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

Histogram wskazuje, że litery "a" i "b" pojawiają się raz; "o" pojawia się dwa razy, i tak dalej.

Słowniki posiadają metodę `get`, która przyjmuje klucz i domyślną wartość. Jeśli klucz pojawia się w słowniku, `get` zwraca odpowiednią wartość; w przeciwnym razie zwraca wartość domyślną. Na przykład:

```
>>> counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print(counts.get('jan', 0))
100
>>> print(counts.get('tim', 0))
0
```

Możemy użyć `get` do bardziej zwięzłego napisania naszej pętli z histogramem. Ponieważ metoda `get` automatycznie zajmuje się przypadkiem gdy danego klucza nie ma w słowniku, możemy zredukować cztery linie do jednej i wyeliminować instrukcję `if`.

```
word = 'brontosaurus'
d = dict()
for c in word:
    d[c] = d.get(c,0) + 1
print(d)
```

Użycie metody `get` do uproszczenia tej pętli zliczania kończy się bardzo często używanym "idiomem" w Pythonie i będziemy go używać wiele razy w pozostałej części książki. Powinieneś poświęcić chwilę na porównanie pętli przy użyciu instrukcji `if` i operatora `in` z pętlą przy użyciu metody `get`. Robią dokładnie to samo, ale drugi sposób jest bardziej zwięzły.

## 9.2. Słowniki i pliki

Jednym z częstych zastosowań słownika jest zliczanie występowania słów w pliku zawierającego jakiś tekst. Zacznijmy od bardzo prostego pliku zawierającego słowa wzięte z książki *Romeo i Julia* (użyjemy tekstu w wersji angielskiej).

Do pierwszego zestawu przykładów wykorzystamy skróconą i uproszczoną wersję tekstu bez interpunkcji. Później będziemy pracować z tekstem zawierającym interpunkcję.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Napiszemy w Pythonie program, które odczyta wiersze pliku, rozbijemy każdy wiersz na listę słów, a następnie przejdziemy w pętli przez każde słowo zawarte w linii i za pomocą słownika policzymy występienie każdego słowa.

Za chwilę zobaczysz, że mamy dwie pętle `for`. Pętla zewnętrzna odczytuje wiersze pliku, a pętla wewnętrzna iteruje przez każde ze słów w danym wierszu. Jest to przykład schematu zwanego *pętlą zagnieżdżoną*, ponieważ jedna z pętli jest pętlą *zewnętrzną*, a druga pętlą *wewnętrzną*.

Pętla wewnętrzna wykonuje wszystkie swoje iteracje za każdym razem gdy pętla zewnętrzna wykonuje jedną iterację. W związku z tym myślimy o pętli wewnętrznej jako iterującej "szybciej", a o pętli zewnętrznej jako iterującej wolniej.

Połączenie dwóch zagnieżdżonych pętli zapewnia, że będziemy zliczać każde słowo w każdym wierszu pliku wejściowego.

```
fname = input('Podaj nazwę pliku: ')
try:
    fhand = open(fname)
except:
    print('Nie można otworzyć pliku:', fname)
    exit()

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Kod źródłowy: https://pl.py4e.com/code3/count1.py
```

W naszej instrukcji `else` używamy zwięzłej alternatywy dla inkrementacji zmiennej. `counts[word] += 1` jest odpowiednikiem `counts[word] = counts[word] + 1`. Każdej z tych metod można użyć do zmiany wartości zmiennej o dowolną pożądaną wielkość. Podobne alternatywy istnieją dla `-=`, `*=` i `/=`.

Kiedy uruchamiamy program, widzimy surowy zrzut wszystkich zliczeń w nieposortowanej kolejności tablicy haszującej. (Plik *romeo.txt* jest dostępny pod adresem pl.py4e.com/code3/romeo.txt).

```
python count1.py
Podaj nazwę pliku: romeo.txt
```

```
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

Trochę niewygodne jest przeglądanie słownika w celu znalezienia najczęściej używanych słów i ich wystąpień, więc musimy dodać trochę więcej kodu Pythona, tak aby uzyskać wynik, które będzie bardziej pomocny.

## 9.3. Pętle i słowniki

Jeśli używasz słownika jako sekwencji w instrukcji `for`, pętla przechodzi wtedy przez klucze słownika. Poniższa pętla wyświetla każdy klucz i odpowiadającą mu wartość:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    print(key, counts[key])
```

Oto jak wygląda wynik działania programu:

```
jan 100
chuck 1
annie 42
```

Tak jak poprzednio, klucze nie są w żadnej konkretnej kolejności.

Możemy użyć tego schematu do zaimplementowania różnych idiomów pętli, które opisaliśmy wcześniej. Na przykład, jeśli chcielibyśmy znaleźć wszystkie wpisy w słowniku o wartości powyżej dziesięciu, moglibyśmy napisać następujący kod:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    if counts[key] > 10 :
        print(key, counts[key])
```

Pętla `for` iteruje przez *klucze* słownika, więc dla każdego klucza musimy użyć operatora indeksu w celu pobrania odpowiedającej mu *wartości*. Poniżej mamy wynik działania programu:

```
jan 100
annie 42
```

Widzimy teraz tylko te elementy, które mają wartość powyżej 10.

Jeżeli chcesz wyświetlić klucze w porządku alfabetycznym, to korzystając z metody `keys` dostępnej w obiektach słownikowych sporządzasz listę kluczy występujących w słowniku, a następnie sortujesz tę listę i przechodzisz w pętli po tej posortowanej liście, przeglądając każdy klucz i wyświetlając pary klucz-wartość w posortowanej kolejności, tak jak pokazano poniżej:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = list(counts.keys())
print(lst)
lst.sort()
for key in lst:
    print(key, counts[key])
```

Oto jak wygląda wynik programu:

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

Najpierw widzisz listę kluczy w nieposortowanej kolejności, które otrzymujemy z metody `keys`. Następnie widzimy uporządkowane pary klucz-wartość, wyświetlane w pętli `for`.

## 9.4. Zaawansowane parsowanie tekstu

W powyższym przykładzie, używając pliku *romeo.txt*, uczyniliśmy ten plik tak prostym, jak to tylko możliwe, usuwając ręcznie całą interpunkcję. Rzeczywisty tekst ma dużo interpunkcji, tak jak to pokazano poniżej.

```
But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
Who is already sick and pale with grief,
```

Ponieważ funkcja `split` szuka spacji i traktuje słowa jako tokeny oddzielone spacjami, traktowalibyśmy słowa "soft!" i "soft" jako *różne* słowa i dla każdego z nich tworzylibyśmy osobny wpis w słowniku.

Również ze względu na to, że plik ma tekst pisany dużymi literami, traktowalibyśmy "who" i "Who" jako różne słowa o różnej liczności.

Oba te problemy możemy rozwiązać za pomocą metod związanych z obiektami ciągów znaków: `lower`, `punctuation` i `translate`. Metoda `translate` jest najbardziej wyrafinowaną z tych metod. Oto dokumentacja dla tej metody:

```
line.translate(str.maketrans(fromstr, tostr, deletestr))
```

*Zmienia znaki w* `fromstr` *na znak na tej samej pozycji w* `tostr` *i usuwa wszystkie znaki, które są w* `deletestr`*. Znaki z* `fromstr` *i* `tostr` *mogą być pustymi ciągami znaków, a parametr* `deletestr` *może zostać pominięty.*

Nie będziemy określać parametru `tostr`, ale użyjemy parametru `deletestr` do usunięcia wszystkich znaków interpunkcyjnych. Pozwolimy nawet Pythonowi wskazać nam listę znaków, które uważa za "interpunkcję":

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Parametry używane przez `translate` były inne w Pythonie 2.0.

Dokonujemy następujących zmian w naszym programie:

```
import string

fname = input('Podaj nazwę pliku: ')
try:
    fhand = open(fname)
except:
    print('Nie można otworzyć pliku:', fname)
    exit()

counts = dict()
for line in fhand:
    line = line.rstrip()
    line = line.translate(line.maketrans('', '',
↪   string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Kod źródłowy: https://pl.py4e.com/code3/count2.py
```

Częścią uczenia się "Sztuki Pythona" lub "Myślenia po pythonowemu" jest uświadomienie sobie, że Python często ma wbudowane rozwiązania dla wielu prostych problemów związanych z analizą danych. Z czasem zobaczysz wystarczająco dużo kodu przykładowego i przeczytasz wystarczająco dużo dokumentacji, tak aby wiedzieć gdzie szukać by sprawdzić czy ktoś już nie napisał czegoś co znacznie ułatwi Ci pracę.

Poniżej znajduje się skrócona wersja wyniku programu:

```
Podaj nazwę pliku: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2,
'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak': 1,
a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,
'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman': 1,
'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

Przeglądanie powyższego wyniku nadal jest niewygodne i choć możemy użyć Pythona by dał nam dokładnie to czego szukamy, ale żeby to zrobić musimy najpierw się dowiedzieć o *krotkach*. Wrócimy do tego przykładu gdy dowiemy się o krotkach.

## 9.5. Debugowanie

Podczas pracy z większymi zbiorami danych, debugowanie poprzez wyświetlanie i ręczne sprawdzanie danych może okazać się niewygodne. Oto kilka sugestii dotyczących debugowania dużych zbiorów danych:

**Stopniowe redukowanie danych wejściowych** Jeśli to możliwe, zmniejsz rozmiar zbioru danych. Na przykład, jeśli program odczytuje plik tekstowy, zacznij od pierwszych 10 linii lub od najmniejszego przykładu, jaki możesz znaleźć. Możesz albo edytować same pliki, albo (lepiej) zmodyfikować program tak, by czytał tylko pierwsze `n` linii.

Jeśli jest jakiś błąd, możesz zredukować `n` do najmniejszej wartości, która generuje błąd, a następnie zwiększać tę wartość stopniowo, w międzyczasie znajdując i poprawiając błędy.

**Sprawdź podsumowania i typy** Zamiast wyświetlać i sprawdzać cały zbiór danych, zastanów się nad wyświetlaniem podsumowań danych: na przykład, liczbę pozycji w słowniku lub sumę liczb.

Częstą przyczyną błędów czasu wykonania jest wartość, która nie jest właściwego typu. Do debugowania tego typu błędów często wystarczy wydrukować typ wartości.

**Napisz mechanizm do samokontroli** Czasami możesz napisać kod do automatycznego sprawdzania błędów. Na przykład, jeśli obliczasz średnią z listy liczb, możesz sprawdzić, czy wynik nie jest większy od największego elementu na liście lub mniejszy od najmniejszego. Nazywa się to "sprawdzaniem poczytalności" (ang. sanity check), ponieważ wykrywa ono wyniki, które są "zupełnie nielogiczne".

Inny rodzaj sprawdzenia porównuje wyniki dwóch różnych obliczeń, tak aby sprawdzić czy są one spójne. Nazywa się to "sprawdzaniem spójności" (ang. consistency check).

**Ładne wyświetlenie wyniku** Czytelne sformatowanie wyników debugowania może ułatwić wykrycie błędu.

Ponownie, czas spędzony na budowaniu dodatkowych rusztowań w Twoim programie może skrócić czas, który spędzasz na debugowaniu.

## 9.6. Słowniczek

**element słownika** Inna nazwa dla pary klucz-wartość.
**funkcja haszująca** Funkcja używana przez tablice mieszające do obliczania lokalizacji dla danego klucza.
**histogram** Zbiór liczników.
**implementacja** Sposób przeprowadzenia obliczeń.
**klucz** Obiekt, który pojawia się w słowniku jako pierwsza część pary klucz-wartość.

**para klucz-wartość** Reprezentacja mapowania z klucza do wartości.

**pętle zagnieżdżone** Kiedy istnieje jedna lub więcej pętli "wewnątrz" innej pętli. Pętla wewnętrzna wykonuje się do końca za każdym razem, gdy pętla zewnętrzna iteruje się raz.

**słownik** Mapowanie z zestawu kluczy do odpowiadających im wartości.

**tablica mieszająca** Algorytm używany do implementacji słowników Pythona; inaczej tablica z haszowaniem.

**wartość** Obiekt, który pojawia się w słowniku jako druga część pary klucz-wartość. Jest to bardziej szczegółowe określenie niż nasze poprzednie użycia słowa "wartość".

**wyszukiwanie** Operacja słownikowa, która dla podanego klucza znajduje odpowiadającą mu wartość.

## 9.7. Ćwiczenia

**Ćwiczenie 2: Napisz program, który skategoryzowałby każdą wiadomość pocztową względem czasu wykonania rewizji w projekcie. Aby to zrobić, poszukaj wierszy rozpoczynający się od "From", a następnie poszukaj trzeciego słowa i zachowaj bieżące zliczenia dla każdego dnia tygodnia. Na koniec programu wyświetl zawartość swojego słownika (kolejność nie ma znaczenia).**

**Przykładowa linia:**

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

**Przykładowe uruchomienie:**

```
python dow.py
Podaj nazwę pliku: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

**Ćwiczenie 3: Napisz program, który odczytuje dane z dziennika skrzynki pocztowej, tworzy histogram przy użyciu słownika by zliczyć ile wiadomości przyszło z każdego adresu e-mail, a pod koniec wyświetla ten słownik.**

```
Podaj nazwę pliku: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

**Ćwiczenie 4: Do powyższego programu dodaj kod, tak by dowiedzieć się kto otrzymał najwięcej e-maili. Po przeczytaniu wszystkich danych i utworzeniu słownika, przeszukaj słownik za pomocą pętli wyszukującej największą wartość (patrz: Rozdział 5. Pętle typu maksimum i minimum), tak by dowiedzieć się kto ma najwięcej wiadomości, a na koniec wyświetl informację o tym ile wiadomości ma dana osoba.**

```
Podaj nazwę pliku: mbox-short.txt
cwen@iupui.edu 5

Podaj nazwę pliku: mbox.txt
zqian@umich.edu 195
```

**Ćwiczenie 5: Napisz program, który zapamiętuje nazwę domeny, z której została wysłana wiadomość (zamiast informacji od kogo pochodziła wiadomość, tzn. całego adresu e-mail). Na koniec programu wyświetl zawartość swojego słownika.**

```
python schoolcount.py
Podaj nazwę pliku: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

# Rozdział 10

# Krotki

## 10.1. Krotki są niezmienne

Krotka[1] jest sekwencją wartości, podobnie jak lista. Wartości zapisane w krotce mogą być dowolnego typu i są indeksowane liczbami całkowitymi. Ważną różnicą jest to, że krotki są *niezmienne*. Krotki są również *porównywalne* i *haszowalne*, więc możemy sortować ich listy i używać krotek jako kluczy w słownikach.

Składniowo, krotka jest oddzieloną przecinkami listą wartości:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Chociaż nie jest to konieczne, krotki często ogranicza się nawiasami, tak aby pomóc nam szybko zidentyfikować krotki podczas przeglądania kodu Pythona:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Aby utworzyć krotkę z pojedynczym elementem (tzw. *singleton*), należy na końcu dołączyć przecinek:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Bez przecinka Python traktuje `('a')` jako wyrażenie z ciągiem znaków w nawiasach, które ewaluuje się do ciągu znaków:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

[1] Ciekawostka: Angielskie tłumaczenie słowa "krotka", czyli *tuple*, pochodzi od nazw nadawanych sekwencjom liczb o różnej długości: *single, double, triple, quadruple, quintuple, sextuple, septuple* itd.

Innym sposobem na skonstruowanie krotki jest wbudowana funkcja `tuple`. Bez żadnego argumentu, tworzy ona pustą krotkę:

```
>>> t = tuple()
>>> print(t)
()
```

Jeśli argumentem jest sekwencja (ciąg znaków, lista lub krotka), to wynikiem wywołania funkcji `tuple` jest krotka z elementami podanej sekwencji:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

Ponieważ słowo `tuple` jest nazwą konstruktora, powinieneś unikać używania go jako nazwy zmiennej.

Większość operatorów związanych z listami działa również na krotkach. Operator nawiasów kwadratowych służy do podawania indeksu (pozycji) elementu:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

Operator wycinania wybiera szereg elementów.

```
>>> print(t[1:3])
('b', 'c')
```

Ale jeśli spróbujesz zmodyfikować jeden z elementów krotki, to otrzymasz błąd:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

Nie możesz modyfikować elementów krotki, ale możesz zastąpić jedną krotkę drugą krotką:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

## 10.2. Porównywanie krotek

Operatory porównania działają z krotkami i innymi sekwencjami. Python zaczyna od porównania pierwszego elementu z każdej sekwencji. Jeśli są one równe, przechodzi do następnego elementu itd., aż znajdzie takie elementy, które się różnią. Kolejne elementy nie są już brane pod uwagę (nawet jeśli ich wartość jest bardzo duża).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

Funkcja `sort` działa w ten sam sposób. Sortuje przede wszystkim według pierwszego elementu, ale w przypadku takich samych wartości, sortuje według drugiego elementu itd.

Powyższa własność prowadzi do schematu zwanego *DSU* oznaczającego

**Dekorowanie** sekwencji poprzez zbudowanie listy krotek z jednym lub więcej kluczami sortowania, które poprzedzają elementy sekwencji,
**Sortowanie** listy krotek przy pomocy funkcji wbudowanej `sort`,
**Usunięcie dekorowania** poprzez wyodrębnienie posortowanych elementów sekwencji.

Na przykład, załóżmy, że posiadasz listę wyrazów i chcesz je posortować od najdłuższego do najkrótszego:

```
txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()
for word in words:
    t.append((len(word), word))

t.sort(reverse=True)

res = list()
for length, word in t:
    res.append(word)

print(res)

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

Pierwsza pętla buduje listę krotek, gdzie każda krotka jest wyrazem poprzedzonym jego długością.

Funkcja `sort` porównuje najpierw pierwsze elementy krotek, czyli długości wyrazów, a drugie elementy rozpatruje tylko w przypadku takich samych długości wyrazów. Argument z podaną nazwą `reverse=True` wskazuje funkcji `sort` by sortować w kolejności malejącej.

Druga pętla przechodzi przez listę krotek i tworzy listę wyrazów według ich malejącej długości. Czteroznakowe wyrazy są posortowane w odwrotnej kolejności alfabetycznej, więc słowo “what” pojawia się przed słowem “soft”.

Wynik działania programu jest następujący:

```
['yonder', 'window', 'breaks', 'light', 'what',
'soft', 'but', 'in']
```

Oczywiście linia ta, gdy zostanie przekształcona w listę Pythona i posortowana w porządku malejącym według długości słów, traci swój poetycki urok.

## 10.3. Krotki i operacja przypisania

Jedną z unikalnych cech syntaktycznych języka Python jest możliwość posiadania krotki po lewej stronie instrukcji przypisania. Pozwala to na przypisanie w tym samym czasie więcej niż jednej zmiennej, w momencie gdy lewa strona jest sekwencją.

W poniższym przykładzie mamy dwuelementową listę (która jest sekwencją) i w jednej instrukcji przypisujemy pierwszy i drugi element sekwencji do zmiennych `x` i `y`.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

To nie jest magia, Python *z grubsza* tłumaczy składnię przypisania krotek na następującą:[2].

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

Stylistycznie rzecz ujmując, gdy używamy krotki po lewej stronie instrukcji przypisania, pomijamy nawiasy, ale poniższa składnia jest równie poprawna:

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
>>>
```

Sprytne zastosowanie operacji przypisania krotki pozwala nam na *zamianę* wartości dwóch zmiennych przy pomocy jednej instrukcji:

[2]Python nie tłumaczy składni dosłownie. Na przykład, jeśli spróbujesz takiej operacji na słownikach, to nie będzie to działać tak, jak można by się tego spodziewać

```
>>> a, b = b, a
```

Obie strony tego wyrażenia są krotkami, ale lewa strona jest krotką zmiennych; prawa strona jest krotką wyrażeń. Każda wartość po prawej stronie jest przypisana do odpowiadającej jej zmiennej po lewej stronie. Wszystkie wyrażenia po prawej stronie są ewaluowane przed każdym z przypisań.

Liczba zmiennych po lewej stronie i liczba wartości po prawej stronie musi być taka sama:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

Ogólnie rzecz biorąc, prawa strona może być dowolną sekwencją (ciągiem, listą lub krotką). Na przykład, aby podzielić adres e-mail na nazwę użytkownika i domenę, można napisać poniższy kod:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

Wartość zwracana przez `split` jest listą z dwoma elementami; pierwszy element jest przypisany do `uname`, a drugi do `domain`.

```
>>> print(uname)
monty
>>> print(domain)
python.org
```

## 10.4. Słowniki i krotki

Słowniki posiadają metodę zwaną `items`, która zwraca listę krotek, gdzie każda krotka jest parą klucz-wartość:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
[('b', 1), ('a', 10), ('c', 22)]
```

Jak mogłeś się spodziewać po słowniku, powyższe pary nie są w żadnej konkretnej kolejności.

Ponieważ jednak lista krotek jest listą, a krotki są porównywalne, to możemy teraz posortować listę krotek. Konwertowanie słownika na listę krotek jest dla nas sposobem na wyciągnięcie zawartości słownika posortowanego po kluczach:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> t
```

```
[('b', 1), ('a', 10), ('c', 22)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

Nowa lista jest posortowana rosnąco w porządku alfabetycznym według kluczy.

## 10.5. Wielokrotne przypisanie w słownikach

Łącząc `items`, przypisanie krotki oraz `for`, możesz zobaczyć ciekawy schemat kodu do przechodzenia po kluczach i wartościach słownika w jednej pętli:

```
for key, val in list(d.items()):
    print(val, key)
```

Powyższa pętla ma dwie *zmienne iteracyjne*, ponieważ `items` zwraca listę krotek i `key, val` jest przypisaniem krotki, które kolejno iteruje przez każdą z par klucz-wartość w słowniku.

W każdej kolejnej iteracji pętli, zarówno `key`, jak i `val` posuwają się naprzód do kolejnej pary klucz-wartość w słowniku (nadal w kolejności haszach).

Wynik tej pętli jest następujący:

```
10 a
22 c
1 b
```

I ponownie, mamy tutaj kolejności kluczy po haszach (tzn. bez konkretnej kolejności).

Jeżeli połączymy te dwie techniki, możemy wypisać zawartość słownika posortowaną po *wartościach* zapisanych w każdej parze klucz-wartość.

Aby to zrobić, najpierw sporządzamy listę krotek, gdzie każda krotka to `(value, key)`. Metoda `items` zwróci nam listę `(key, value)` krotek, ale tym razem chcemy sortować według wartości, a nie klucza. Po skonstruowaniu listy z parami klucz-wartość, możemy łatwo posortować listę w odwrotnej kolejności i wypisać nową, posortowaną listę.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
...     l.append( (val, key) )
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

Ostrożnie konstruując listę krotek tak, aby w każdej z nich wartość była pierwszym elementem, możemy posortować listę krotek i uzyskać zawartość słownika posortowaną po wartościach.

## 10.6. Najczęściej występujące słowa

Wracając do naszego przykładu z tekstem z *Romeo i Julia* (Akt II, Scena II), możemy rozszerzyć nasz program po to by użyć opisanej wyżej techniki do wypisania dziesięciu najczęściej używanych słów w tekście:

```
import string
fhand = open('romeo-full.txt')
counts = dict()
for line in fhand:
    line = line.translate(str.maketrans('', '', string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

# Posortuj słownik po wartościach
lst = list()
for key, val in list(counts.items()):
    lst.append((val, key))

lst.sort(reverse=True)

for key, val in lst[:10]:
    print(key, val)

# Kod źródłowy: https://pl.py4e.com/code3/count3.py
```

Pierwsza część programu, która odczytuje plik i tworzy słownik, który mapuje każde słowo do liczbę jego wystąpień w dokumencie, pozostaje niezmieniona. Ale zamiast po prostu wypisać `counts` i zakończyć program, konstruujemy listę krotek `(val, key)`, a następnie sortujemy tę listę w odwrotnej kolejności.

Ponieważ wartość jest pierwsza, będzie ona użyta do porównań. Jeśli jest więcej niż jedna krotka o tej samej wartości, to operacja sortowania spojrzy na drugi element (klucz), więc krotki, w których wartość jest taka sama, będą posortowane w kolejności alfabetycznej po kluczu.

Na koniec piszemy ładną pętlę `for`, która podczas iteracji wykonuje operację przypisania z fragmentu listy (`lst[:10]`) i wypisuje dziesięć najczęściej występujących słów.

Tak więc teraz wynik programu w końcu wygląda tak jak tego oczekujemy od naszej analizy częstości słów.

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

Fakt, że takie skomplikowane parsowanie i analiza danych może być wykonana za pomocą łatwego do zrozumienia 19-wierszowego programu Pythona, jest jednym z powodów, dla których Python jest dobrym wyborem jako język do eksploracji informacji.

## 10.7. Używanie krotek jako kluczy w słownikach

Ponieważ krotki są *haszowalne*, a listy nie są, to jeśli chcemy utworzyć *złożony* klucz do użycia w słowniku, to musimy użyć krotki jako klucza.

Napotkalibyśmy klucz złożony, gdybyśmy chcieli stworzyć książkę telefoniczną, która mapuje z nazwisko-imię do numerów telefonów. Zakładając, że zdefiniowaliśmy zmienne `last`, `first` i `number`, moglibyśmy napisać instrukcję przypisania do słownika w następujący sposób:

```
directory[last,first] = number
```

Wyrażenie w nawiasach jest krotką. Moglibyśmy użyć przypisania krotki w pętli `for`, tak aby przejść po takim słownik.

```
for last, first in directory:
    print(first, last, directory[last,first])
```

Powyższa pętla przechodzi przez klucze w `directory`, które są krotkami. Przypisuje elementy każdej krotki do `last` i `first`, a następnie wypisuje nazwę i odpowiadający jej numer telefonu.

## 10.8. Sekwencje: ciągi znaków, listy i krotki - O rany!

Skupiłem się na listach krotek, ale prawie wszystkie przykłady w tym rozdziale działają również z listami list, krotkami krotek i krotkami list. Aby uniknąć wyliczania możliwych kombinacji, czasem łatwiej jest mówić o sekwencjach sekwencji.

W wielu kontekstach różne rodzaje sekwencji (ciągów znaków, list i krotek) mogą być używane zamiennie. Więc jak i dlaczego wybierasz jedną z nich zamiast innych?

Zaczynając od tego, co oczywiste, ciągi znaków są bardziej ograniczone niż inne sekwencje, ponieważ ich elementy muszą być znakami. Są one również niezmienne. Jeśli potrzebujesz możliwości zmiany znaków w takim ciągu (w przeciwieństwie do tworzenia nowego ciągu), możesz w zamian użyć listy znaków.

Listy są popularniejsze w użyciu niż krotki, głównie dlatego, że są zmienne. Jednak jest kilka przypadków, w których możesz preferować krotki:

1. W niektórych kontekstach, jak np. instrukcja `return`, stworzenie krotki jest składniowo prostsze od stworzenia listy. W innych przepadkach możesz preferować listę.

2. Jeśli chcesz użyć sekwencji jako klucza w słowniku, musisz użyć typu niezmiennego takiego jak krotka lub ciąg znaków.

3. Jeżeli przekazujesz sekwencję jako argument do funkcji, używanie krotek zmniejsza możliwość nieoczekiwanego działania kodu spowodowanego aliasowaniem.

Ponieważ krotki są niezmienne, nie dostarczają one metod takich jak `sort` i `reverse`, które modyfikują istniejące listy. Jednakże Python dostarcza wbudowane funkcje `sorted` i `reversed`, które przyjmują każdą sekwencję jako parametr i zwracają nową sekwencję z tymi samymi elementami w innej kolejności.

## 10.9. Debugowanie

Listy, słowniki i krotki są ogólnie znane jako *struktury danych*; w tym rozdziale zaczęliśmy zauważać złożone struktury danych, takie jak listy krotek oraz słowniki, które zawierają krotki jako klucze i listy jako wartości. Złożone struktury danych są użyteczne, ale są podatne na to, co nazywam *błędami kształtu*, tzn. błędy spowodowane niewłaściwym typem, rozmiarem lub zawartością struktury danych; być może piszesz też jakiś kod i zapominasz o kształcie swoich danych, przez co nieświadomie umieszczasz błąd w swoim programie. Na przykład, jeśli oczekujesz listy składającej się z jednej liczby całkowitej, a ja daję Ci po prostu zwykłą liczbę całkowitą (która nie jest opakowana listą), to Twój kod nie będzie działać.

## 10.10. Słowniczek

**DSU** Skrót od "dekorowanie-sortowanie-usunięcie dekorowania"; schemat, który polega na budowaniu listy krotek, sortowaniu i wyodrębnianiu części wyniku.

**haszowalny** Typ, który posiada funkcję haszowania. Typy niezmienne, takie jak liczby całkowite, zmiennoprzecinkowe i ciągi znaków, są haszowalne; typy zmienne, takie jak listy i słowniki, nie są haszowalne.

**krotka** Niezmienna sekwencja elementów.

**kształt (struktury danych)** Podsumowanie typu, rozmiaru i zawartości struktury danych.

**porównywalny** Typ, w którym można sprawdzić, czy jedna wartość jest większa, mniejsza lub równa innej wartości tego samego typu. Typy, które są porównywalne, można umieścić na liście i posortować.

**przypisanie do krotki** Przypisanie z sekwencją po prawej stronie i krotką zmiennych po lewej. Prawa strona jest ewaluowana, a następnie jej elementy są przypisywane do zmiennych po lewej stronie.

**struktura danych** Kolekcja powiązanych ze sobą wartości, często zorganizowanych w listy, słowniki, krotki itp.

## 10.11. Ćwiczenia

**Ćwiczenie 1: Zrewiduj swój poprzedni program w następujący sposób: Wczytaj i przetwórz linie zaczynające się od "From" oraz wyciągnij z nich adresy. Policz liczbę wiadomości od każdej osoby za pomocą słownika.**

**Po przeczytaniu wszystkich danych wypisz osobę, która zrobiła najwięcej rewizji, tworząc ze słownika listę krotek (liczba, email). Następnie posortuj listę w odwrotnej kolejności i wypisz osobę, która zrobiła najwięcej rewizji.**

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

Podaj nazwę pliku: mbox-short.txt
cwen@iupui.edu 5

Podaj nazwę pliku: mbox.txt
zqian@umich.edu 195
```

**Ćwiczenie 2: Napisz program, który dla każdej wiadomości zlicza rozkład godzin w ciągu dnia. Możesz wyciągnąć godzinę z wiersza "From", odszukując ciąg znaków związany z czasem, a następnie dzieląc ten ciąg na części za pomocą znaku dwukropka. Kiedy już zgromadzisz wartości dla każdej godziny, wypisz zliczenia, po jednym na wiersz, posortowane według godzin, tak jak pokazano poniżej.**

```
python timeofday.py
Podaj nazwę pliku: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

**Ćwiczenie 3: Napisz program, który wczytuje plik i wypisuje *litery* malejąco po częstości ich występowania. Twój program powinien przekonwertować wszystkie dane wejściowe na małe litery i zliczać tylko litery a-z. Program nie powinien uwzględniać spacji, cyfr, interpunkcji ani niczego innego poza literami a-z. Znajdź próbki tekstów z kilku różnych języków i sprawdź, jak bardzo częstość liter różni się w zależności od języka. Porównaj swoje wyniki z tabelami pod adresem https://en.wikipedia.org/wiki/Letter_frequency.**

# Rozdział 11

# Wyrażenia regularne

Do tej pory odczytywaliśmy dane poprzez pliki, szukaliśmy wzorców i wyodrębnialiśmy różne fragmenty linii, które wydawały nam się interesujące. Używaliśmy metod związanych z ciągami znaków, takich jak `split` i `find`, a także używaliśmy list i wycinania fragmentów napisów do wyodrębnienia części linii.

Zadanie wyszukiwania i wyodrębniania jest tak często wykonywane, że Python ma bardzo wszechstronną i wydajną bibliotekę o nazwie *wyrażenia regularne*, która dość elegancko radzi sobie z tymi zadaniami. Powodem, dla którego nie wprowadziliśmy wcześniej w książce wyrażeń regularnych jest to, że choć mają dużo możliwości i są bardzo przydatne, to niestety są trochę skomplikowane, a przyzwyczajenie się do ich składnia wymaga trochę czasu.

Wyrażenia regularne (w skrócie z ang. *regex*) są prawie własnym, małym językiem programowania do wyszukiwania i parsowania ciągów znaków. W rzeczywistości, o wyrażeniach regularnych napisano już wiele książek. W tym rozdziale zajmiemy się tylko podstawami wyrażeń regularnych. Więcej szczegółów na temat wyrażeń regularnych znajdziesz na stronie:

https://pl.wikipedia.org/wiki/Wyrażenie_regularne

https://docs.python.org/library/re.html

Biblioteka wyrażeń regularnych `re` musi być zaimportowana w Twoim programie, zanim będziesz mógł z niej korzystać. Najprostszym wykorzystaniem biblioteki wyrażeń regularnych jest funkcja `search()`. Poniższy program demonstruje jej proste użycie.

```
# Szukaj linii, które zawierają 'From'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line):
        print(line)

# Kod źródłowy: https://pl.py4e.com/code3/re01.py
```

Otwieramy plik, przechodzimy przez każdą linię i używamy wyrażenia regularnego w `search()` do wypisania tylko tych linii, które zawierają napis "From:". Powyższy program nie wykorzystuje prawdziwej mocy wyrażeń regularnych, ponieważ równie łatwo mogliśmy użyć `line.find()` do uzyskania tego samego wyniku.

Moc wyrażeń regularnych ujawnia się wtedy, gdy dodajemy do ciągu znaków wyszukiwania specjalne znaki, które pozwalają nam dokładniej kontrolować to, które linie pasują do tego napisu. Dodanie tych specjalnych znaków do naszego wyrażenia regularnego pozwala nam na wyrafinowane dopasowanie i ekstrakcję danych przy jednoczesnym zachowaniu bardzo małej ilości kodu.

Na przykład, znak karety (tzw. daszek) jest używany w wyrażeniach regularnych by dopasować "początek" linii. Moglibyśmy w następujący sposób zmienić nasz program na dopasowywanie tylko tych linii, w których "From:" było na początku linii:

```
# Szukaj linii, które zawierają 'From'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line):
        print(line)

# Kod źródłowy: https://pl.py4e.com/code3/re02.py
```

Teraz będziemy dopasowywać tylko te linie, które *zaczynają się od* ciągu znaków "From:". Jest to wciąż bardzo prosty przykład, który mogliśmy zrealizować równoważnie przy pomocy metody `startswith()` dla ciągów znaków. Jednak wspomniany wyżej symbol służy tylko do zwrócenia uwagi, że wyrażenia regularne zawierają specjalne znaki akcji, które dają nam większą kontrolę nad tym, co będzie pasować do wyrażenia regularnego.

## 11.1. Dopasowywanie znaków w wyrażeniach regularnych

Istnieje wiele innych specjalnych znaków, które pozwalają nam budować jeszcze bardziej wyrafinowane wyrażenia regularne. Najczęściej używanym znakiem specjalnym jest kropka, która dopasowuje każdy znak.

W poniższym przykładzie, wyrażenie regularne `F..m:` pasuje do któregokolwiek z ciągów znaków "From:", "Fxxm:", "F12m:" lub "F!@m:", ponieważ kropka w wyrażeniu regularnym oznacza dopasowanie dowolnego znaku.

```
# Wyszukaj linie zaczynające się od 'F',
# po których następują 2 dowolne znaki,
# a następnie po których występuje 'm:'.
import re
hand = open('mbox-short.txt')
```

```
for line in hand:
    line = line.rstrip()
    if re.search('^F..m:', line):
        print(line)

# Kod źródłowy: https://pl.py4e.com/code3/re03.py
```

Takie wyrażenie regularne ma szczególnie dużą moc, gdy przy pomocy `*` lub `+` poszerzy się je o możliwością wskazania, że dany znak może być powtórzony dowolną liczbę razy. Te znaki specjalne oznaczają, że zamiast dopasowywać pojedynczy znak we wzorcu wyszukiwania, dopasowują zero lub więcej znaków (w przypadku asteryksa/gwiazdki) lub jeden lub więcej znaków (w przypadku plusa).

W poniższym przykładzie możemy jeszcze bardziej zawęzić linie, które dopasowujemy, używając powtarzającego się *wieloznacznika* (symbolu wieloznacznego, symbolu maski):

```
# Szukaj linii, które zaczynają się od 'From:' i zawierają symbol
↪  małpy
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:.+@', line):
        print(line)

# Kod źródłowy: https://pl.py4e.com/code3/re04.py
```

Wzorzec wyszukiwania `^From:.+@` pomyślnie dopasuje te linie, które zaczynają się od "From:", po których następuje jeden lub więcej znaków (`.+`) i po których następuje znak małpy `@`. Zatem będzie on pasował do następującej linii:

```
From: stephen.marquard@uct.ac.za
```

Możesz myśleć o symbolu wieloznacznym `.+` jako o rozszerzeniu, które dopasowuje wszystkie znaki między znakiem dwukropka a znakiem mapły.

```
From:.+@
```

Dobrze jest też myśleć o plusie i gwiazdce jako o znakach "ekspansywnych". Na przykład, następujący wzorzec dopasowuje tekst aż do ostatniego znaku mapły w napisie, jako że `.+` wypycha na zewnątrz:

```
From: stephen.marquard@uct.ac.za, csev@umich.edu, and cwen @iupui.edu
```

Poprzez dodanie kolejnego znaku można też wskazać, że znak gwiazdki lub plusa nie ma być aż tak "zachłanny". Więcej informacji na temat wyłączania zachłannego zachowania tych znaków znajduje się w dokumentacji.

## 11.2. Wyciąganie danych przy użyciu wyrażeń regularnych

Jeśli w Pythonie chcemy wyodrębnić dane z ciągu znaków, to możemy użyć metody `findall()` do wyciągnięcia wszystkich podciągów, które pasują do wyrażenia regularnego. Użyjemy przykładu, w którym w dowolnej linii chcemy wyodrębnić wszystko to, co wygląda jak adres email, niezależnie od jej formatu. Na przykład, chcemy wyodrębnić adresy email z każdej z poniższych linii:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
          for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```

Nie chcemy pisać specjalnego kodu dla każdego z typów linii oraz różnego dzielenia i wycinania napisów dla przeczególnych typów linii. Następujący program używa `findall()` do znalenia linii z adresami mailowymi i do wyciągnięcia jednego lub więcej adresów mailowych z tych linii.

```
import re
s = 'A message from csev@umich.edu to cwen@iupui.edu about meeting
↪  @2PM'
lst = re.findall('\S+@\S+', s)
print(lst)

# Kod źródłowy: https://pl.py4e.com/code3/re05.py
```

Metoda `findall()` przeszukuje napis podany w drugim argumencie i zwraca listę wszystkich ciągnów znaków, które wyglądają jak adresy e-mail. Używamy sekwencji dwuznakowej, która oznacza dopasowanie dowolnego znaku, który nie jest tzw. biały znakiem[1] (`\S`).

Wynik programu jest następujący:

```
['csev@umich.edu', 'cwen@iupui.edu']
```

Tłumacząc wyrażenie regularne na ludzki język, szukamy podciągów, które zawierają co najmniej jeden *niebiały* znak, po którym następuje znak małpy, po którym następuje co najmniej jeszcze jeden niebiały znak. `\S+` dopasowuje tyle niebiałych znaków, ile to możliwe.

Wyrażenie regularne pasuje dwukrotnie (csev@umich.edu i cwen@iupui.edu), ale nie pasuje do napisuje "@2PM", ponieważ nie ma w nim żadnych niebiałych znaków *przed* znakiem małpy. Możemy w następujący sposób użyć tego wyrażenia regularnego by odczytać wszystkie linie w pliku i wypisać wszystko to, co wygląda jak adres e-mail:

[1] Przypomnijmy, że jest to zbiorcze określenie takich znaków jak spacja, znak tabulacji, znak końca linii lub dowolny inny znak niemający kształtu na ekranie.

```
# Szukaj linii, które zawierają znak małpy pomiędzy innymi znakami
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('\S+@\S+', line)
    if len(x) > 0:
        print(x)

# Kod źródłowy: https://pl.py4e.com/code3/re06.py
```

Czytamy każdą linie, a następnie wyodrębniamy wszystkie podciągi, które pasują do naszego wyrażenia regularnego. Ponieważ `findall()` zwraca listę, po prostu sprawdzamy czy liczba elementów na tej liście jest większa od zera, tak by wypisać tylko te linie, w których znaleźliśmy przynajmniej jeden podciąg wyglądający jak adres e-mail.

Jeśli uruchomimy program na pliku *mbox.txt*, otrzymamy następujący wynik:

```
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['<postmaster@collab.sakaiproject.org>']
['<200801032122.m03LMFo4005148@nakamura.uits.iupui.edu>']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
['source@collab.sakaiproject.org;']
```

Niektóre z naszych adresów e-mail mają nieprawidłowe znaki, takie jak "<" lub ";" na początku lub na końcu. Zaznaczmy w programie, że interesuje nas tylko ta część napisu, która zaczyna i kończy się literą lub cyfrą.

Aby to zrobić, używamy innej cechy wyrażeń regularnych. Kwadratowe nawiasy są używane do wskazania zbioru akceptowalnych znaków, które jesteśmy skłonni rozważyć jako pasujące. W pewnym sensie, `\S` prosi o dopasowanie zestawu "niebiałych znaków". Teraz będziemy nieco dokładniejsi, jeśli chodzi o znaki, które będziemy dopasowywać.

Oto nasze nowe wyrażenie regularne:

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

Robi się to trochę skomplikowane i możesz zacząć widzieć, dlaczego wyrażenia regularne są własnym, małym językiem. Tłumacząc to wyrażenie regularne na ludzki język, szukamy podciągów, które rozpoczynają się *pojedynczą* małą literą, dużą literą lub liczbą "[a-zA-Z0-9]", po których następuje zero lub więcej niebiałych znaków (`\S*`), po których następuje znak małpy, po których następuje zero lub więcej niebiałych znaków (`\S*`), po których następuje duża lub mała litera. Zauważ, że zmieniliśmy znak `+` na `*`, tak by wskazać zero lub więcej niebiałych znaków, ponieważ `[a-zA-Z0-9]` jest już jednym niebiałym znakiem. Pamiętaj, że `*` lub `+`

odnosi się do pojedynczego znaku znajdującego się bezpośrednio po lewej stronie plusa lub gwiazdki.

Jeśli użyjemy tego wyrażenia regularnego w naszym programie, nasze dane są bardziej przyzwoite:

```
# Szukaj linii, które zawierają znak małpy pomiędzy innymi znakami
# Znak początkowy musi być literą lub cyfrą
# Znak końcowy musi być literą
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('[a-zA-Z0-9]\S+@\S+[a-zA-Z]', line)
    if len(x) > 0:
        print(x)

# Kod źródłowy: https://pl.py4e.com/code3/re07.py
```

```
...
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

Zauważ, że w liniach `source@collab.sakaiproject.org`, nasze wyrażenie regularne wyeliminowało dwa znaki na końcu napisu (“>;”). Dzieje się tak dlatego, że kiedy dołączamy `[a-zA-Z]` na końcu naszego wyrażenia regularnego, wymagamy by jakikolwiek ciąg znaków, który znajdzie parser wyrażenia regularnego, musiał kończyć się literą. Więc kiedy parser widzi “>” na końcówce “sakaiproject.org>;”, to po prostu zatrzymuje się na ostatniej znalezionej “pasującej” literze (w tym przypadku “g” było ostatnim dobrym dopasowaniem).

Zauważ również, że wynik programu jest listą Pythona, która ma ciąg znaków jako pojedynczy element listy.

## 11.3. Łączenie wyszukiwania i wyodrębniania tekstu

Jeśli chcemy znaleźć liczby w liniach, które zaczynają się od ciągu znaków “X-”, np.:

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

nie chcemy dowolnych liczb zmiennoprzecinkowych z jakichkolwiek linii. Chcemy wyciągnąć z tekstu tylko te liczby z linii, które mają powyższą składnię, czyli zaczynają się od "X-".

Możemy skonstruować następujące wyrażenie regularne, tak aby wybrać interesujące nas linie:

```
^X-.*: [0-9.]+
```

Tłumacząc to na ludzki, mówimy, że chcemy te linie, które zaczynają się od `X-`, po których następuje zero lub więcej znaków (`.*`), po których następuje dwukropek (`:`), a następnie spacja. Po spacji szukamy jednego lub więcej znaków, które są albo cyfrą (0-9), albo kropką `[0-9.]+`. Zauważ, że wewnątrz nawiasów kwadratowych, kropka odpowiada rzeczywistej kropce (tzn. pomiędzy nawiasami kwadratowymi nie jest ona traktowana jako wieloznacznik).

Jest to bardzo zwięzłe wyrażenie, które będzie pasowało tylko do interesujących nas linii, tak jak to widać poniżej:

```
# Szukaj linii, które zaczynają się od 'X',
# po których występują dowolne niebiałe znaki oraz ':',
# po których występuje spacja i dowolna liczba.
# Liczba może być całkowita.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line):
        print(line)

# Kod źródłowy: https://pl.py4e.com/code3/re10.py
```

Kiedy uruchamimy program, widzimy ładnie przefiltrowane dane, które zawierją tylko szukane przez nas linie.

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
```

Teraz musimy rozwiązać problem wyodrębniania liczb. Chociaż byłoby to wystarczająco proste zrobić to przy użyciu `split`, to jednak możemy użyć kolejnej cechy wyrażeń regularnych, tak by w tym samym czasie zarówno wyszukać jak i przetworzyć linię.

Nawiasy okrągłe są kolejną szczególną cechą wyrażeń regularnych. Kiedy dodajemy nawiasy do wyrażenia regularnego, są one ignorowane podczas dopasowywania ciągu znaków. Ale kiedy używasz `findall()`, nawiasy wskazują, że podczas gdy chcesz by całe wyrażenie pasowało, to jesteś zainteresowany tylko wyciągnięciem części podciągu, który pasuje do wyrażenia regularnego.

Tak więc dokonujemy następującej zmiany w naszym programie:

```
# Szukaj linii, które zaczynają się od 'X',
# po których występują dowolne niebiałe znaki oraz ':',
# po których występuje spacja i dowolna liczba.
# Liczba może być całkowita.
# Wypisz liczbę jeśli jest ona większa niż zero.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^X\S*: ([0-9.]+)', line)
    if len(x) > 0:
        print(x)

# Kod źródłowy: https://pl.py4e.com/code3/re11.py
```

Zamiast wywołania `search()`, dodajemy nawiasy wokół części wyrażenia regularnego, która reprezentuje liczbę zmiennoprzecinkową, tak by wskazać `findall()`, że chcemy tylko pozyskać część liczby zmiennoprzecinkowej z pasującego napisu.

Wynik programu jest następujący:

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
..
```

Liczby wciąż są elementami list i muszą zostać przekonwertowane z ciągów znaków na typ zmiennoprzecinkowy. Niemniej, użyliśmy wachlarza możliwości wyrażeń regularnych zarówno do wyszukiwania, jak i wydobywania informacji, które uznaliśmy za interesujące.

Kolejny przykład użycia powyższje techniki jest to następujący. Jeśli spojrzymy na zawartość pliku, to znajdziemy tam wiele linii w następującej postaci:

```
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

Gdybyśmy chcieli wyciągnąć wszystkie numery rewizji (liczba całkowita na końcu tego typu wierszy) przy użyciu tej samej techniki, co chwilę wcześniej, moglibyśmy napisać następujący program:

```
# Szukaj linii, które mają postać 'Details:...rev='
# i są zakończone liczbą
# Jeśli liczba jest większa niż zero, to wypisz jej wartość
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^Details:.*rev=([0-9.]+)', line)
```

```
    if len(x) > 0:
        print(x)

# Kod źródłowy: https://pl.py4e.com/code3/re12.py
```

Tłumacząc nasze wyrażenie regularne po ludzku, szukamy linii, które rozpoczynają się od `Details:`, po których następuje dowolna liczba znaków (`.*`), po których następuje `rev=`, a następnie jedna lub więcej cyfr. Chcemy znaleźć te wiersze, które pasują do całego wyrażenia, ale chcemy z takiej linii wyciągnąć tylko liczbę całkowitą znajdującą się na końcu wiersza, więc otaczamy `[0-9]+` nawiasami okrągłymi.

Kiedy uruchamiamy program, otrzymujemy następujący wynik:

```
['39772']
['39771']
['39770']
['39769']
...
```

Pamiętaj, że `[0-9]+` jest "zachłanne", więc zanim wydobędziemy te cyfry, to stara się on uzyskać jak największy ciąg cyfr. To "zachłanne" zachowanie jest powodem dla którego otrzymujemy wszystkie pięć cyfr dla każdej liczby. Biblioteka wyrażeń regularnych przeszukuje w obu kierunkach tak długo aż napotka na coś innego niż cyfra albo na początek lub koniec wiersza.

Teraz możemy użyć wyrażeń regularnych, aby powtórzyć ćwiczenie z wcześniejszej części książki, w której byliśmy zainteresowani porą dnia każdej wiadomości. Szukaliśmy linii w postaci:

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

i dla każdej linii chcieliśmy wyodrębnić godzinę. Wcześniej robiliśmy to za pomocą dwóch wywołań `split`. Najpierw linia była dzielona na słowa, a następnie wyciągaliśmy piąte słowo i ponownie dzieliliśmy je po znaku dwukropka, tak aby wyciągnąć pierwsze dwa znaki, które nas interesowały.

Pomimo działania, w rzeczywistości skutkowało to dość *kruchym kodem*, który zakłada, że linie są ładnie sformatowane. Jeśli dodałbyś wystarczająco dużo sekcji sprawdzających błędy (lub duży blok try/except) by upewnić się, że Twój program nigdy nie zawiedzie podczas analizy źle sformatowanych linii, kod rozrósłby się do 10-15 linii kodu, które byłyby dość ciężkie do odczytania.

Na szczęście możemy to osiągnąć w znacznie prostszy sposób, używając następującego wyrażenia regularnego:

```
^From .* [0-9][0-9]:
```

Tłumaczenie tego wyrażenia regularnego jest takie, że szukamy linii, które zaczynają się od `From` (zwróć uwagę na spację), po których następuje dowolna liczba znaków (`.*`), po których następuje spacja, po której następują dwie cyfry `[0-9][0-9]`, po których następuje znak dwukropka. Jest definicja typów linii, których szukamy.

Aby wyciągnąć tylko godzinę przy użyciu `findall()`, dodajemy nawiasy okrągłe wokół dwóch cyfr w następujący sposób:

```
^From .* ([0-9][0-9]):
```

W ten sposób powstaje następujący program:

```
# Szukaj linii, które zaczynają się od 'From '
# i dowolnych znaków, po których następuje spacja
# i dwie cyfry od 00 do 99, po których występuje ':'
# Następnie wypisz liczbę jeśli jest większa niż zero
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^From .* ([0-9][0-9]):', line)
    if len(x) > 0: print(x)

# Kod źródłowy: https://pl.py4e.com/code3/re13.py
```

Gdy uruchomimy program, uzyskamy następujący wynik:

```
['09']
['18']
['16']
['15']
...
```

## 11.4. Znak ucieczki

Ponieważ w wyrażeniach regularnych używamy znaków specjalnych, tak aby dopasować początek lub koniec wiersza lub określić wieloznaczniki, potrzebujemy jakiegoś sposobu na wskazanie, że te znaki są "normalne" i chcemy dopasować rzeczywisty znak, taki jak znak dolara lub daszka.

Możemy wskazać, że chcemy po prostu dopasować znak, poprzedzając go odwrotnym ukośnikiem \ (jest to tzw. znak ucieczki; inne nazwy to znak modyfikacji lub znak uwalniania). Na przykład, przy pomocy poniższego wyrażenia regularnego możemy znaleźć kwoty pieniędzy.

```
import re
x = 'Uzyskaliśmy $10.00 za ciasteczka.'
y = re.findall('\$[0-9.]+',x)
```

Ponieważ poprzedzamy znak dolara odwrotnym ukośnikiem, w rzeczywistości odpowiada on znakowi dolara w wejściowym napisie, zamiast oznaczać "koniec linii", a reszta wyrażenia regularnego odpowiada jednej lub kilku cyfrom lub znakowi kropki. *Uwaga:* w nawiasach kwadratowych znaki nie są traktowane jako "specjalne". Jeśli napiszemy `[0-9.]`, to w rzeczywistości oznacza to cyfry lub kropkę. Poza nawiasami kwadratowymi, kropka jest znakiem symbolu wieloznacznego i dopasowuje każdy znak. W nawiasach kwadratowych, kropka jest kropką.

## 11.5. Podsumowanie

Podczas gdy to tylko zarysowaliśmy niewielki zakres wyrażeń regularnych, to jednak nauczyliśmy się trochę o ich języku. Są to ciągi specjalnych znaków przeznaczone do wyszukiwania, które przekazują Twoje życzenia do systemu wyrażeń regularnych, który z kolei definiuje "dopasowanie" i wydobywa tekst z dopasowanych ciągów. Oto niektóre z tych specjalnych znaków i sekwencji znaków:

`^` Dopasowuje początek linii.

`$` Dopasowuje koniec linii.

`.` Dopasowuje dowolny znak (wieloznacznik).

`\s` Dopasowuje biały znak.

`\S` Dopasowuje niebiały znak (przeciwieństwo \s).

`*` Dotyczy bezpośrednio poprzedzającego(ych) znaku(ów) i wskazuje, by dopasować go/je zero lub więcej razy.

`*?` Dotyczy bezpośrednio poprzedzającego(ych) znaku(ów) i wskazuje, by dopasować go/je zero lub więcej razy "wyłączając tryb zachłanny".

`+` Dotyczy bezpośrednio poprzedzającego(ych) znaku(ów) i wskazuje by dopasować go/je jeden lub więcej razy.

`+?`Dotyczy bezpośrednio poprzedzającego(ych) znaku(ów) i wskazuje by dopasować go/je jeden lub więcej razy "wyłączając tryb zachłanny".

`?` Dotyczy bezpośrednio poprzedzającego(ych) znaku(ów) i wskazuje by dopasować go/je zero lub jeden raz.

`??` Dotyczy bezpośrednio poprzedzającego(ych) znaku(ów) i wskazuje by dopasować go/je zero lub jeden raz "wyłączając tryb zachłanny".

`[aeiou]` Dopasowuje pojedynczy znak o ile ten znak jest w określonym zestawie. W tym przykładzie dopasowanie dotyczy "a", "e", "i", "o" lub "u", ale nie dopasowuje żadnych innych znaków.

`[a-z0-9]` Możesz określić zakresy znaków, używając znaku minus. Ten przykład to pojedynczy znak, który musi być małą literą lub cyfrą.

`[^A-Za-z]` Gdy pierwszym znakiem w zapisie zestawu znaków jest kareta (daszek), to oznacza on zaprzeczenie. Ten przykład odpowiada pojedynczemu znakowi, który jest *czymś innym* niż duże lub małe litery.

`( )` Gdy nawiasy okrągłe są dodawane do wyrażenia regularnego, to są one ignorowane podczas dopasowania, ale pozwalają na wyodrębnienie konkretnego podzbioru dopasowanego napisu, a nie całego ciągu znaków, podczas użycia `findall()`.

`\b` Dopasowuje pusty ciąg znaków, ale tylko na początku lub na końcu wyrazu.

`\B` Dopasowuje pusty ciąg znaków, ale ani na początku ani na końcu słowa.

`\d` Dopasowuje dowolną cyfrę dziesiętną; odpowiednik [0-9].

`\D` Dopasowuje dowolny znak niebędący cyfrą; odpowiednik zestawu [^0-9].

## 11.6. Dodatek dla użytkowników systemu Unix / Linux

Wsparcie wyszukiwania plików za pomocą wyrażeń regularnych zostało wbudowane w system operacyjny Unix w latach 60. i jest dostępne w prawie wszystkich językach programowania w takiej czy innej formie.

W rzeczywistości Unix posiada wbudowany program wiersza poleceń o nazwie *grep* (Generalized Regular Expression Parser, czyli uogólniony parser wyrażeń regularnych), który robi prawie to samo, co przykłady z `search()` umieszczone w tym rozdziale. Więc jeśli masz system Macintosh lub Linux, możesz spróbować następujących komend w oknie linii poleceń.

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

Powyższa linia mówi programowi `grep` żeby pokazał Ci linie, które zaczynają się od ciągu znaków "From:" w pliku *mbox-short.txt*. Jeśli trochę poeksperymentujesz z komendą `grep` i przeczytasz dokumentację tego programu, znajdziesz kilka subtelnych różnic między obsługą wyrażeń regularnych w Pythonie a obsługą wyrażeń regularnych w `grep`. Dla przykładu, `grep` nie obsługuje `\S` oznaczającego niebiały znak, więc będziesz musiał użyć nieco bardziej złożonego zapisu (w uproszczeniu) `[^ \n\r\t]`, co po prostu oznacza dopasowanie znaku, który jest czymś innym niż spacja, znaki końca linii lub tabulacja.

## 11.7. Debugowanie

Python ma wbudowaną prostą i podstawową dokumentację, która może być bardzo pomocna, jeśli musisz szybko przypomnieć sobie nazwę danej metody. Dokumentacja ta może być przeglądana w trybie interaktywnym w interpreterze Pythona.

Możesz wywołać interaktywny system pomocy używając `help()`.

```
>>> help()

help> modules
```

Aby wyjść z systemu pomocy wpisz `quit`.

Jeśli wiesz, jakiego modułu chcesz użyć, możesz użyć polecenia `dir()` w następujący sposób, tak aby znaleźć dostępne metody w tym module:

```
>>> import re
>>> dir(re)
[.. 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer', 'match', 'purge', 'search', 'split', 'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

Możesz również uzyskać niewielką część dokumentacji na temat określonej metody za pomocą polecenia `help`.

```
>>> help(re.search)
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern,
↪   returning
    a match object, or None if no match was found.
>>>
```

Wbudowana dokumentacja nie jest zbyt obszerna, ale może być pomocna, gdy potrzebujesz coś szybko znaleźć lub nie masz dostępu do przeglądarki internetowej lub wyszukiwarki.

## 11.8. Słowniczek

**dopasowanie zachłanne** Pojęcie oznaczające, że znaki `+` i `*` w wyrażeniu regularnym rozszerzają się na zewnątrz, tak aby dopasować jak największy możliwy ciąg znaków.

**grep** Polecenie dostępne w większości systemów unixowych, które przeszukuje pliki tekstowe w poszukiwaniu linii pasujących do wyrażeń regularnych. Nazwa komendy oznacza "Generalized Regular Expression Parser", czyli uogólniony parser wyrażeń regularnych.

**kruchy kod** Kod programu, który działa tylko wtedy, gdy dane wejściowe są w określonym formacie, ale jest podatny na wysypanie całego programu, jeśli wystąpią jakieś odchylenia od właściwego formatu. Nazywamy to "kruchym kodem", ponieważ można go łatwo zepsuć.

**wieloznacznik** Znak specjalny, który pasuje do dowolnego znaku. W wyrażeniach regularnych znakiem wieloznacznika jest kropka.

**wyrażenie regularne** Język służący do tworzenia bardziej złożonych ciągów wyszukiwania. Wyrażenie regularne może zawierać znaki specjalne, które wskazują, że szukane wyrażenie pasuje tylko do początku lub końca wiersza lub wielu innych podobnych możliwości.

## 11.9. Ćwiczenia

**Ćwiczenie 1: Napisz prosty program symulujący działanie unixowej komendy `grep`. Poproś użytkownika o wpisanie wyrażenia regularnego i zliczenie linii, które pasowały do wyrażenia regularnego:**

```
$ python grep.py
Podaj wyrażenie regularne: ^Author
mbox.txt ma 1798 linii, które pasują do ^Author

$ python grep.py
```

```
Podaj wyrażenie regularne: ^X-
mbox.txt ma 14368 linii, które pasują do ^X-

$ python grep.py
Podaj wyrażenie regularne: java$
mbox.txt ma 4175 linii, które pasują do java$
```

**Ćwiczenie 2: Napisz program zajdujący linie w postaci:**

```
New Revision: 39772
```

**Wyodrębnij liczbę z każdej pasującej linii za pomocą wyrażenia regularnego i metody `findall()`. Oblicz średnią z tych liczb i wypisz ją w postaci liczby całkowitej.**

```
Podaj nazwę pliku: mbox.txt
38549

Podaj nazwę pliku: mbox-short.txt
39756
```

# Rozdział 12

# Programy sieciowe

Podczas gdy wiele przykładów w tej książce koncentruje się na odczytywaniu plików i wyszukiwaniu danych w tych plikach, istnieje wiele różnych źródeł informacji, jeśli wziąć pod uwagę internet.

W tym rozdziale będziemy udawać, że jesteśmy przeglądarką internetową i będziemy wyszukiwać strony internetowe za pomocą protokołu HTTP (Hypertext Transfer Protocol, czyli protokół przesyłania dokumentów hipertekstowych). Następnie będziemy odczytywać dane ze stron internetowych oraz będziemy te dane przetwarzać.

## 12.1. Hypertext Transfer Protocol - HTTP

Protokół sieciowy, który napędza sieć internetową, jest w rzeczywistości dość prosty i ma wbudowane w Pythona wsparcie poprzez funkcję `socket`, która ułatwia w programie Pythona nawiązywanie połączeń sieciowych i pobieranie danych przez tzw. gniazda (lub z ang. sockety).

*Gniazdo* jest bardzo podobne do pliku, z tą różnicą, że jedno gniazdo zapewnia dwukierunkowe połączenie między dwoma programami. Przy pomocy tego samego gniazda możesz zarówno odczytywać dane, jak i wysyłać/zapisywać dane. Jeżeli wyślesz jakieś dane do gniazda, to zostaną one wysłane do aplikacji znajdującej się na drugim końcu tego gniazda. Jeżeli czytasz dane z gniazda, to otrzymujesz dane, które wysłała do Ciebie aplikacja po drugiej stronie.

Ale jeżeli próbujesz odczytać z gniazda dane wtedy, gdy program na drugim końcu nie wysłał jeszcze żadnych danych, to po prostu siadasz i czekasz. Jeżeli programy na obu końcach gniazda po prostu czekają na jakieś dane, nie wysyłając niczego, to będą czekać bardzo długo, więc ważną częścią programów komunikujących się przez Internet jest posiadanie pewnego rodzaju protokołu.

Protokół jest zestawem precyzyjnych reguł, które określają kto zaczyna pierwszy krok, co ma zrobić, a następnie jakie są możliwe odpowiedzi na tę wiadomość, kto wysyła następną wiadomość itd. W pewnym sensie te dwie aplikacje na obu końcach gniazda wykonują taniec i upewniają się, że nie nadepną na siebie nawzajem.

Istnieje wiele dokumentów, które opisują protokoły sieciowe. Hypertext Transfer Protocol jest opisany w poniższym dokumencie:

https://www.w3.org/Protocols/rfc2616/rfc2616.txt

Jest to długi i skomplikowany 176-stronicowy dokument z dużą ilością szczegółów. Jeśli uważasz, że jest interesujący, przeczytaj go w całości. Ale jeśli rozejrzysz się po stronie nr 36 dokumentu RFC2616, to znajdziesz składnię dla żądania GET. Aby poprosić o dokument z serwera WWW, wykonujemy połączenie z serwerem `data.pr4e.org` na porcie 80, a następnie wysyłamy linię w postaci

```
GET http://data.pr4e.org/romeo.txt HTTP/1.0
```

gdzie drugim parametrem jest strona internetowa, o którą prosimy, a następnie wysyłamy pustą linię. Serwer internetowy odpowie na zapytanie pewnym nagłówkiem i pustym wierszem, po którym wyśle treść dokumentu.

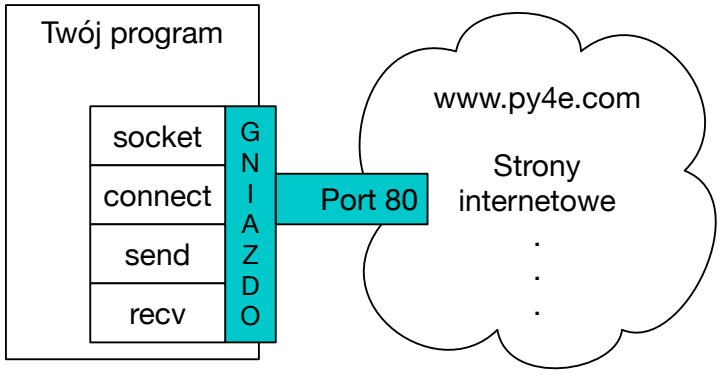
Natomiast czym jest port? Jest to liczba, który generalnie wskazuje, z którą aplikacją kontaktujesz się podczas połączenia z serwerem. Dla przykładu, ruch internetowy zazwyczaj korzysta z portu 80 (HTTP) lub 443 (szyfrowany HTTP), podczas gdy ruch mailowy korzysta z portu 25.

## 12.2. Najprostsza przeglądarka internetowa na świecie

Być może najprostszym sposobem na pokazanie, jak działa protokół HTTP, jest napisanie bardzo prostego programu, który nawiązuje połączenie z serwerem WWW i postępuje zgodnie z zasadami protokołu HTTP, tak by zażądać dokumentu i wyświetlić to, co serwer wysyłał z powrotem.

```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('data.pr4e.org', 80))
cmd = 'GET http://data.pr4e.org/romeo.txt HTTP/1.0\r\n\r\n'.encode()
mysock.send(cmd)

while True:
    data = mysock.recv(512)
    if len(data) < 1:
        break
    print(data.decode(),end='')

mysock.close()

# Kod źródłowy: https://pl.py4e.com/code3/socket1.py
```

Najpierw program nawiązuje połączenie z serwerem na porcie 80 data.pr4e.org. Ponieważ nasz program pełni rolę "przeglądarki internetowej", protokół HTTP mówi, że musimy wysłać polecenie GET, a następnie pustą linię. `\r\n` oznacza




Rysunek 12.1: Połączenie przez gniazdo

EOL (ang. end of line, czyli koniec linii), a więc `\r\n\r\n` oznacza pustkę pomiędzy dwiema sekwencjami EOL. Jest to odpowiednik pustego wiersza.

Gdy wyślemy ten pusty wiersz, to piszemy pętlę, która odbiera z gniazda dane w 512-znakowych kawałkach i drukuje te dane aż do momentu gdy nie będzie więcej danych do odczytania (tj. `recv()` zwróci pusty ciąg znaków).

Program generuje następujące wynik:

```
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 18:52:55 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Sat, 13 May 2017 11:22:22 GMT
ETag: "a7-54f6609245537"
Accept-Ranges: bytes
Content-Length: 167
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: text/plain

But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Wyjście zaczyna się od nagłówków, które serwer WWW wysyła w celu opisania dokumentu. Na przykład, nagłówek `Content-Type` wskazuje, że dokument jest zwykłym dokumentem tekstowym (`text/plain`).

Po wysłaniu przez serwer nagłówków, dodaje on pustą linię wskazującą koniec nagłówków, a następnie wysyła rzeczywiste dane zawarte w pliku *romeo.txt*.

Powyższy przykład pokazuje jak przy pomocy gniazd wykonać niskopoziomowe połączenie sieciowe. Gniazda mogą być używane do komunikacji z serwerem WWW lub z serwerem pocztowym albo z wieloma innymi rodzajami serwerów. Wystarczy

znaleźć dokument, który opisuje protokół, i napisać kod do wysłania i odbioru danych zgodnie z tym protokołem.

Ponieważ jednak najczęściej używanym przez nas protokołem jest protokół internetowy HTTP, Python posiada specjalną bibliotekę zaprojektowaną specjalnie do obsługi protokołu HTTP w celu pobierania dokumentów i danych umieszczonych w sieci.

Jednym z wymogów korzystania z protokołu HTTP jest konieczność wysyłania i odbierania danych w postaci obiektów bajtowych, a nie ciągów znaków. W poprzednim przykładzie, metody `encode()` i `decode()` przekształcają odpowiednio ciągi znaków na obiekty bajtowe i obiekty bajtowe na ciągi znaków.

Następny przykład używa notacji `b''` do określenia, że zmienna powinna być przechowywana jako obiekt bajtowy. `encode()` i `b''` są równoważne, o ile nie używamy np. znaków diakrytycznych (generalnie użycie `encode()` jest bardziej uniwersalne).

```
>>> b'Hello world'
b'Hello world'
>>> 'Hello world'.encode()
b'Hello world'
```

## 12.3. Pobieranie obrazu przez HTTP

W poprzednim przykładzie pobraliśmy zwykły plik tekstowy, który zawierał w pliku znaki końca linii i po prostu wyświetliliśmy te dane na ekranie w trakcie działania programu. Podobnego programu możemy użyć do odtworzenia obrazu za pomocą HTTP. Zamiast wyświetlać dane na ekranie w trakcie działania programu, zgromadzimy je w obiekcie bajtowym, usuniemy nagłówki, a następnie zapiszemy dane obrazu do pliku:

```
import socket
import time

HOST = 'data.pr4e.org'
PORT = 80
mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect((HOST, PORT))
mysock.sendall(b'GET http://data.pr4e.org/cover3.jpg
↪  HTTP/1.0\r\n\r\n')
count = 0
picture = b""

while True:
    data = mysock.recv(5120)
    if len(data) < 1: break
    #time.sleep(0.25)
    count = count + len(data)
    print(len(data), count)
```

```
    picture = picture + data

mysock.close()

# Znajdź koniec nagłówka (2 CRLF)
pos = picture.find(b"\r\n\r\n")
print('Header length', pos)
print(picture[:pos].decode())

# Pomiń nagłówek i zapisz dane obrazu
picture = picture[pos+4:]
fhand = open("stuff.jpg", "wb")
fhand.write(picture)
fhand.close()

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

Po uruchomieniu programu, generuje on następujące wyjście:

```
$ python urljpeg.py
5120 5120
5120 10240
4240 14480
5120 19600
...
5120 214000
3200 217200
5120 222320
5120 227440
3167 230607
Header length 393
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 18:54:09 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: image/jpeg
```

Widać, że dla tego adresu URL nagłówek `Content-Type` wskazuje, że treść dokumentu jest obrazem (`image/jpeg`). Gdy program zakończy pracę, możesz obejrzeć dane obrazu otwierając plik `stuff.jpg` w programie do przeglądania obrazów.

W trakcie działania programu widać, że nie otrzymujemy 5120 znaków za każdym razem, gdy wywołujemy metodę `recv()`. Otrzymujemy tyle znaków, ile zostało wysłanych do nas przez serwer WWW w momencie wywołania `recv()`. W tym

przykładzie, albo otrzymujemy od 3200 do 5120 znaków za każdym razem gdy żądamy danych.

Twoje wyniki mogą być różne w zależności od szybkości Twojej sieci. Zauważ również, że przy ostatnim wywołaniu `recv()` otrzymujemy 3167 bajtów, co jest końcem strumienia, a przy następnym wywołaniu `recv()` otrzymujemy ciąg o zerowej długości, który mówi nam, że serwer wywołał `close()` na końcu gniazda i nie ma już więcej danych.

Możemy spowolnić nasze kolejne wywołania `recv()` przez odkomentowanie wywołania `time.sleep()`. W ten sposób czekamy ćwierć sekundy po każdym wywołaniu, tak aby serwer mógł "wyprzedzić" nas i przesłać nam więcej danych, zanim ponownie wywołamy `recv()`. Z włączonym opóźnieniem program wykonuje się w następujący sposób:

```
$ python urljpeg.py
5120 5120
5120 10240
5120 15360
...
5120 225280
5120 230400
207 230607
Header length 393
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 21:42:08 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: image/jpeg
```

Teraz, poza pierwszym i ostatnim wywołaniem `recv()`, otrzymujemy 5120 znaków za każdym razem gdy prosimy o nowe dane.

Istnieje bufor pomiędzy serwerem wykonującym żądania `send()` a naszą aplikacją wykonującą żądania `recv()`. Kiedy uruchamiamy program z założonym opóźnieniem, w pewnym momencie serwer może zapełnić bufor w gnieździe i być zmuszony do pauzy tak długo aż nasz program zacznie opróżniać bufor. Wstrzymanie aplikacji wysyłającej lub odbierającej w takiej sytuacji nazywane jest "kontrolą przepływu".

## 12.4. Pobieranie stron internetowych przy pomocy `urllib`

Choć możemy ręcznie wysyłać i odbierać dane za pomocą protokołu HTTP przy użyciu biblioteki gniazd, istnieje w Pythonie znacznie prostszy sposób na wykonanie

tego zadania poprzez użycie biblioteki `urllib`.

Używając `urllib`, możesz traktować stronę internetową jak plik. Po prostu wskazujesz stronę internetową do pobrania, a `urllib` obsługuje wszystkie szczegóły dotyczące protokołu HTTP i nagłówka.

Równoważny kod odczytujący z sieci plik *romeo.txt* przy użyciu `urllib` jest następujący:

```
import urllib.request

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
for line in fhand:
    print(line.decode().strip())

# Kod źródłowy: https://pl.py4e.com/code3/urllib1.py
```

Po otwarciu strony internetowej za pomocą `urllib.urlopen`, możemy potraktować ją jak plik i odczytać ją za pomocą pętli `for`.

Po uruchomieniu programu widzimy tylko zawartość pliku. Nagłówki są nadal wysyłane przez serwer, ale kod `urllib` pochłania nagłówki i zwraca nam tylko dane.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Jako kolejny przykład możemy napisać program, który pobierze dane z `romeo.txt` i obliczy liczbę wystąpień każdego słowa:

```
import urllib.request, urllib.parse, urllib.error

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')

counts = dict()
for line in fhand:
    words = line.decode().split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1
print(counts)

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

Po otwarciu strony internetowej, możemy odczytać jej zawartość jak z lokalnego pliku.

## 12.5. Odczytywanie plików binarnych za pomocą `urllib`

Czasami chcesz pobrać nietekstowy (lub binarny) plik, taki jak zdjęcie lub plik wideo. Dane w tych plikach na ogół nie są przydatne do wypisywania na ekranie, jednak za pomocą `urllib` możesz łatwo wykonać kopię pliku znajdującego się pod adresem URL do lokalnego pliku na dysku twardym.

Schemat postępowania polega na otwarciu adresu URL i użyciu `read` do pobrania całej zawartości dokumentu do zmiennej zawierającej ciąg znaków (`img`), a następnie zapisaniu tej informacji do pliku lokalnego:

```
import urllib.request, urllib.parse, urllib.error

img =
↪  urllib.request.urlopen('http://data.pr4e.org/cover3.jpg').read()
fhand = open('cover3.jpg', 'wb')
fhand.write(img)
fhand.close()

# Kod źródłowy: https://pl.py4e.com/code3/curl1.py
```

Program ten odczytuje przez sieć wszystkie dane na raz i przechowuje je w zmiennej `img` w pamięci głównej komputera, a następnie otwiera plik `cover.jpg` i zapisuje dane na dysku. Argument `wb` w wywołaniu funkcji `open()` otwiera plik binarny tylko do zapisu. Program ten będzie działał, jeśli rozmiar pliku jest mniejszy niż rozmiar pamięci Twojego komputera.

Jeśli jednak jest to duży plik audio lub wideo, powyższy program może się zawiesić lub przynajmniej działać bardzo wolno w momencie gdy na Twoim komputerze zabraknie pamięci. Aby uniknąć wyczerpania pamięci, pobieramy dane w blokach (lub buforach), a następnie zapisujemy każdy blok na dysku przed pobraniem kolejnego bloku. W ten sposób program może odczytać plik o dowolnym rozmiarze bez zajmowania całej pamięci, którą posiadasz w komputerze.

```
import urllib.request, urllib.parse, urllib.error

img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg')
fhand = open('cover3.jpg', 'wb')
size = 0
while True:
    info = img.read(100000)
    if len(info) < 1: break
    size = size + len(info)
    fhand.write(info)

print('Skopiowano', size, 'znaków.')
fhand.close()

# Kod źródłowy: https://pl.py4e.com/code3/curl2.py
```

W powyższym przykładzie odczytujemy jednocześnie tylko 100 000 znaków, a następnie zapisujemy te znaki w pliku `cover.jpg` przed pobraniem z sieci kolejnych 100 000 znaków danych.

Program działa następująco:

```
python curl2.py
Skopiowano 230210 znaków.
```

## 12.6. Parsowanie HTMLa, roboty internetowe i web scraping

Jednym z częstych zastosowań `urllib` jest tzw. *web scraping* (dosł. wyskrobywanie danych z sieci). Mamy z nim do czynienia wtedy, gdy piszemy program, który udaje przeglądarkę internetową i pobiera strony, a następnie analizujemy dane zawarte na tych stronach w poszukiwaniu interesujących nas wzorców.

Na przykład, wyszukiwarka taka jak Google, sprawdza kod źródłowy jednej strony internetowej, wyodrębnia linki do innych stron, a następnie pobiera te strony, wyodrębnia z nich linki itd. Używając tej techniki, Google poprzez swoje *roboty internetowe* indeksuje/przechodzi przez prawie wszystkie strony znajdujące się w sieci internetowej.

Google wykorzystuje również informacje liczbie linków z innych stron do danej strony, jako jedną z miar tego jak "ważna" jest dana strona i jak wysoko powinna się ona znajdować w wynikach wyszukiwania.

## 12.7. Parsowanie HTMLa przy użyciu wyrażeń regularnych

Jednym z prostych sposobów parsowania HTMLa jest użycie wyrażeń regularnych do wielokrotnego wyszukiwania i wyodrębnienia podciągów znaków, które pasują do konkretnego wzorca.

Poniżej mamy prostą stronę internetową:

```
<h1>The First Page</h1>
<p>
If you like, you can switch to the
<a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>.
</p>
```

Możemy skonstruować wyrażenie regularne, tak aby dopasować i wyodrębnić z powyższego tekstu linki:

```
href="http[s]?://.+?"
```

Nasze wyrażenie regularne szuka napisów, które zaczynają się od “href="http://” lub “href="https://”, po których następuje jeden lub więcej znaków (`.+?`), po których następuje kolejny cudzysłów. Znak zapytania `?` za `[s]` wskazuje na szukanie łańcucha “http”, po którym następuje zero lub jedno “s”.

Znak zapytania dodany do ‘.+¿‘ wskazuje, że dopasowanie nie być wykonane w trybie “zachłannym”. Takie dopasowanie próbuje znaleźć *najmniejszy* możliwy pasujący ciąg znaków, a z kolei zachłanne dopasowanie próbuje znaleźć *największy* możliwy pasujący ciąg znaków.

Do naszego wyrażenia regularnego dodajemy nawiasy, aby wskazać, którą część pasującego napisu chcielibyśmy wydobyć. Program jest następujący:

```
# Wyszukaj linki w danych z URL
import urllib.request, urllib.parse, urllib.error
import re
import ssl

# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Podaj link - ')
html = urllib.request.urlopen(url, context=ctx).read()
links = re.findall(b'href="(http[s]?://.*?)"', html)
for link in links:
    print(link.decode())

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

Biblioteka `ssl` pozwala naszemu programowi na dostęp do stron internetowych, które ściśle wymuszają HTTPS. Metoda `read` zwraca kod źródłowy HTML jako obiekt bajtowy, zamiast zwracać obiekt typu HTTPResponse. Metoda `findall` daje nam listę wszystkich ciągów znaków, które pasują do naszego wyrażenia regularnego, zwracając tylko tekst linku zawarty pomiędzy cudzysłowami.

Kiedy uruchomimy program i wprowadzimy adres URL, otrzymamy mniej więcej następujący wynik:

```
Podaj link - https://docs.python.org
https://docs.python.org/3/index.html
https://www.python.org/
https://docs.python.org/3.8/
https://docs.python.org/3.7/
https://docs.python.org/3.5/
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
https://www.python.org/dev/peps/
https://wiki.python.org/moin/BeginnersGuide
https://wiki.python.org/moin/PythonBooks
https://www.python.org/doc/av/
https://www.python.org/
```

```
https://www.python.org/psf/donations/
http://sphinx.pocoo.org/
```

Wyrażenia regularne działają bardzo dobrze wtedy, gdy analizowany kod HTML jest dobrze sformatowany i przewidywalny. Niestety istnieje wiele "zepsutych" stron HTML, więc poleganie na rozwiązaniu opierającym się tylko na wyrażeniach regularnych może albo pominąć niektóre poprawne linki, albo skończyć się pozyskaniem niepoprawnych danych.

Problem ten może zostać rozwiązany za pomocą solidnej i stabilnej biblioteki do parsowania HTMLa.

## 12.8. Parsowanie HTMLa przy użyciu BeautifulSoup

Nawet jeśli HTML wygląda jak XML[1] i niektóre strony faktycznie są starannie skonstruowane jako XML, to niestety większość stron HTML jest zazwyczaj popsuta w taki sposób, że parser XMLa odrzuca całą stronę HTML jako nieprawidłowo sformatowany dokument.

Istnieje wiele bibliotek Pythona, które mogą pomóc Ci w parsowaniu HTMLa i wyodrębnieniu danych ze stron. Każda z tych bibliotek ma swoje mocne i słabe strony, a Ty możesz wybrać jedną z nich w zależności od swoich potrzeb.

Jako przykład, przetworzymy trochę danych HTML i wyodrębnimy linki za pomocą biblioteki *BeautifulSoup*. Biblioteka BeautifulSoup toleruje bardzo wadliwy kod HTML i nadal pozwala na łatwe wyodrębnienie potrzebnych danych. Możesz pobrać i zainstalować kod BeautifulSoup ze strony:

https://pypi.org/project/beautifulsoup4/

Bibliotekę możemy zainstalować poprzez wiersz linii poleceń wydając komendę:

```
pip3 install beautifulsoup4
```

Bardziej szczegółowe informacje na temat instalacji bibliotek za pomocą narzędzia *Python Package Index* `pip` są dostępne na stronie internetowej:

https://packaging.python.org/tutorials/installing-packages/

Użyjemy `urllib` do odczytania strony, a następnie użyjemy `BeautifulSoup` do wyodrębnienia atrybutów `href` ze znacznika hiperłącza (`a`).

```
# Aby uruchomić poniższy kod, poprzez wiersz linii
# poleceń zainstaluj bibliotekę BeautifulSoup:
#
#    pip3 install beautifulsoup4
#

import urllib.request, urllib.parse, urllib.error
```

[1] Format XML jest opisany w następnym rozdziale.

```
from bs4 import BeautifulSoup
import ssl

# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Podaj link - ')
html = urllib.request.urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')

# Pobierz wszystkie znaczniki hiperłączy
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

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

Program pyta o adres internetowy, następnie otwiera stronę, odczytuje dane i przekazuje je do parsera BeautifulSoup, a następnie pobiera wszystkie znaczniki hiperłączy i dla każdego z nich wypisuje atrybut `href`.

Po uruchomieniu programu uzyskamy mniej więcej następujący wynik:

```
Enter - https://docs.python.org
genindex.html
py-modindex.html
https://www.python.org/
#
whatsnew/3.6.html
whatsnew/index.html
tutorial/index.html
library/index.html
reference/index.html
using/index.html
howto/index.html
installing/index.html
distributing/index.html
extending/index.html
c-api/index.html
faq/index.html
py-modindex.html
genindex.html
glossary.html
search.html
contents.html
bugs.html
about.html
license.html
copyright.html
download.html
https://docs.python.org/3.8/
```

```
https://docs.python.org/3.7/
https://docs.python.org/3.5/
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
https://www.python.org/dev/peps/
https://wiki.python.org/moin/BeginnersGuide
https://wiki.python.org/moin/PythonBooks
https://www.python.org/doc/av/
genindex.html
py-modindex.html
https://www.python.org/
#
copyright.html
https://www.python.org/psf/donations/
bugs.html
http://sphinx.pocoo.org/
```

Tym razem lista jest o wiele dłuższa, ponieważ niektóre znaczniki hiperłączy są ścieżkami względnymi (np. "tutorial/index.html") lub odniesieniami wewnątrz strony (np. "#"), które nie zawierają "http://" lub "https://", co było wymogiem w naszym podejściu z wyrażeniem regularnym.

Możesz również użyć BeautifulSoup do wyciągnięcia różnych elementów związanych ze znacznikiem HTML:

```
# Aby uruchomić poniższy kod, poprzez wiersz linii
# poleceń zainstaluj bibliotekę BeautifulSoup:
#
#    pip3 install beautifulsoup4
#

from urllib.request import urlopen
from bs4 import BeautifulSoup
import ssl

# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, "html.parser")

# Pobierz wszystkie znaczniki hiperłączy
tags = soup('a')
for tag in tags:
    # Przejrzyj elementy związane ze znacznikiem
    print('TAG:', tag)
    print('URL:', tag.get('href', None))
    print('Contents:', tag.contents[0])
    print('Attrs:', tag.attrs)
```

```
# Kod źródłowy: https://pl.py4e.com/code3/urllink2.py

python urllink2.py
Podaj link - http://www.dr-chuck.com/page1.htm
TAG: <a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
Content: ['\nSecond Page']
Attrs: [('href', 'http://www.dr-chuck.com/page2.htm')]
```

`html.parser` to parser HTMLa zawarty w standardowej bibliotece Pythona 3. Informacje o innych parserach HTMLa są dostępne na stronie internetowej:

https://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser

Powyższe przykłady są tylko namiastką możliwości jakie stoją za BeautifulSoup jeśli chodzi o parsowanie HTMLa.

## 12.9. Dodatek dla użytkowników systemu Unix / Linux

Jeśli posiadasz komputer z systemem Linux, Unix lub macOS, prawdopodobnie posiadasz wbudowane w system komendy, które pobierają zarówno zwykły tekst, jak i pliki binarne za pomocą protokołów HTTP lub FTP (File Transfer Protocol, czyli protokół transferu plików). Jedną z tych komend jest `curl`:

```
$ curl -O https://www.py4e.com/cover.jpg
```

Polecenie `curl` jest skrótem od "copy URL" (z ang. skopiuj URL) i dlatego dwa wymienione wcześniej na pl.py4e.com/code3 przykłady pobierania plików binarnych z `urllib` mają nazwy `curl1.py` i `curl2.py`, ponieważ implementują podobną funkcjonalność jak polecenie `curl`. Istnieje również przykładowy program `curl3.py`, który wykonuje to zadanie nieco efektywniej, na wypadek gdybyś rzeczywiście chciał użyć tego schematu w swoim programie.

Drugim poleceniem, które działa bardzo podobnie, jest `wget`:

```
$ wget http://www.py4e.com/cover.jpg
```

Obie te komendy sprawiają, że pobieranie stron internetowych i plików jest prostą czynnością.

## 12.10. Słowniczek

**BeautifulSoup** Biblioteka Pythona do przetwarzania dokumentów HTML i wyodrębniania danych z dokumentów HTML, która kompensuje większość niedoskonałości w kodzie HTML, które przeglądarki internetowe zazwyczaj ignorują. Biblioteka BeautifulSoup ma swoją stronę pod adresem www.crummy.com/software/BeautifulSoup.

**gniazdo** Połączenie sieciowe pomiędzy dwiema aplikacjami, w którym aplikacje mogą wysyłać i odbierać dane w obu kierunkach.

**port** Liczba, który generalnie wskazuje, z którą aplikacją kontaktujesz się podczas połączenia z serwerem. Dla przykładu, ruch internetowy zazwyczaj korzysta z portu 80 (HTTP) lub 443 (szyfrowany HTTP), podczas gdy ruch mailowy korzysta z portu 25.

**robot internetowy** Działanie wyszukiwarki internetowej polegające na pobieraniu strony, a następnie wszystkich stron, do których prowadzą linki z danej strony itd., aż do momentu, gdy mają prawie wszystkie strony w internecie, których używają do zbudowania swojego indeksu wyszukiwania.

**web scraping** Sytuacja w której program podszywa się pod przeglądarkę internetową, pobiera stronę i przegląda jej zawartość. Często programy podążają za linkami na jednej stronie, by znaleźć następną, dzięki czemu mogą przemierzyć duży zbiór stron lub serwisy społecznościowe.

## 12.11. Ćwiczenia

**Ćwiczenie 1: Zmień program używający gniazda `socket1.py` tak, aby poprosić użytkownika o podanie adresu URL i by mógł on przeczytać dowolną stronę internetową. Możesz użyć `split('/')` aby rozbić URL na części składowe, dzięki czemu łatwo wyodrębnisz nazwę hosta dla metody `connect`. Dodaj sprawdzanie błędów przy użyciu `try` i `except`, tak aby poradzić sobie z przypadkiem, w którym użytkownik wprowadzi nieprawidłowo sformatowany lub nieistniejący adres URL.**

**Ćwiczenie 2: Zmień swój program używający gniazda tak, by zliczał liczbę otrzymanych znaków i przestawał wyświetlać jakikolwiek tekst po wyświetleniu 3000 znaków. Program powinien pobrać cały dokument, zliczyć całkowitą liczbę znaków oraz na końcu ją wyświetlić.**

**Ćwiczenie 3: Użyj `urllib` aby powtórzyć poprzednie ćwiczenie dotyczące (1) pobrania dokumentu z adresu URL, (2) wyświetlenia do 3000 znaków, (3) zliczenia całkowitej liczby znaków w dokumencie. Nie martw się o nagłówki, po prostu pokaż pierwsze 3000 znaków dokumentu.**

**Ćwiczenie 4: Zmień program `urllinks.py` tak, by wyodrębnić i policzyć znaczniki akapitu (`p`) z pobranego dokumentu HTML i wyświetlić liczbę akapitów jako wynik programu. Nie wyświetlaj tekstu akapitu, a jedynie je zliczaj. Przetestuj swój program na kilku niewielkich stronach internetowych, jak również na kilku większych.**

**Ćwiczenie 5: (Zaawansowane) Zmień program używający gniazda tak, by wyświetlał dane dopiero po otrzymaniu nagłówków i pustej linii. Pamiętaj, że `recv` otrzymuje znaki (wszystkie, włączając w to znaki końca linii), a nie linie.**

# Rozdział 13

# Korzystanie z usług sieciowych

Kiedy już pozyskiwanie dokumentów przez HTTP i ich przetwarzenie stało się łatwe, ludziom nie zajęło dużo czasu opracowanie podejścia, w którym zaczęli tworzyć dokumenty specjalnie zaprojektowane do użytku przez inne programy (tj. coś innego niż HTML, który ma być wyświetlany w przeglądarce).

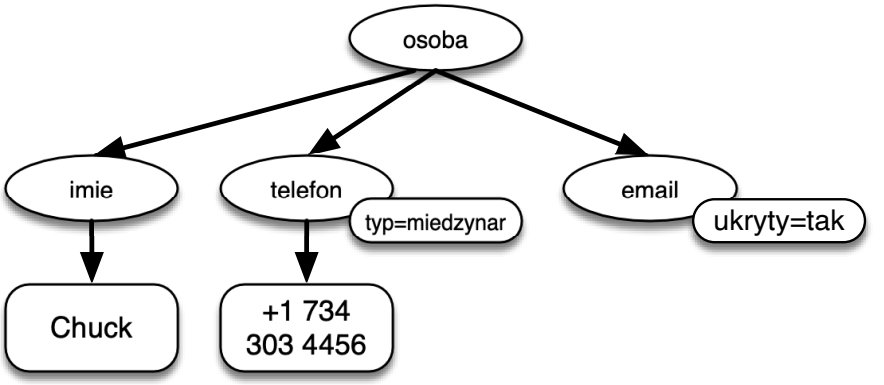
Istnieją dwa popularne formaty, których używamy przy wymianie danych w sieci. eXtensible Markup Language (XML, z ang. rozszerzalny język znaczników) jest używany od bardzo dawna i najlepiej nadaje się do wymiany danych w formie dokumentów. Gdy programy chcą tylko wymieniać ze sobą słowniki, listy lub inne wewnętrzne informacje, używają JavaScript Object Notation (JSON, z ang. notacja obiektowa JavaScript, patrz www.json.org). Przyjrzymy się obu formatom.

## 13.1. XML

XML wygląda bardzo podobnie do HTMLa, ale XML jest bardziej uporządkowany. Oto fragment dokumentu XML:

```
<osoba>
  <imie>Chuck</imie>
  <telefon typ="miedzynar">
    +1 734 303 4456
  </telefon>
  <email ukryty="tak" />
</osoba>
```

Każda para otwierająca (np. `<osoba>`) i zamykająca tagi (np. `</osoba>`) reprezentuje *element* lub *węzeł* (ang. node) o tej samej nazwie co znacznik (np. `osoba`). Każdy element może mieć jakiś tekst, pewne atrybuty (np. `ukryty`) i inne zagnieżdżone elementy. Jeśli element XML jest pusty (tzn. nie ma treści), to może być przedstawiony za pomocą samozamykającego się tagu (np. `<email />`).




Rysunek 13.1: Drzewiasta reprezentacja XMLa

Często pomocne jest myślenie o dokumencie XML jako o strukturze drzewa, w którym znajduje się najwyższy element (tutaj: `osoba`), a inne znaczniki (np. `telefon`) są oznaczane jako *dzieci* elmenetów będących ich *rodzicami*.

## 13.2. Parsowanie XML

Oto prosta aplikacja, która przetwarza część XMLa i wyodrębnia z niego niektóre elementy danych:

```
import xml.etree.ElementTree as ET

data = '''
<osoba>
  <imie>Chuck</imie>
  <telefon typ="intl">
    +1 734 303 4456
  </telefon>
  <email ukryty="tak" />
</osoba>'''

tree = ET.fromstring(data)
print('Imię:', tree.find('imie').text)
print('Attr:', tree.find('email').get('ukryty'))

# Kod źródłowy: https://pl.py4e.com/code3/xml1.py
```

Potrójny apostrof (`'''`), jak również potrójny cudzysłów (`"""`), pozwalają na tworzenie napisów, które obejmują wiele linii.

Wywołanie `fromstring` zamienia reprezentację XMLa z ciągu znaków na "drzewo" elementów XML. Gdy element XML znajduje się w drzewie, posiadamy szereg metod, które możemy wywołać by wyodrębnić interesujące nas dane z ciągu znaków będącego XMLem. Funkcja `find` przeszukuje drzewo XML i wyciąga element, który pasuje do określonego znacznika.

```
Imię: Chuck
Attr: tak
```

Użycie parsera XML takiego jak `ElementTree` ma tę zaletę, że choć XML w tym przykładzie jest dość prosty, to okazuje się, że istnieje wiele reguł dotyczących poprawności składni XML, a użycie `ElementTree` pozwala nam na wydobycie danych z XML bez martwienia się o te reguły.

## 13.3. Przechodzenie w pętli po węzłach

Często XML ma wiele węzłów i musimy napisać pętlę, tak by je wszystkie przetworzyć. Przyjmijmy, że mamy do przeanalizowania prostą bazę użytkowników, która jest zapisana w postaci XMLa. W poniższym programie przechodzimy w pętli przez wszystkie elementy o nazwie `user`:

```
import xml.etree.ElementTree as ET

input = '''
<stuff>
  <users>
    <user x="2">
      <id>001</id>
      <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
    </user>
  </users>
</stuff>'''

stuff = ET.fromstring(input)
lst = stuff.findall('users/user')
print('Liczba użytkowników:', len(lst))

for item in lst:
    print('Name', item.find('name').text)
    print('Id', item.find('id').text)
    print('Attribute', item.get('x'))

# Kod źródłowy: https://pl.py4e.com/code3/xml2.py
```

Metoda `findall` zwraca Pythonową listę poddrzew, które reprezentują struktury `user` w drzewie XML. Następnie możemy napisać pętlę `for`, która analizuje każdy węzeł o nazwie `user` i wypisuje jego elementy tekstowe `name` i `id` oraz atrybut `x`.

```
Liczba użytkowników: 2
Name Chuck
```

```
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

Ważne jest, by uwzględnić wszystkie elementy poziomu nadrzędnego w instrukcji `findall` z wyjątkiem elementu najwyższego poziomu (np. `users/user`). W przeciwnym razie, Python nie znajdzie żadnych szukanych przez nas węzłów.

```
import xml.etree.ElementTree as ET

input = '''
<stuff>
  <users>
    <user x="2">
      <id>001</id>
      <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
    </user>
  </users>
</stuff>'''

stuff = ET.fromstring(input)

lst = stuff.findall('users/user')
print('Liczba użytkowników:', len(lst))

lst2 = stuff.findall('user')
print('Liczba użytkowników:', len(lst2))
```

Zmienna `lst` przechowuje wszystkie elementy `user`, które są zagnieżdżone w ich rodzicu, tj. w `users`. Zmienna `lst2` szuka elementów `user`, które miałyby być zagnieżdżone w obrębie najwyższego poziomu, czyli `staff`, ale tam nie ma żadnego takiego węzła.

```
Liczba użytkowników: 2
Liczba użytkowników: 0
```

## 13.4. JSON

Format JSON został zainspirowany obiektowym i tablicowym formatem używanym w języku JavaScript. Ponieważ Python został wymyślony przed JavaScriptem, składnia Pythona dla słowników i list wpłynęła na składnię JSON. Format JSON jest więc prawie identyczny z połączeniem list i słowników Pythona.

Oto kodowanie JSON, które w przybliżeniu odpowiada prostemu XMLowi, który widzieliśmy wcześniej:

```
{
  "imie" : "Chuck",
  "telefon" : {
    "typ" : "miedzynar",
    "numer" : "+1 734 303 4456"
   },
   "email" : {
     "ukryty" : "tak"
   }
}
```

Zapewne zauważysz pewne różnice. Po pierwsze, w XMLu możemy dodać do znacznika "telefon" atrybuty takie jak "miedzynar". W JSON mamy po prostu pary klucz-wartość. Również znacznik XML "osoba" zniknął i został zastąpiony zestawem zewnętrznych nawiasów klamrowych.

Ogólnie rzecz biorąc, struktury JSON są prostsze niż XML, ponieważ JSON ma mniej możliwości niż XML. Ale JSON ma tę zaletę, że mapuje *bezpośrednio* do jakiegoś połączenia słowników i list. A ponieważ prawie wszystkie języki programowania mają coś w rodzaju słowników i list Pythona, JSON jest bardzo naturalnym formatem wymiany danych pomiędzy dwoma współpracującymi ze sobą programami.

JSON szybko staje się najczęściej wybieranym formatem dla prawie całej wymiany danych pomiędzy aplikacjami, właśnie ze względu na swoją relatywną prostotę w porównaniu do XMLa.

## 13.5. Parsowanie JSONa

Budujemy nasz JSON poprzez zagnieżdżanie słowników i list w zależności od naszych potrzeb. W poniższym przykładzie przedstawiamy listę użytkowników, gdzie każdy użytkownik jest zestawem par klucz-wartość (czyli słownikiem). Mamy więc listę słowników.

W poniższym programie wykorzystujemy wbudowaną bibliotekę `json` do parsowania JSONa i odczytywania z niego danych. Porównaj kod z wcześniejszym przykładem pracującym na XMLu. JSON ma mniej detali, więc musimy z góry wiedzieć, że otrzymujemy listę i że lista ta składa się z użytkowników, a każdy użytkownik jest zestawem par klucz-wartość. JSON jest bardziej zwięzły (zaleta), ale również mniej samoopisujący się (wada).

```
import json

data = '''
[
  { "id" : "001",
    "x" : "2",
```

```
    "name" : "Chuck"
  } ,
  { "id" : "009",
    "x" : "7",
    "name" : "Brent"
  }
]'''

info = json.loads(data)
print('Liczba użytkowników:', len(info))

for item in info:
    print('Name', item['name'])
    print('Id', item['id'])
    print('Attribute', item['x'])

# Kod źródłowy: https://pl.py4e.com/code3/json2.py
```

Jeśli porównasz kod wyodrębniający dane z JSONa i XMLa, to zauważysz w tym przykładzie, że to, co otrzymujemy z `json.loads()`, jest listą Pythona, po której przechodzimy pętlą `for`, a każda pozycja na tej liście jest słownikiem Pythona. Kiedy już JSON zostanie przeparsowany, możemy użyć operatora indeksu, tak by wyodrębnić różne fragmenty danych związane z użytkownikiem. Nie musimy korzystać z biblioteki JSON by przekopać się przez parsowanego JSONa, ponieważ zwracane dane są po prostu natywnymi strukturami Pythona.

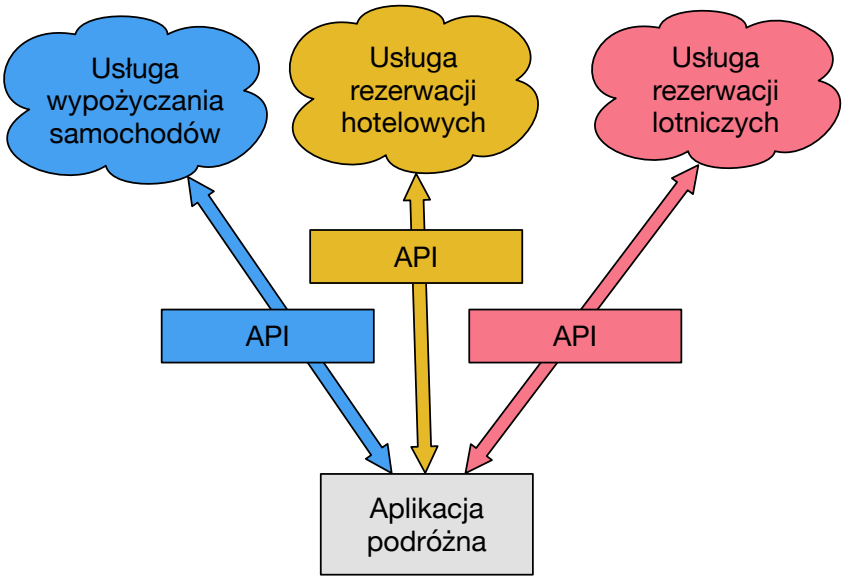
Wynik powyższego programu jest dokładnie taki sam jak wersji z XMLem.

```
Liczba użytkowników: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

Ogólnie rzecz biorąc, w obrębie usług sieciowych istnieje branżowy trend polegający na odchodzeniu od XMLa i zmierzaniu w kierunku JSONa. JSON jest prostszy i bardziej bezpośrednio mapuje do natywnych struktur danych, które mamy już w językach programowania, więc kod parsujący JSON i wyodrębniający dane jest w takim przypadku zazwyczaj prostszy i bardziej czytelny. Z drugiej strony, XML jest bardziej samoopisujący się niż JSON, a więc są pewne zastosowania, w których XML zachowuje przewagę. Na przykład większość edytorów tekstu przechowuje dokumenty wewnętrznie przy użyciu XMLa, a nie JSONa.

## 13.6. API – Interfejsy programowania aplikacji

Posiadamy teraz możliwość wymiany danych pomiędzy aplikacjami przy użyciu protokołu HTTP oraz sposób na reprezentowanie skomplikowanych danych, które wysyłamy tam i z powrotem pomiędzy aplikacjami, przy użyciu XMLa lub JSONa.




Rysunek 13.2: SOA – architektura zorientowana na usługi

Następnym krokiem jest rozpoczęcie definiowania i dokumentowania "umów" pomiędzy aplikacjami wykorzystującymi powyższe techniki. Ogólna nazwa takiej umowy pomiędzy aplikacjami to *interfejsy programowania aplikacji* (ang. Application Program Interfaces, API). Kiedy używamy API, zazwyczaj jeden z programów udostępnia zestaw *usług* do wykorzystania przez inne aplikacje i publikuje API (tzn. "zasady"), których należy przestrzegać, by uzyskać dostęp do usług świadczonych przez ten program.

Gdy zaczynamy budować nasze programy, w których działanie obejmuje dostęp do usług świadczonych przez inne programy, nazywamy to podejście *architekturą zorientowaną na usługi* (ang, Service-Oriented Architecture, SOA). Podejście SOA to takie, w którym cała nasza aplikacja korzysta z usług innych aplikacji. Podejście inne niż SOA to takie, w którym aplikacja jest pojedynczą, samodzielną aplikacją, która zawiera cały kod niezbędny do jej implementacji i wdrożenia.

Podczas korzystania z sieci widzimy wiele przykładów SOA. Możemy wejść na stronę internetową i zarezerwować podróże lotnicze, hotele i wypożyczyć samochody, a wszystko to z poziomu jednej strony. Dane dotyczące hoteli nie są przechowywane na serwerach linii lotniczych. Zamiast tego, serwery linii lotniczych kontaktują się z usługami znajdującymi się na serwerach rezerwacji hotelowychm pobierają dane dotyczące hoteli i przedstawiają je użytkownikowi. Kiedy użytkownik wyraża zgodę na dokonanie rezerwacji hotelowej za pomocą strony internetowej linii lotniczych, strona ta korzysta z innej usługi internetowej w systemach rezerwacji hotelowych, tak aby faktycznie dokonać tej rezerwacji. A gdy przychodzi czas, by obciążyć Twoją kartę płatniczą za całą transakcję, w proces ten włączają się jeszcze inne serwery.

Architektura zorientowana na usługi ma wiele zalet, w tym: (1) zawsze przechowujemy tylko jedną kopię danych (jest to szczególnie ważne w przypadku rezerwacji hotelowych, w których nie chcemy wykonywać nadmiernych i niewykonalnych zobowiązań) oraz (2) właściciele danych mogą ustalić zasady korzystania z ich

danych. Dzięki tym zaletom, system SOA musi być starannie zaprojektowany, tak aby działał wydajnie i spełniał potrzeby użytkownika.

Kiedy aplikacja udostępnia przez internet zestaw usług w swoim API, nazywamy to *usługą sieciową* (ang. web service).

## 13.7. Bezpieczeństwo i korzystanie z API

Dość często zdarza się, że chcąc skorzystać z API jakiegoś dostawcy, musisz posiadać tzw. klucz API. Ogólna idea jest taka, że dostawcy usług chcą wiedzieć, kto korzysta z ich usług i w jakim stopniu ich używa. Być może mają oni darmowe i płatne wersje swoich usług lub mają politykę ograniczającą liczbę zapytań, które pojedyncza osoba może wysłać w danym odcinku czasu.

Czasami jest tak, że po otrzymaniu klucza API po prostu dołączasz go jako część danych w zapytaniu POST lub jako parametr w adresie URL podczas wywoływania API.

Innym razem dostawca usług chce mieć większą pewność co do źródła zapytań i z tego powodu oczekuje, że będziesz wysyłał kryptograficznie podpisane wiadomości z wykorzystaniem tzw. współdzielonego klucza i sekretu. Bardzo popularną technologią, która jest używana do podpisywania tego rodzaju zapytań, jest *OAuth*. Więcej o tym protokole możesz przeczytać na stronie www.oauth.net.

Na szczęście istnieje wiele wygodnych i darmowych bibliotek OAuth, dzięki czemu możesz uniknąć implementowania OAuth od zera. Biblioteki mają różny stopień skomplikowania i możliwości, a informacje o nich znajdują się na stronie internetowej OAuth.

## 13.8. Słowniczek

**API** Interfejs programowania aplikacji. Umowa pomiędzy aplikacjami, która określa schematy interakcji pomiędzy dwoma komponentami tych aplikacji.

**ElementTree** Wbudowana biblioteka Pythona służąca do parsowania danych XML.

**JSON** Notacja obiektowa JavaScript. Format, który pozwala na oznakowanie ustrukturyzowanych danych w oparciu o składnię obiektów JavaScript.

**SOA** Architektura zorientowana na usługi. Sytuacja, w której aplikacja składa się z połączonych w sieci komponentów.

**XML** Rozszerzalny język znaczników. Format, który pozwala na oznakowanie danych strukturalnych.

## 13.9. Aplikacja nr 1: usługa sieciowa OpenStreetMap do geokodowania

OpenStreetMap posiada usługę internetową Nominatim, który pozwala nam korzystać z ich dużej bazy danych geograficznych. Możemy przesłać do ich geokodującego

API ciąg znaków geograficznych, takich jak "Ann Arbor, MI", i sprawić, że OpenStreetMap zwróci swoje najbardziej prawdopodobne przypuszczenie dotyczące tego, gdzie na mapie możemy odnaleźć nasz ciąg znaków.

Usługa geokodowania jest bezpłatna, ale ograniczona ilościowo, więc w aplikacji komercyjnej nie możesz korzystać z API w sposób nieograniczony. Ale jeśli np. masz jakieś dane z ankiety, w której użytkownicy wprowadzili w polu danych jakąś lokalizację w dowolnym formacie, to możesz użyć tego API, tak by całkiem dobrze oczyścić swoje dane.

*Używając bezpłatnego API, takiego jak API geokodowania OpenStreetMap, korzystanie z zasobów odbywa się z poszanowaniem pewnych zasad. Jeśli zbyt wiele osób będzie nadużywać danej usługi, to np. OpenStreetMap może zrezygnować z udostępniania bezpłatnej wersji lub znacznie ją ograniczyć.*

Możesz zapoznać się z dokumentacją online dotyczącą tej usługi[1], ale jest ona na tyle prosta, że możesz ją nawet przetestować za pomocą przeglądarki, wpisując w niej następujący adres URL:

https://nominatim.openstreetmap.org/search.php?q=Ann%20Arbor%2C%20MI&format=geojson&limit=1

Usługa zwraca nam dane w formacie GeoJSON. Jest to otwarty standard formatu danych przeznaczony do przedstawiania prostych obiektów geograficznych wraz z ich atrybutami nieprzestrzennymi i bazuje on na JSONie.

Poniżej znajduje się prosta aplikacja zachęcająca użytkownika do wyszukania informacji geograficznych o wprowadzonej nazwie miejsca. Aplikacja wywołuje API geokodowania OpenStreetMap i pobiera informacje ze zwróconego JSONa.

```
import urllib.request, urllib.parse, urllib.error
import json
import ssl

serviceurl = 'https://nominatim.openstreetmap.org/search.php?'

# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    address = input('Podaj nazwę miejsca: ')
    if len(address) < 1: break

    parms = dict()
    parms['q'] = address
    parms['format'] = 'geojson'
    parms['limit'] = 1

    url = serviceurl + urllib.parse.urlencode(parms)
```

[1] https://nominatim.org/release-docs/develop/api/Search/

```
    print('Pobieranie', url)
    uh = urllib.request.urlopen(url, context=ctx)
    data = uh.read().decode()
    print('Pobrano', len(data), 'znaków')

    try:
        js = json.loads(data)
    except:
        js = None

    if not js or 'features' not in js or len(js['features']) == 0:
        print('==== Błąd pobierania ====')
        print(data)
        continue

    print(json.dumps(js, indent=4))

    lng = js['features'][0]['geometry']['coordinates'][0]
    lat = js['features'][0]['geometry']['coordinates'][1]
    print('szer. geogr.', lat, 'dł. geogr.', lng)
    location = js['features'][0]['properties']['display_name']
    print(location)

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

Program pobiera ciąg znaków i konstruuje adres URL zawierający wprowadzony ciąg wyszukiwania jako prawidłowo zakodowany parametr adresu URL, a następnie używa `urllib` do pobrania tekstu z API geokodowania OpenStreetMap. W przeciwieństwie do stałej/niezmiennej strony internetowej, otrzymywane dane zależą od parametrów, które wysyłamy, oraz od danych geograficznych przechowywanych na serwerach OpenStreetMap.

Kiedy pobierzemy dane JSON, parsujemy je za pomocą biblioteki `json` i sprawdzamy kilka rzeczy, tak aby upewnić się, że otrzymaliśmy dobre dane, a następnie pobieramy informacje, których szukamy.

Wynik działania programu jest następujący (część zwróconych danych JSON została usunięta):

```
$ python3 geojson.py
Podaj nazwę miejsca: Ann Arbor, MI
Pobieranie https://nominatim.openstreetmap.org/search.php?q=...
Pobrano 587 znaków

{
    "type": "FeatureCollection",
    "licence": "Data \u00a9 OpenStreetMap contributors, ODbL 1.0.
    ↪  https://osm.org/copyright",
    "features": [
        {
            "type": "Feature",
```

```
            "properties": {
                "place_id": 146750,
                "osm_type": "node",
                "osm_id": 27023455,
                "display_name": "Ann Arbor, Washtenaw County,
                ↪  Michigan, 48104, United States of America",
                "place_rank": 16,
                "category": "place",
                "type": "city",
                "importance": 0.837069344370284,
                "icon": "https://nominatim.openstreetmap.org/..."
            },
            "bbox": [
                -83.8912291,
                42.1081569,
                -83.5712291,
                42.4281569
            ],
            "geometry": {
                "type": "Point",
                "coordinates": [
                    -83.7312291,
                    42.2681569
                ]
            }
        }
    ]
}
szer. geogr. 42.2681569 dł. geogr. -83.7312291
Ann Arbor, Washtenaw County, Michigan, 48104, United States of
↪  America

Podaj nazwę miejsca:
```

Możesz pobrać pl.py4e.com/code3/geoxml.py, aby sprawdzić wariant XML w API geokodowania OpenStreetMap.

**Ćwiczenie 1: Zmień `geojson.py` lub `geoxml.py` w celu wypisania nazwy stanu z pobranych danych (jeżeli podane miejsce jest w USA). Dodaj sprawdzanie błędów, tak aby Twój program nie wyświetlał danych z mechanizmu traceback w momencie gdy w danych nie ma nazwy stanu. Gdy kod już zacznie działać, wyszukaj "Atlantic Ocean" i upewnij się, że obsługuje on lokalizacje, które nie znajdują się w żadnym kraju.**

## 13.10. Aplikacja nr 2: Twitter

O czasu gdy API Twittera stało się coraz bardziej wartościowe, Twitter przeszedł z otwartego i publicznego API do takiego, które przy każdym

Nasze poniższe dwa przykłady opierają się na plikach *twurl.py*, *hidden.py*, *oauth.py*, *twitter1.py* i *twitter2.py*, które możesz pobrać z pl.py4e.com/code3; po pobraniu plików, umieść je wszystkie w jednym katalogu.

Aby móc korzystać z tych programów, musisz mieć konto na Twitterze i autoryzować swój kod Pythona jako aplikację, ustawić klucz, sekret, token i sekret tokena. Wyedytuj plik *hidden.py* i umieść w nim cztery ciągi znaków w odpowiednich miejscach:

```
# Przechowuj ten plik oddzielnie

# 1. Zaloguj się lub utworz konto na https://twitter.com
# 2. Wejdź na stronę https://developer.twitter.com/en/apps
#    i spróbuj utworzyć nową aplikację ("Create an app"):
#    a) zostaniesz poproszony o utworzenie konta deweloperskiego
#       (Please apply for a Twitter developer account);
#    b) kliknij na "Apply"
#    c) wybierz "Hobbyist" > "Exploring the API" i wybierz "Next"
#    d) uzupełnij formularz (może być wymagana weryfikacja numeru
↪  telefonu)
#    e) w kolejnym kroku opisz po angielsku jak zamierzasz
↪  wykorzystać dane
#       Twittera (wspomnij, że uczysz się Pythona, chcesz
↪  dowiedzieć
#       się jak korzystać z API Twittera, zamierzasz napisać dwa
↪  mini-projekty,
#       które w interaktywny sposób pobierają nazwę konta Twittera i
↪  dla podanego
#       konta w projekcie nr 1 pobierzesz i wyświetlisz oś czasu
↪  użytkownika
#       zwróconą w formacie JSON, a w projekcie nr 2 pobierszesz
↪  jego listę jego
#       znajomych w formacie JSON i przeparsujesz ją w celu
↪  wyciągnięcia informacji
#       o znajomych; opiszę tę sekcję w miarę szczegółowo, gdyż
↪  wpływa ona
#       na akceptację Twojego zgłoszenia); w sekcji "Specifics"
↪  zaznacz wszystko
#       na "No"
#    f) zweryfikuj zgłoszenie, przeczytaj warunki umowy i wyślij
↪  zgłoszenie
#    g) potwierdź zgłoszenie klikając na link w wiadomości
↪  mailowej,
#       którą otrzymasz po wysłaniu zgłoszenia
# 3. Gdy Twoje zgłoszenie zostanie zaakceptowane, wejdź jeszcze raz
#    na https://developer.twitter.com/en/apps i ponownie spróbuj
#    utworzyć nową aplikację ("Standalone Apps" > "Create App")
# 4. Utwórz aplikację, w karcie "Keys and tokens" wygeneruj klucze i
↪  podmień
#    poniżej cztery ciągi znaków dotyczące consumer_key,
↪  consumer_secret,
```

```
#    token_key i token_secret (podmień wszystko co znajduje się
↪  między
#    cudzysłowami, łącznie z nawiasami ostrokątnymi)

def oauth():
    return {"consumer_key":    "<tutaj umieść Consumer Keys - API
    ↪  key>",
            "consumer_secret": "<tutaj umieść Consumer Keys - API
            ↪  key secret>",
            "token_key":       "<tutaj umieść Authentication Tokens
            ↪  - Access Token & Secret - Access token>",
            "token_secret":    "<tutaj umieść Authentication Tokens
            ↪  - Access Token & Secret - Access token secret>"}

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

Usługa sieciowa Twittera jest dostępna za pomocą podobnego do poniższego adresu URL:

https://api.twitter.com/1.1/statuses/user_timeline.json

Jednak po dodaniu wszystkich informacji dotyczących uwierzytelniania, adres URL będzie wyglądał mniej więcej tak:

```
https://api.twitter.com/1.1/statuses/user_timeline.json?count=2
&oauth_version=1.0&oauth_token=101...SGI&screen_name=drchuck
&oauth_nonce=09239679&oauth_timestamp=1380395644
&oauth_signature=rLK...BoD&oauth_consumer_key=h7Lu...GNg
&oauth_signature_method=HMAC-SHA1
```

Jeśli chcesz dowiedzieć więcej o znaczeniu różnych parametrów, które są dodawane w celu spełnienia wymogów bezpieczeństwa OAuth, to możesz przeczytać specyfikację OAuth.

W naszych programach korzystających z Twittera, wszystkie skomplikowane mechanizmy będą ukryte w plikach *oauth.py* i *twurl.py*. Po prostu ustawiamy sekrety w *hidden.py*, następnie wysyłamy żądany adres URL do funkcji *twurl.augment()*, a kod biblioteki dodaje za nas wszystkie niezbędne parametry do adresu URL.

Poniższy program pobiera oś czasu dla konkretnego użytkownika Twittera i zwraca go nam w formacie JSON w postaci ciągu znaków. Po uzyskaniu danych po prostu wypisujemy pierwsze 250 znaków ciągu:

```
import urllib.request, urllib.parse, urllib.error
import twurl
import ssl

# https://developer.twitter.com/en/apps
# Utwórz aplikację i wstaw w hidden.py cztery ciągi znaków dotyczące
↪  OAuth

TWITTER_URL =
↪  'https://api.twitter.com/1.1/statuses/user_timeline.json'
```

```
# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    print('')
    acct = input('Podaj nazwę konta na Twitterze: ')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '2'})
    print('Pobieranie', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()
    print(data[:250])
    headers = dict(connection.getheaders())
    # wypisz nagłówki
    print('Pozostało', headers['x-rate-limit-remaining'])

# Kod źródłowy: https://pl.py4e.com/code3/twitter1.py
```

Po uruchomieniu programu uzyskamy mniej więcej następujący wynik:

```
Podaj nazwę konta na Twitterze: drchuck
Pobieranie https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 17:30:25 +0000 2013","
id":384007200990982144,"id_str":"384007200990982144",
"text":"RT @fixpert: See how the Dutch handle traffic
intersections: http:\/\/t.co\/tIiVWtEhj4\n#brilliant",
"source":"web","truncated":false,"in_rep
Pozostało 178

Podaj nazwę konta na Twitterze: fixpert
Pobieranie https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 18:03:56 +0000 2013",
"id":384015634108919808,"id_str":"384015634108919808",
"text":"3 months after my freak bocce ball accident,
my wedding ring fits again! :)\n\nhttps:\/\/t.co\/2XmHPx7kgX",
"source":"web","truncated":false,
Pozostało 177

Podaj nazwę konta na Twitterze:
```

Wraz z danymi zwróconymi z osi czasu, Twitter zwraca również w nagłówkach odpowiedzi HTTP metadane dotyczące żądania. W szczególności jeden nagłówek, `x-rate-limit-remaining`, informuje nas o tym ile jeszcze żądań możemy wysłać, zanim zostaniemy zablokowani na pewien krótki czas. Jak widać, nasze kolejne wywołania API powodują zmniejszenie tej wartości o jeden.

W poniższym przykładzie, pobieramy znajomych użytkownika na Twitterze, analizujemy zwrócony JSON i wyodrębniamy niektóre informacje o znajomych. Po

przeparsowaniu JSONa do postaci list i słowników, ponownie zamieniamy go do zwykłej postaci tekstowej, ale tym razem nadajemy mu po dwie spacje na wcięcie i "ładnie" wypisujemy na ekranie, tak aby umożliwić nam dalsze zgłębianie otrzymanych danych (co może być przydatne jeśli będziemy chcieli wyodrębnić więcej pól).

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
import ssl

# https://developer.twitter.com/en/apps
# Utwórz aplikację i wstaw w hidden.py cztery ciągi znaków dotyczące
↪  OAuth

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    print('')
    acct = input('Podaj nazwę konta na Twitterze: ')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '5'})
    print('Pobieranie', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()

    js = json.loads(data)
    print(json.dumps(js, indent=2))

    headers = dict(connection.getheaders())
    print('Pozostało', headers['x-rate-limit-remaining'])

    for u in js['users']:
        print(u['screen_name'])
        if 'status' not in u:
            print('   * Nie odnaleziono klucza "status"')
            continue
        s = u['status']['text']
        print('  ', s[:50])

# Kod źródłowy: https://pl.py4e.com/code3/twitter2.py
```

Ponieważ JSON staje się zestawem zagnieżdżonych list Pythona i słowników, to przy pomocy niewielkiej ilości kodu możemy użyć kombinacji operacji indeksowania i pętli `for` do przejścia po zwróconych strukturach danych.

Wynik programu wygląda następująco (niektóre elementy danych zostały skrócone, tak aby zmieścić wszystko na stronie):

```
Podaj nazwę konta na Twitterze: drchuck
Pobieranie https://api.twitter.com/1.1/friends ...
Pozostało 14

{
  "next_cursor": 1444171224491980205,
  "users": [
    {
      "id": 662433,
      "followers_count": 28725,
      "status": {
        "text": "@jazzychad I just bought one .__.",
        "created_at": "Fri Sep 20 08:36:34 +0000 2013",
        "retweeted": false,
      },
      "location": "San Francisco, California",
      "screen_name": "leahculver",
      "name": "Leah Culver",
    },
    {
      "id": 40426722,
      "followers_count": 2635,
      "status": {
        "text": "RT @WSJ: Big employers like Google ...",
        "created_at": "Sat Sep 28 19:36:37 +0000 2013",
      },
      "location": "Victoria Canada",
      "screen_name": "_valeriei",
      "name": "Valerie Irvine",
    }
  ],
 "next_cursor_str": "1444171224491980205"
}

leahculver
   @jazzychad I just bought one .__.
_valeriei
   RT @WSJ: Big employers like Google, AT&amp;amp;T are h
ericbollens
   RT @lukew: sneak peek: my LONG take on the good &a
halherzog
   Learning Objects is 10. We had a cake with the LO,
scweeker
   @DeviceLabDC love it! Now where so I get that "etc

Podaj nazwę konta na Twitterze:
```

Ostatni fragment wyniku programu jest taki, że widzimy pętlę `for` odczytującą pięciu ostatnich "znajomych" konta *@drchuck* na Twitterze i wypisującą najnowszy

status dla każdego znajomego. W zwróconym JSONie znajduje się dużo więcej danych. Jeśli spojrzysz na wynik programu, to zobaczysz również, że operacja "znalezienia znajomych" danego konta ma inny limit wywołań niż dozwolona liczba zapytań dotycząca osi czasu.

Korzystanie z bezpiecznych kluczy API pozwala Twitterowi mieć solidną pewność, że wie, kto korzysta z jego API i danych oraz w jakim zakresie. Podejście oparte na ustanawianiu limitów zapytań pozwala nam na proste wyszukiwanie danych osobowych, ale nie pozwala nam na zbudowanie produktu, który pobierałby przez API Twittera jakieś dane miliony razy dziennie.

# Rozdział 14

# Programowanie obiektowe

## 14.1. Zarządzanie większymi programami

Na początku książki omówiliśmy cztery podstawowe wzorce programowania, których używamy do tworzenia programów:

- Kod sekwencyjny
- Kod warunkowy (instrukcje `if`)
- Kod powtarzalny (pętle)
- Zapisanie i ponowne użycie (funkcje)

W późniejszych rozdziałach analizowaliśmy proste zmienne, jak również struktury danych, takie jak listy, krotki i słowniki.

Podczas tworzenia programów, projektujemy struktury danych i piszemy kod służący do operowania tymi strukturami danych. Istnieje wiele sposobów pisania programów i prawdopodobnie do tej pory napisałeś już kilka programów, które nie są "zbyt eleganckie", oraz kilka innych programów, które są "bardziej eleganckie". Mimo to, że Twoje programy mogą być małe, to zapewne zaczynasz dostrzegać, że pisanie kodu wymaga odrobiny sztuki i estetyki.

W miarę jak programy stają się długie na miliony wierszy, coraz ważniejsze staje się pisanie kodu, który jest łatwy do zrozumienia. Jeśli pracujesz nad programem o długości miliona linii, nigdy nie możesz mieć w tym samym czasie w głowie całego programu. Potrzebujemy sposobów, aby rozbić duże programy na wiele mniejszych kawałków, tak abyśmy mieli mniej kodu do przeglądania podczas rozwiązywania jakiegoś problemu, naprawiania błędów lub dodawania nowych funkcji.

W pewnym sensie, programowanie obiektowe jest sposobem na uporządkowanie kodu tak, abyś mógł skupić się na jego 50 liniach i go zrozumieć, ignorując na chwilę pozostałe 999 950 linii kodu.

## 14.2. Rozpoczęcie pracy

Podobnie jak w przypadku wielu innych aspektów programowania, konieczne jest poznanie koncepcji programowania obiektowego, zanim będzie można je skutecznie wykorzystać. Powinieneś podejść do tego rozdziału jako do sposobu na poznanie niektórych terminów i pojęć oraz powinieneś popracować z kilkoma prostymi przykładami, aby stworzyć podstawy do dalszej nauki.

Kluczowym rezultatem tego rozdziału jest podstawowe zrozumienie, jak konstruowane są obiekty i jak one funkcjonują, a co najważniejsze, jak wykorzystujemy możliwości obiektów, które są nam dostarczane przez Pythona i jego biblioteki.

## 14.3. Korzystanie z obiektów

Jak się okazuje, w tej książce przez cały czas używaliśmy obiektów. Python dostarcza nam wiele wbudowanych obiektów. Oto prosty kod, w którym kilka pierwszych linii powinno być dla Ciebie w pewien sposób naturalnych i bardzo łatwych do zrozumienia.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Kod źródłowy: https://pl.py4e.com/code3/party1.py
```

Zamiast skupiać się na tym, co te linie ostatecznie realizują, spójrzmy na to, co *naprawdę* dzieje się w tym kodzie z punktu widzenia programowania obiektowego. Nie martw się, jeśli poniższe akapity nie mają dla Ciebie żadnego sensu gdy pierwszy raz je czytasz, ponieważ nie zdefiniowaliśmy jeszcze wszystkich tych terminów.

Pierwsza linia *konstruuje* obiekt typu `list`, druga i trzecia *wywołuje metodę* `append()`, czwarta linia wywołuje metodę `sort()`, a piąta linia *pobiera* (*wyszukuje*) element z pozycji 0.

Szósta linia wywołuje metodę `__getitem__()` na liście `stuff` z parametrem zero.

```
print (stuff.__getitem__(0))
```

Siódma linia jest jeszcze bardziej rozwlekłym sposobem na odzyskanie elementu z zerowej pozycji listy `stuff`.

```
print (list.__getitem__(stuff,0))
```

W powyższym fragmencie kodu wywołujemy metodę `__getitem__` w klasie `list` i *przekazujemy* jako parametry listę oraz pozycję elementu, który chcemy pobrać z tej listy.

Ostatnie trzy wiersze programu są równoważne, ale wygodniej jest po prostu użyć składni używającej nawiasów kwadratowych, tak aby wyszukać element znajdujący się na konkretnym miejscu listy.

Możemy przyjrzeć się możliwościom danego obiektu, patrząc na wynik funkcji `dir()`:

```
>>> stuff = list()
>>> dir(stuff)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

Pozostała część tego rozdziału zdefiniuje wszystkie powyższe terminy, więc pamiętaj by po jego zakończeniu wrócić do tej sekcji i ponownie przeczytać powyższe akapity, tak aby sprawdzić czy je rozumiesz.

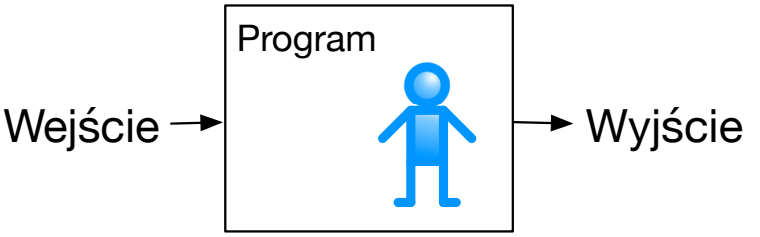
## 14.4. Zaczynając od programów...

Program w swojej najbardziej podstawowej formie pobiera pewną ilość danych wejściowych, przetwarza je i wytwarza pewną ilość danych wyjściowych. Nasz program do konwersji numerów pięter pokazuje bardzo krótki, ale kompletny program pokazujący wszystkie te trzy kroki.

```
usf = input('Wprowadź numer piętra w zapisie amerykańskim: ')
wf = int(usf) - 1
print('Numer piętra w zapisinie nie amerykańskim to', wf)

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

Jeśli zastanowimy trochę dłużej nad tym programem, to zauważymy, że istnieje "świat zewnętrzny" oraz nasz program. Aspekty wejściowe i wyjściowe są tymi miejcami, w których program wchodzi w interakcję ze światem zewnętrznym. Wewnątrz programu mamy kod i dane do wykonania zadania, do którego jest przeznaczony ten program.

Jednym ze sposobów na myślenie o programowaniu obiektowym jest rozdzielenie naszego programu na wiele "stref". Każda strefa zawiera pewien kod i dane (jak




Rysunek 14.1: Program

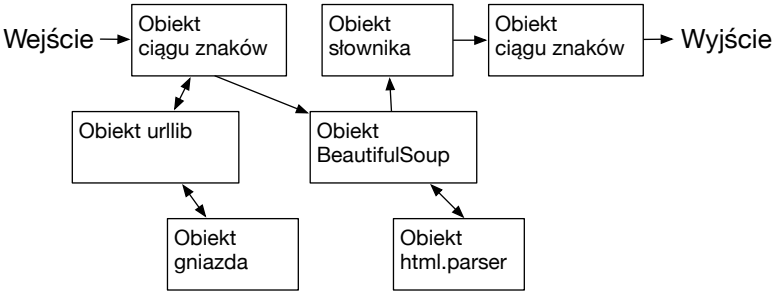
program) oraz ma dobrze zdefiniowane interakcje ze światem zewnętrznym i innymi strefami w programie.

Jeśli spojrzymy ponownie na aplikację do wyodrębniania linków, w której korzystaliśmy z biblioteki BeautifulSoup, to możemy zobaczyć program, który jest skonstruowany poprzez złączenie różnych obiektów w celu wykonania tego zadania:

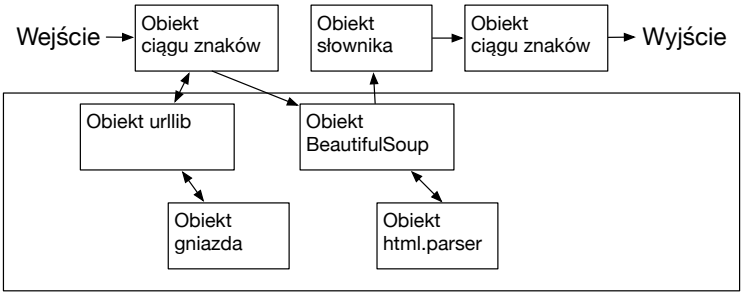
```
# Aby uruchomić poniższy kod, poprzez wiersz linii
# poleceń zainstaluj bibliotekę BeautifulSoup:
#
#    pip3 install beautifulsoup4
#

import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup
import ssl

# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Podaj link - ')
html = urllib.request.urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')

# Pobierz wszystkie znaczniki hiperłączy
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

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

Wczytujemy adres URL do zmiennej przchowującej ciąg znaków, a następnie przekazujemy ją do `urllib`, tak aby pobrać dane z sieci. Biblioteka `urllib` wykorzystuje w rzeczywistości bibliotekę `socket` do nawiązania połączenia z siecią w celu pobrania danych. Bierzemy ciąg znaków, który zwraca `urllib`, i przekazujemy go do BeautifulSoup do dalszej analizy. BeautifulSoup korzysta z obiektu `html.parser`[1]

[1] https://docs.python.org/3/library/html.parser.html




Rysunek 14.2: Program jako sieć obiektów




Rysunek 14.3: Pomijanie szczegółów podczas używania obiektu

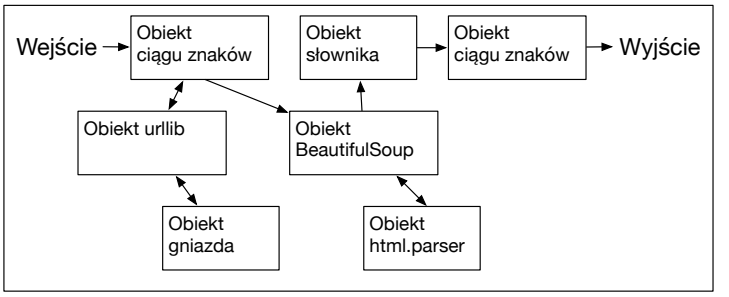
i w wyniku też zwraca nam obiekt. Na zwróconym obiekcie wywołujemy metodę `tags()`, która zwraca słownik obiektów będących znacznikami. Przechodzimy w pętli po znacznikach i dla każdego znacznika wywołujemy metodę `get()`, tak aby wypisać jego atrybut `href`.

Możemy narysować dla tego programu diagram z oznaczeniem jak te obiekty ze sobą współpracują.

W tym przypadku kluczowe nie jest idealne zrozumienie jak ten program działa, ale zobaczenie jak w celu stworzenia programu budujemy sieć współdziałających ze sobą obiektów i zarządzamy pomiędzy nimi przepływem informacji. Ważne jest również zauważenie, że gdy kilka rozdziałów temu natrafiłeś na ten program, to mogłeś w pełni zrozumieć co się w nim dzieje, nawet nie zdając sobie sprawy z tego, że program "zarządzał przepływem danych pomiędzy obiektami". To były po prostu tylko linie kodu, które wykonały dane zadanie.

## 14.5. Dzielenie problemu na mniejsze podproblemy

Jedną z zalet podejścia obiektowego jest to, że może ono ukryć złożoność jakiegoś problemu. Na przykład, podczas gdy my musimy wiedzieć jak używać biblioteki `urllib` i BeautifulSoup, nie musimy wiedzieć jak w środku one działają. Pozwala nam to skupić się na tej części problemu, którą musimy rozwiązać, i pominąć pozostałe części programu.




Rysunek 14.4: Pomijanie szczegółów podczas budowania obiektu

Możliwość skupienia się wyłącznie na tej części programu, na której nam zależy, i pomijanie reszty, jest również pomocna dla twórców używanych przez nas obiektów. Na przykład, programiści tworzący BeautifulSoup nie muszą wiedzieć ani dbać o to, w jaki sposób pobieramy naszą stronę HTML, jakie części chcemy przeczytać lub co planujemy zrobić z danymi, które pobieramy ze strony.

## 14.6. Nasz pierwszy obiekt w Pythonie

W podstawowym zakresie, obiektem to po prostu jakiś kod plus struktury danych, które są mniejsze niż cały program. Zdefiniowanie funkcji pozwala nam na zapisanie trochę kodu i nadanie mu nazwy, a następnie wywołanie tego kodu za pomocą nazwy funkcji.

Obiekt może zawierać szereg funkcji (które nazywamy *metodami*), jak również dane, które są wykorzystywane przez te funkcje. Dane będące częścią obiektu nazywamy *atrybutami*.

Używamy słowa kluczowego `class` do zdefiniowania danych i kodu, które będą składały się na każdy z tych obiektów. Słowo kluczowe `class` zawiera nazwę klasy i rozpoczyna wcięty blok kodu, w którym umieszczamy atrybuty (dane) i metody (kod).

```
class PartyAnimal:
   x = 0

   def party(self) :
     self.x = self.x + 1
     print("Jak na razie", self.x)

an = PartyAnimal()
an.party()
an.party()
an.party()
PartyAnimal.party(an)

# Kod źródłowy: https://pl.py4e.com/code3/party2.py
```

Rysunek 14.5: Klasa i dwa obiekty

Każda metoda wygląda jak funkcja, zaczynająca się od słowa kluczowego `def` i składająca się z wciętego bloku kodu. Nasz obiekt ma jeden atrybut (`x`) i jedną metodę (`party`). Metody te mają specjalny pierwszy parametr, który nazywamy umownie `self`.

Tak jak słowo kluczowe `def` nie powoduje wykonania kodu funkcji, tak samo słowo kluczowe `class` nie tworzy obiektu. Zamiast tego, słowo kluczowe `class` definiuje szablon mówiący o tym jakie dane i kod będą zawarte w każdym obiekcie typu `PartyAnimal`. Klasa jest jak foremka do wykrawania ciastek, a obiekty tworzone przy jej użyciu to ciasteczka[2]. Nie umieszczasz lukru na foremce do wykrawania ciastek; lukier umieszczasz na ciasteczkach, a na każdym ciasteczku możesz umieścić inny lukier.

Kontynuując analizę naszego przykładowego program, widzimy pierwszą wykonywalną linię kodu:

```
an = PartyAnimal()
```

Tutaj instruujemy Pythona, by skonstruował (tzn. stworzył) *obiekt* lub *instancję* klasy `PartyAnimal`. Wygląda to jak wywołanie funkcji o nazwie klasy. Python konstruuje obiekt z odpowiednimi danymi i metodami i zwraca obiekt, który jest następnie przypisany do zmiennej `an`. W pewnym sensie jest to dość podobne do poniższej linii, której używaliśmy już wcześniej:

```
counts = dict()
```

Tutaj instruujemy Pythona, by skonstruował obiekt przy użyciu szablonu `dict` (obecnego już w Pythonie), zwrócił instancję słownika i przypisał ją do zmiennej `counts`.

Klasa `PartyAnimal` jest używana do konstruowania obiektu, natomiast zmienna `an` jest używana do wskazywania na ten obiekt. Używamy `an` do dostępu do kodu i danych dla tej konkretnej instancji klasy `PartyAnimal`.

[2]Prawa autorskie do obrazu ciasteczka: CC-BY https://www.flickr.com/photos/dinnerseries/23570475099

Każdy obiekt/instancja `PartyAnimal` zawiera w sobie zmienną `x` oraz metodę/funkcję o nazwie `party`. W tej linii wywołujemy metodę `party`:

```
an.party()
```

Kiedy metoda `party` jest wywoływana, pierwszy parametr (który nazywamy umownie `self`) wskazuje na konkretną instancję obiektu `PartyAnimal`, z której wywoływana jest metoda `party`. W obrębie metody `party` widzimy linię:

```
self.x = self.x + 1
```

Składnia ta, używająca operatora *kropki*, mówi '`x` wewnątrz `self`'. Przy każdym wywołaniu `party()`, wewnętrzna wartość `x` jest zwiększana o 1 (a później wartość ta jest wypisywana na ekran przy pomocy `print()`).

Poniższa linia przedstawia kolejny sposób wywołania metody `party` wewnątrz obiektu `an`:

```
PartyAnimal.party(an)
```

W tym wariancie uzyskujemy dostęp do kodu bezpośrednio poprzez klasę i jawnie przekazujemy wskaźnik obiektu `an` jako pierwszy parametr metody (tj. `self`). Możesz myśleć o `an.party()` jako o skróconym zapisie powyższej linii.

Po uruchomieniu programu uzyskujemy następujący wynik:

```
Jak na razie 1
Jak na razie 2
Jak na razie 3
Jak na razie 4
```

Konstruujemy obiekt i czterokrotnie wywołujemy metodę `party`, zarówno zwiększając, jak i wypisując wartość `x` będącą wewnątrz obiektu `an`.

## 14.7. Klasy i typy

Jak już widzieliśmy, w Pythonie wszystkie zmienne mają określony typ. Możemy użyć wbudowanej funkcji `dir` do zbadania możliwości danej zmiennej. Możemy również użyć `type` i `dir` z tworzonymi klasami.

```
class PartyAnimal:
   x = 0

   def party(self) :
     self.x = self.x + 1
     print("Jak na razie", self.x)

an = PartyAnimal()
```

```
print ("Type", type(an))
print ("Dir ", dir(an))
print ("Type", type(an.x))
print ("Type", type(an.party))

# Kod źródłowy: https://pl.py4e.com/code3/party3.py
```

Po uruchomieniu programu zobaczymy następujący wynik:

```
Type <class '__main__.PartyAnimal'>
Dir  ['__class__', '__delattr__', ...
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'party', 'x']
Type <class 'int'>
Type <class 'method'>
```

Możesz zauważyć, że przy użyciu słowa kluczowego `class` stworzyliśmy nowy typ. W wyniku `dir` możesz zobaczyć, że w obiekcie jest dostępny zarówno atrybut całkowitoliczbowy `x`, jak i metoda `party`.

## 14.8. Cykl życia obiektu

W poprzednich przykładach definiowaliśmy klasę (szablon), używaliśmy jej do stworzenia instancji (obiektu) tej klasy, a następnie korzystaliśmy z instancji. Kiedy program zakończy pracę, wszystkie zmienne są porzucane. Zazwyczaj nie zastanawiamy się zbytnio nad tworzeniem i usuwaniem zmiennych. Jednak gdy nasze obiekty stają się coraz bardziej złożone, często musimy podjąć wewnątrz obiektu pewne działania, tak aby ustawić niektóre rzeczy w momencie gdy obiekt jest konstruowany i ewentualnie usunąć pewne rzeczy gdy obiekt jest porzucany.

Jeśli chcemy by nasz obiekt był świadomy momentów konstruowania i niszczenia, dodajemy do niego specjalnie nazwane metody:

```
class PartyAnimal:
   x = 0

   def __init__(self):
     print('Jestem tworzony')

   def party(self) :
     self.x = self.x + 1
     print('Jak na razie', self.x)

   def __del__(self):
     print('Jestem niszczony', self.x)

an = PartyAnimal()
an.party()
an.party()
```

```
an = 42
print('an zawiera', an)

# Kod źródłowy: https://pl.py4e.com/code3/party4.py
```

Po uruchomieniu programu zobaczymy następujący wynik:

```
Jestem tworzony
Jak na razie 1
Jak na razie 2
Jestem niszczony 2
an zawiera 42
```

Gdy Python tworzy nasz obiekt, to wywołuje naszą metodę `__init__`, tak aby dać nam szansę na ustawienie pewnych domyślnych lub początkowych wartości dla tego obiektu. Kiedy Python natrafi się na linię:

```
an = 42
```

to w rzeczywistości "odrzuca nasz obiekt", więc może ponownie użyć zmiennej `an` do przechowywania wartości `42`. Właśnie w tym momencie, w którym nasz obiekt `an` jest "niszczony", wywoływany jest nasz kod destruktora (`__del__`). Nie możemy powstrzymać naszej zmiennej przed zniszczeniem, ale możemy dokonać niezbędnego czyszczenia tuż przed tym, gdy nasz obiekt już nie będzie istniał.

Często podczas prac nad kodem obiektu zapada decyzja by dodać do niego konstruktor, tak aby ustawić jego początkowe wartości. Natomiast stosunkowo rzadko się zdarzy byśmy potrzebowali destruktora dla danego obiektu.

## 14.9. Wiele instancji

Jak do tej pory zdefiniowaliśmy klasę, stworzyliśmy pojedynczy obiekt, użyliśmy go, a następnie porzuciliśmy. Jednak prawdziwa moc w programowaniu obiektowym leży w tworzeniu wielu instancji naszej klasy.

Gdy tworzymy wiele obiektów na podstawie naszej klasy, możemy mieć potrzebę ustawienia dla każdego z nich różnych wartości początkowych. Dnae te możemy przekazać do konstruktorów, tak aby nadać każdemu z obiektów inną wartość początkową:

```
class PartyAnimal:
   x = 0
   name = ''
   def __init__(self, nam):
     self.name = nam
     print(self.name, '- utworzenie')

   def party(self) :
     self.x = self.x + 1
```

```
        print(self.name, '- zliczenie imprezek -', self.x)

s = PartyAnimal('Sally')
j = PartyAnimal('Jim')

s.party()
j.party()
s.party()

# Kod źródłowy: https://pl.py4e.com/code3/party5.py
```

Konstruktor posiada zarówno parametr `self`, który wskazuje na instancję obiektu, jak i dodatkowe parametry, które są przekazywane do konstruktora w trakcie tworzenia obiektu:

```
s = PartyAnimal('Sally')
```

Wewnątrz konstruktora, druga linia kopiuje parametr (`nam`), który jest przekazywany do atrybutu `name` będącego już w instancji obiektu.

```
self.name = nam
```

Wynik programu pokazuje, że każdy z obiektów (`s` i `j`) zawiera swoje własne, niezależne kopie `x` oraz `nam`:

```
Sally - utworzenie
Jim - utworzenie
Sally - zliczenie imprezek - 1
Jim - zliczenie imprezek - 1
Sally - zliczenie imprezek - 2
```

## 14.10. Dziedziczenie

Inną przydatną cechą programowania obiektowego jest możliwość stworzenia nowej klasy poprzez rozszerzenie już istniejącej. Podczas tej operacji klasę źródłową nazywamy *klasą bazową*, a nową klasę nazywamy *klasą pochodną* lub *klasą potomną*.

Kontynuując poprzedni przykład, przeniesiemy naszą klasę `PartyAnimal` do osobnego pliku `party.py`.

```
class PartyAnimal:
   x = 0
   name = ''
   def __init__(self, nam):
     self.name = nam
     print(self.name, '- utworzenie')

   def party(self) :
```

```
        self.x = self.x + 1
        print(self.name, '- zliczenie imprezek -', self.x)

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

Następnie, w nowym pliku możemy "zaimportować" klasę `PartyAnimal` i ją rozszerzyć ją, tak jak pokazano poniżej:

```
from party import PartyAnimal

class CricketFan(PartyAnimal):
   points = 0
   def six(self):
      self.points = self.points + 6
      self.party()
      print(self.name, "- punkty -", self.points)

s = PartyAnimal("Sally")
s.party()
j = CricketFan("Jim")
j.party()
j.six()
print(dir(j))

# Kod źródłowy: https://pl.py4e.com/code3/party6.py
```

Kiedy definiujemy klasę `CricketFan`, wskazujemy, że rozszerzamy klasę `PartyAnimal`. Oznacza to, że wszystkie zmienne (`x`) i metody (`party`) z klasy `PartyAnimal` są *dziedziczone* przez klasę `CricketFan`. Na przykład, w ramach metody `six` klasy `CricketFan` wywołujemy metodę `party` z klasy `PartyAnimal`.

Podczas uruchomienia programu tworzymy zmienne `s` i `j` jako niezależne instancje `PartyAnimal` i `CricketFan`. Obiekt `j` ma dodatkowe możliwości w porównaniu do obiektu `s`.

```
Sally - utworzenie
Sally - zliczenie imprezek - 1
Jim - utworzenie
Jim - zliczenie imprezek - 1
Jim - zliczenie imprezek - 2
Jim - punkty - 6
['__class__', '__delattr__', ... '__weakref__',
'name', 'party', 'points', 'six', 'x']
```

W wyniku funkcji `dir` na obiekcie `j` (instancja klasy `CricketFan`) widzimy, że obiekt ten ma atrybuty i metody klasy nadrzędnej, a ponadto atrybuty i metody, które zostały dodane poprzez rozszerzoną klasę `CricketFan`.

## 14.11. Podsumowanie

Powyższy rozdział jest bardzo szybkim wprowadzeniem do programowania obiektowego, które skupia się głównie na terminologii i składni używanej podczas definiowania i używania obiektów. Przejrzyjmy szybko kod, który widzieliśmy na początku rozdziału. W tym momencie powinieneś w pełni rozumieć, co się w nim dzieje.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Kod źródłowy: https://pl.py4e.com/code3/party1.py
```

Pierwsza linia tworzy *obiekt* typu `list`. Gdy Python tworzy obiekt tpypu `list`, to wywołuje metodę *konstruktora* (o nazwie `__init__`), tak by ustawić wewnętrzne atrybuty, które będą używane do przechowywania danych listy. Nie przekazaliśmy żadnych parametrów do *konstruktora*. Gdy konstruktor zakończy działanie, używamy zmiennej `stuff` tak, by móc wskazywać na zwróconą instancję klasy `list`.

Druga i trzecia linia wywołują metodę `append` z jednym parametrem, tak aby dodać nową pozycję na końcu listy, aktualizując atrybuty w ramach obiektu `stuff`. Następnie w czwartej linii wywołujemy metodę `sort` bez parametrów, po to by posortować dane wewnątrz obiektu `stuff`.

Następnie wypisujemy pierwszą pozycję z listy za pomocą nawiasów kwadratowych, które są skrótem do wywołania metody `__getitem__` w ramach `stuff`. Jest to równoważne z wywołaniem metody `__getitem__` w ramach *klasy* `list` oraz przekazaniem obiektu `stuff` jako pierwszego parametru i szukanej pozycji jako drugiego parametru.

Na końcu programu obiekt `stuff` jest porzucany, ale nie przed wywołaniem *destruktora* (o nazwie `__del__`), tak aby obiekt mógł pod koniec swego istnienia wyczyścić wszystkie niezbędne elementy, o ile jest to konieczne.

Omówiliśmy tutaj podstawy programowania obiektowego. Jest wiele innych dodatkowych tematów, np. jak najlepiej używać podejścia obiektowego podczas tworzenia dużych aplikacji i bibliotek, jednak są one poza zakresem tego rozdziału.[3].

## 14.12. Słowniczek

**atrybut** Zmienna, która jest częścią klasy.

[3] Jeśli jesteś ciekaw, gdzie jest zdefiniowana klasa `list`, to zajrzyj pod adres (miejmy nadzieję, że adres URL się nie zmieni) https://github.com/python/cpython/blob/master/Objects/listobject.c - klasa listy jest napisana w języku "C". Jeśli przejrzysz wspomniany kod źródłowy i okaże się on dla Ciebie interesujący, to być może powinieneś zrobić dodatkowo kilka kursów z dziedziny informatyki.

**destruktor** Opcjonalna, specjalnie nazwana metoda (`__del__`), która jest wywoływana w momencie, gdy obiekt jest niszczony. Destruktory są rzadko używane.

**dziedziczenie** Kiedy tworzymy nową klasę poprzez rozszerzenie istniejącej już klasy. Klasa pochodna posiada wszystkie atrybuty i metody klasy bazowej oraz dodatkowe atrybuty i metody zdefiniowane przez klasę pochodną.

**klasa** Szablon, który może być użyty do stworzenia obiektu. Definiuje atrybuty i metody, które będą składały się na obiekt.

**klasa bazowa** Klasa, która jest rozszerzana by utworzyć nową klasę pochodną. Klasa bazowa udostępnia klasie pochodnej wszystkie swoje metody i atrybuty.

**klasa pochodna** Nowa klasa utworzona w momencie rozszerzenia klasy bazowej. Klasa pochodna dziedziczy wszystkie atrybuty i metody klasy bazowej.

**konstruktor** Opcjonalna, specjalnie nazwana metoda (`__init__`), która jest wywoływana w momencie, gdy klasa jest używana do tworzenia obiektu. Zazwyczaj jest ona używana do ustawienia początkowych wartości obiektu.

**metoda** Funkcja, która jest zawarta w klasie i obiektach, które są z niej utworzone.

**obiekt** Utworzona instancja klasy. Obiekt zawiera wszystkie atrybuty i metody, które zostały zdefiniowane przez klasę. Niektóre dokumentacje obiektowe używają terminu “instancja” zamiennie z terminem “obiekt”.

# Rozdział 15

# Korzystanie z baz danych i języka SQL

## 15.1. Czym jest baza danych?

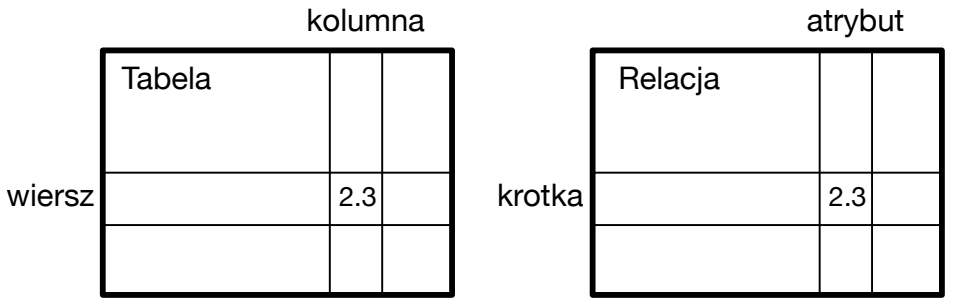
*Baza danych* to plik, który jest zorganizowany w specjalny sposób by mógł przechowywać dane. Większość baz danych jest zorganizowana tak jak słownik w tym sensie, że mapują one klucze na wartości. Największa różnica polega na tym, że baza danych znajduje się na dysku (lub innym trwałym nośniku danych), a więc zachowuje swoje dane po zakończeniu programu. Ponieważ baza danych przechowywana jest na trwałym nośniku, może ona przechowywać znacznie więcej danych niż słownik, co jest ograniczone wielkością pamięci w komputerze.

Podobnie jak słownik, oprogramowanie bazodanowe jest zaprojektowane tak, aby dodawania i dostęp do danych był bardzo szybki, nawet dla dużych ilości danych. Oprogramowanie bazodanowe utrzymuje swoją wydajność poprzez budowanie *indeksów*, dzięki czemu dane są dodawane do bazy w taki sposób by umożliwić komputerowi szybkie przejście do konkretnego wpisu.

Istnieje wiele różnych systemów bazodanowych, które są wykorzystywane do wielu różnych celów, w tym: Oracle, MySQL, Microsoft SQL Server, PostgreSQL i SQLite. W tej książce skupimy się na SQLite, ponieważ jest to bardzo popularna baza danych i jest już wbudowana w Pythona. SQLite jest przeznaczony do bycia *wbudowanym* (*osadzonym*) w innych aplikacjach, tak aby zapewnić obsługę bazy danych w obrębie tych aplikacji. Na przykład, przeglądarka Firefox również korzysta wewnętrznie z bazy danych SQLite, podobnie jak wiele innych aplikacji.

https://sqlite.org/

SQLite doskonale nadaje się do rozwiązywania niektórych informatycznych problemów z zarządzaniem danymi, takich jak robot internetowy do zbierania danych z Twittera, który opiszemy w tym rozdziale.




Rysunek 15.1: Relacyjne bazy danych

## 15.2. Pojęcia związane z bazami danych

Kiedy pierwszy raz spojrzysz na bazę danych, to będzie ona wyglądała jak plik arkusza kalkulacyjnego, który posiada wiele arkuszy. Podstawowe struktury danych w bazie danych to: *tabele*, *wiersze*, i *kolumny*.

W technicznych opisach relacyjnych baz danych pojęcia tabeli, wiersza i kolumny są bardziej formalnie nazywane odpowiednio *relacją*, *krotką* i *atrybutem*. W tym rozdziale użyjemy mniej formalnych pojęć.

## 15.3. Przeglądarka baz SQLite

Co prawda w tym rozdziale skoncentrujemy się na używaniu Pythona do pracy z danymi zawartymi w plikach bazy SQLite, jednak wiele operacji można wykonać wygodniej przy użyciu darmowego oprogramowania *Database Browser for SQLite*:

https://sqlitebrowser.org/

Za pomocą tej aplikacji można łatwo tworzyć tabele, wstawiać i edytować dane oraz uruchamiać proste zapytania SQL dotyczące danych zawartych w bazie.

W pewnym sensie przeglądarka bazy danych jest podobna do edytora tekstowego używanego podczas pracy z plikami tekstowymi. Kiedy chcesz wykonać jedną lub kilka operacji na pliku tekstowym, możesz po prostu otworzyć go w edytorze tekstowym i dokonać żądanych zmian. Kiedy musisz wykonać wiele zmian w pliku tekstowym, to często szybciej i wygodniej będzie Ci napisać prosty program Pythona. Ten sam schemat postępowania możesz zastosować podczas pracy z bazami danych. Proste operacje będziesz wykonywał w menedżerze baz danych, a bardziej złożone operacje wygodniej będzi Ci wykonywać w Pythonie.

## 15.4. Tworzenie tabeli w bazie danych

Bazy danych wymagają bardziej szczegółowo zdefiniowanej struktury niż listy czy słowniki w Pythonie[1].

[1]SQLite faktycznie pozwala na pewną elastyczność w typie danych przechowywanych w kolumnie, ale w tym rozdziale nasze typy danych będą ściśle określone, dzięki czemu pokazane

Kiedy w bazie danych tworzymy *tabelę*, to musimy z wyprzedzeniem podać nazwę każdej *kolumny* występującej w tabeli oraz typ danych, który zamierzamy przechowywać w każdej z *kolumn*. Kiedy oprogramowanie bazy danych wie jaki typ danych będzie użyty w każdej kolumnie, może dzięki temu wybrać najbardziej efektywny sposób przechowywania i wyszukiwania danych w oparciu o ich typ.

Możesz zapoznać się z różnymi typami danych obsługiwanymi przez SQLite pod następującym adresem URL:

https://www.sqlite.org/datatypes.html

Zdefiniowanie z góry struktury dla Twoich danych może się początkowo wydawać niewygodne, ale zyskiem z tego jest szybkie uzyskiwanie dostępu do Twoich danych, nawet jeśli baza danych zawiera ich dużą ilość.

Kod tworzący plik bazy danych i tabelę o nazwie `Utwory` z dwiema kolumnami jest następujący:

```
import sqlite3

conn = sqlite3.connect('muzyka.sqlite')
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Utwory')
cur.execute('CREATE TABLE Utwory (tytuł TEXT, odtworzenia INTEGER)')

conn.close()

# Kod źródłowy: https://pl.py4e.com/code3/db1.py
```

Operacja `connect` nawiązuje "połączenie" z bazą danych przechowywaną w pliku `muzyka.sqlite`, który znajduje się w bieżącym katalogu. Jeśli plik ten nie istnieje, to zostanie utworzony. Powodem, dla którego nazywa się to "połączeniem" jest to, że czasami baza danych przechowywana jest na "serwerze baz danych", który jest innym komputerem niż ten, na którym uruchamiamy naszą aplikację. W naszych prostych przykładach baza danych będzie po prostu plikiem lokalnym w tym samym katalogu co uruchamiany przez nas kod Pythona.

*Kursor* jest podobny uchwytu pliku i używamy go do wykonania operacji na danych przechowywanych w bazie. Wywołanie `cursor()` jest koncepcyjnie bardzo podobne do wywołania `open()` podczas pracy z plikami tekstowymi.

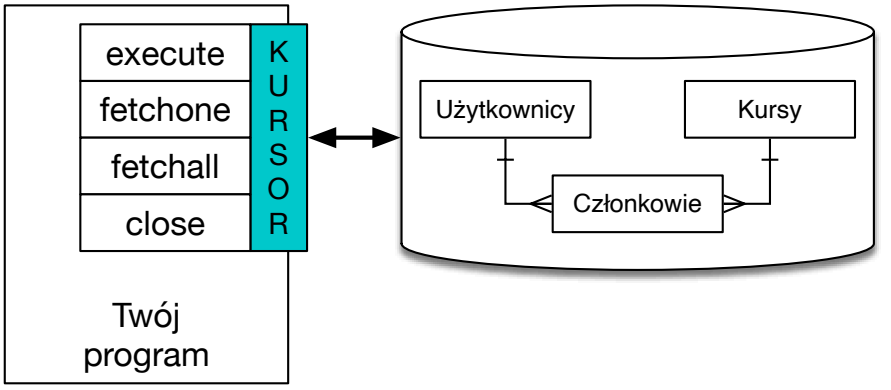
Posiadając kursor, przy użyciu metody `execute()` możemy rozpocząć wykonywanie poleceń działających na zawartości bazy danych.

Komendy bazy danych są wyrażone w specjalnym języku, który został ustandaryzowany u wielu różnych dostawców baz danych, tak aby umożliwić nam naukę jednego języka bazy danych. Język ten nazywa się *Structured Query Language* (z ang. strukturalny język zapytań) lub w skrócie *SQL*.

https://pl.wikipedia.org/wiki/SQL

W powyższym przykładzie na naszej bazie danych wykonujemy dwa polecenia SQL. Przyjmuje się, że słowa kluczowe SQL pisze się dużymi literami, a części polecenia,

tutaj idee będą miały zastosowanie również w innych systemach baz danych, np. w MySQL.




Rysunek 15.2: Kursor bazodanowy

które my dodajemy (takie jak nazwy tabel i kolumn) pisze się małymi literami lub z dużej litery.

Pierwsze polecenie SQL usuwa z bazy danych tabelę `Utwory`, jeśli ta tabela istnieje. Taki wzorzec postępowania ma po prostu pozwolić nam na wielokrotne uruchamianie tego samego programu do tworzenia tabeli `Utwory`, bez generowania błędu. Zauważ, że polecenie `DROP TABLE` usuwa z bazy danych tabelę i całą jej zawartość (tzn. nie ma tutaj żadnej operacji typu "cofnij").

```
cur.execute('DROP TABLE IF EXISTS Utwory')
```

Druga komenda tworzy tabelę o nazwie `Utwory` z kolumną tekstową o nazwie `tytuł` i kolumną całkowitoliczbową o nazwie `odtworzenia`.

```
cur.execute('CREATE TABLE Utwory (tytuł TEXT, odtworzenia INTEGER)')
```

Teraz, gdy stworzyliśmy tabelę o nazwie `Utwory`, możemy umieścić w niej pewne dane używając operacji SQL `INSERT`. Ponownie, zaczynamy od nawiązania połączenia z bazą danych i uzyskania kursora. Następnie możemy wykonać polecenia SQL za pomocą tego kursora.

Polecenie SQL `INSERT` wskazuje na to jakiej używamy tabeli, a następnie definiuje nowy wiersz, wymieniając pola, które chcemy umieścić w nowym wierszu `(tytuł, odtworzenia)`, a następnie w `VALUES` wartości, które chcemy umieścić w nowym wierszu. Wartości określone jako znaki zapytania `(?, ?)` wskazują, że rzeczywiste wartości do wstawienia są przekazywane jako krotka `( 'My Way', 15 )` będąca drugim parametrem wywołania `execute()`.

```
import sqlite3

conn = sqlite3.connect('muzyka.sqlite')
cur = conn.cursor()

cur.execute('INSERT INTO Utwory (tytuł, odtworzenia) VALUES (?, ?)',
    ('Thunderstruck', 20))
```

Utwory

| tytuł | odtworzenia |
|---|---|
| Thunderstruck | 20 |
| My Way | 15 |

Rysunek 15.3: Wiersze w tabeli

```
cur.execute('INSERT INTO Utwory (tytuł, odtworzenia) VALUES (?, ?)',
    ('My Way', 15))
conn.commit()

print('Utwory:')
cur.execute('SELECT tytuł, odtworzenia FROM Utwory')
for row in cur:
     print(row)

cur.execute('DELETE FROM Utwory WHERE odtworzenia < 100')
conn.commit()

cur.close()

# Kod źródłowy: https://pl.py4e.com/code3/db2.py
```

Najpierw wstawiamy poprzez `INSERT` dwa wiersze do naszej tabeli i używamy `commit()` aby wymusić zapis danych do pliku bazy danych.

Następnie używamy polecenia `SELECT` do pobrania wierszy, które właśnie wstawiliśmy w tabeli. W poleceniu `SELECT` wskazujemy, z których kolumn chcielibyśmy pobrać dane `(tytuł, odtworzenia)`, oraz wskazujemy, z której tabeli chcemy pobrać dane. Po wykonaniu polecenia `SELECT`, kursor jest czymś, po czym możemy przejść w pętli `for`. W celu zachowania wydajności, kursor po wykonaniu instrukcji `SELECT` nie odczytuje odrazu wszystkich danych z bazy danych. Zamiast tego, dane są odczytywane na żądanie, tj. w tym wypadku gdy idziemy kolejno przez wiersze w instrukcji `for`.

Wynik działania programu jest następujący:

```
Utwory:
('Thunderstruck', 20)
('My Way', 15)
```

Nasza pętla `for` znajduje dwa wiersze, a każdy z nich jest krotką Pythona, w której pierwsza wartość to `tytuł`, a drugą to `odtworzenia`.

*Uwaga: W innych książkach lub w internecie możesz zobaczyć ciągi znaków rozpoczynające się od* `u'`*. W Pythonie 2 była to wskazówka, że ciągi znaków są napisami z zestawem znaków Unicode, które są w stanie przechowywać niełacińskie znaki.*

*W Pythonie 3, wszystkie ciągi znaków są domyślnie napisami z zestawem znaków Unicode.*

Na samym końcu programu wykonujemy polecenie SQL `DELETE` aby usunąć wiersze, które właśnie utworzyliśmy, dzięki czemu możemy nasz program uruchamiać wielokrotnie bez wystąpienia błędu. Polecenie `DELETE` pokazuje użycie klauzuli `WHERE`, która pozwala nam wyrazić kryterium wyboru, dzięki czemu możemy poprosić bazę danych o zastosowanie polecenia tylko do tych wierszy, które odpowiadają temu kryterium. W tym przykładzie kryterium to stosuje się do wszystkich wierszy, więc czyścimy tabelę, tak by móc wielokrotnie uruchamiać nasz program. Po wykonaniu operacji `DELETE` wywołujemy również `commit()` aby wymusić usunięcie danych z bazy.

## 15.5. Podsumowanie języka SQL

W powyższych przykładach użyliśmy języka SQL i omówiliśmy jego kilka podstawowych poleceń. W tej sekcji przyjrzymy się bliżej językowi SQL i przedstawimy przegląd składni tego języka.

Ponieważ istnieje wielu różnych dostawców baz danych, SQL został ustandaryzowany po to byśmy mogli komunikować się w podobny sposób z bazami danych działających na różnych silnikach.

Relacyjna baza danych składa się z tabel, wierszy i kolumn. Typ danych w kolumnie zazwyczaj jest tekstem, liczbą lub datą. Kiedy tworzymy tabelę, wskazujemy nazwy i typy kolumn:

```
CREATE TABLE Utwory (tytuł TEXT, odtworzenia INTEGER)
```

Aby wstawić wiersz do tabeli, używamy polecenia SQL `INSERT`:

```
INSERT INTO Utwory (tytuł, odtworzenia) VALUES ('My Way', 15)
```

Polecenie `INSERT` określa nazwę tabeli, następnie listę pól/kolumn, które chcesz ustawić w nowym wierszu, a następnie słowo kluczowe `VALUES` i listę odpowiednich wartości dla każdego pola.

Polecenie SQL `SELECT` jest używane do pobierania wierszy i kolumn z bazy danych. Polecenie `SELECT` pozwala określić, które kolumny chcesz pobrać, a także klauzulę `WHERE` by wybrać tylko te wiersze, które chcesz zobaczyć. Polecenie to pozwala także na opcjonalną klauzulę `ORDER BY` kontrolującą sortowanie zwracanych wierszy.

```
SELECT * FROM Utwory WHERE tytuł = 'My Way'
```

Użycie `*` między `SELECT` a `FROM` wskazuje, że chcesz aby baza danych zwróciła wszystkie kolumny dla każdego wiersza, który pasuje do klauzuli `WHERE`.

Zauważ, że w przeciwieństwie do Pythona, w klauzuli SQL `WHERE` używamy pojedynczego znaku równości do wskazania testu na równość, a nie znaku podwójnej równości. Inne operacje logiczne dozwolone w klauzuli `WHERE` obejmują `<`, `>`, `<=`, `>=`, `!=`, jak również `AND` i `OR` oraz nawiasy okrągłe do tworzenia wyrażeń logicznych.

Możesz zażądać, aby zwrócone wiersze były posortowane według którejś z kolumn:

```
SELECT tytuł, odtworzenia FROM Utwory ORDER BY tytuł
```

Aby usunąć wiersz, potrzebna jest klauzula `WHERE` na instrukcji SQL `DELETE`. Klauzula `WHERE` określa, które wiersze mają zostać usunięte:

```
DELETE FROM Utwory WHERE tytuł = 'My Way'
```

Przy użyciu polecenia SQL `UPDATE` możliwa jest aktualiza kolumn w obrębie jednego lub więcej wierszy w tabeli:

```
UPDATE Utwory SET odtworzenia = 16 WHERE tytuł = 'My Way'
```

Polecenie `UPDATE` określa tabelę, a następnie po słowie kluczowym `SET` wskazuje listę pól i wartości, które mają zostać zmienione, a następnie mamy opcjonalną klauzulę `WHERE` do wyboru wierszy, które mają zostać zaktualizowane. Pojedyncze wyrażenie `UPDATE` zmienia wszystkie wiersze odpowiadające klauzuli `WHERE`. Jeśli klauzula `WHERE` nie jest określona, to wykonuje ona aktualizację dla wszystkich wierszy znajdujących się w tabeli.

Opisane wyżej cztery podstawowe polecenia SQL (`INSERT`, `SELECT`, `UPDATE` i `DELETE`) pozwalają na wykonanie podstawowych operacji potrzebnych do utworzenia i utrzymania danych.

## 15.6. Zbieranie informacji z Twittera przy użyciu baz danych

W poniżej sekcji stworzymy prostego robota internetowego, który przejdzie przez kilka kont na Twitterze i na podstawie zebranych informacji zbudujemy bazę danych. *Uwaga: Bądź bardzo ostrożny podczas uruchamiania poniższych programów. Nie chcesz pobierać zbyt dużo danych lub uruchamiać programów zbyt długo, ponieważ Twittera wyłączy Ci dostęp do swojego API.*

Jednym z problemów związanych z każdym rodzajem robota internetowego jest to, że czasami trzeba go wielokrotnie zatrzymywać i uruchamiać ponownie i jednocześnie nie chcemy utracić danych, które do tej pory pobraliśmy. Nie chcemy by ponowne uruchomienie procesu zbierania danych zaczynało się w tym samym punkcie startowym, więc podczas pobierania danych musimy je przechowywać tak, aby nasz program mógł rozpocząć działanie w tym miejscu, w którym ostatnio zakończył pracę.

Zaczniemy od pobrania z Twittera listy znajomych jakiegoś użytkownika i ich statusów, przejrzenia tej listy i dodania każdego ze znajomych do bazy danych, która zostanie użyta w przyszłości do dalszego pobierania danych. Po przetworzeniu znajomych jednej osoby, sprawdzamy naszą bazę danych i bierzemy jednego ze znajomych tego użytkownika. Robimy tak w kółko, wybierając "nieprzetworzną" osobę, pobierając jej listę znajomych i dodając tych znajomych, których jeszcze nie widzieliśmy, do naszej listy do odwiedzenia w przyszłości.

Śledzimy również, ile razy widzieliśmy danego znajomego w bazie danych, aby uzyskać jakiś wskaźnik jego "popularności".

Przechowując na dysku komputera bazę dotyczącą znanych nam kont Twittera, informacji czy już to kontro odwiedziliśmy oraz jak dane konto jest popularne, możemy zatrzymać i uruchomić ponownie nasz program tyle razy, ile chcemy.

Poniższy program jest nieco skomplikowany. Opiera się na kodzie z ćwiczenia z wcześniejszej części książki, która korzysta z API Twittera.

Oto kod źródłowy naszej aplikacji zbierającej informacje z Twittera:

```
from urllib.request import urlopen
import urllib.error
import twurl
import json
import sqlite3
import ssl

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()

cur.execute('''
            CREATE TABLE IF NOT EXISTS Twitter
            (nazwa TEXT, pobrany INTEGER, znajomi INTEGER)''')

# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    acct = input("Podaj nazwę konta na Twitterze lub wprowadź
↪   'koniec': ")
    if (acct == 'koniec'): break
    if (len(acct) < 1):
        cur.execute('SELECT nazwa FROM Twitter WHERE pobrany = 0
↪   LIMIT 1')
        try:
            acct = cur.fetchone()[0]
        except:
            print('Nie znaleziono niepobranych kont Twittera')
            continue

    url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count':
↪   '20'})
    print('Pobieranie', url)
    connection = urlopen(url, context=ctx)
    data = connection.read().decode()
    headers = dict(connection.getheaders())

    print('Pozostało', headers['x-rate-limit-remaining'])
    js = json.loads(data)
```

```
    # Debugowanie
    # print json.dumps(js, indent=4)

    cur.execute('UPDATE Twitter SET pobrany=1 WHERE nazwa = ?',
↪   (acct, ))

    countnew = 0
    countold = 0
    for u in js['users']:
        friend = u['screen_name']
        print(friend)
        cur.execute('SELECT znajomi FROM Twitter WHERE nazwa = ?
↪   LIMIT 1',
                    (friend, ))
        try:
            count = cur.fetchone()[0]
            cur.execute('UPDATE Twitter SET znajomi = ? WHERE nazwa
↪   = ?',
                        (count+1, friend))
            countold = countold + 1
        except:
            cur.execute('''INSERT INTO Twitter (nazwa, pobrany,
↪   znajomi)
                        VALUES (?, 0, 1)''', (friend, ))
            countnew = countnew + 1
    print('Nowe konta=', countnew, ' widziane ponownie=', countold)
    conn.commit()

cur.close()

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

Nasza baza danych jest przechowywana w pliku `spider.sqlite` i ma jedną tabelę o nazwie `Twitter`. Każdy wiersz w tabeli `Twitter` ma osobną kolumnę dla nazwy konta, informacji czy pobraliśmy listę znajomych tego konta oraz tego ile razy konto wystąpiło na listach znajomych.

W głównej pętli programu prosimy o podanie nazwy konta Twittera lub "koniec", aby wyjść z programu. Jeżeli użytkownik poda nazwę konta na Twitterze, pobierzemy listę jego znajomych (i jego statusy) oraz dodamy każdego znajomego do bazy danych, o ile jeszcze go tam nie ma. Jeśli znajomy jest już na liście, to w bazie danych dodajemy 1 do pola `znajomi` w danym wierszu tabeli.

Jeśli użytkownik naciśnie przycisk `<Enter>`, to szukamy w bazie danych kolejnego konta na Twitterze, którego jeszcze nie odwiedziliśmy, pobieramy jego znajomych i statusy tego konta, dodajemy znajomych do bazy danych lub aktualizujemy je, zwiększając ich liczbę w `friends`.

Kiedy pozyskamy już listę znajomych i statusy, idziemy w pętli przez wszystkie elementy `user` będące w zwróconym JSONie i dla każdego użytkownika pobieramy `screen_name`. Następnie używamy instrukcji `SELECT` by sprawdzić, czy ten konkretna wartość `screen_name` została już zapisana w bazie danych i pobieramy

liczbę znajomych (`friends`), jeśli dany rekord istnieje.

```
countnew = 0
countold = 0
for u in js['users']:
    friend = u['screen_name']
    print(friend)
    cur.execute('SELECT znajomi FROM Twitter WHERE nazwa = ? LIMIT
↪   1',
                (friend, ))
    try:
        count = cur.fetchone()[0]
        cur.execute('UPDATE Twitter SET znajomi = ? WHERE nazwa =
↪   ?',
                    (count+1, friend))
        countold = countold + 1
    except:
        cur.execute('''INSERT INTO Twitter (nazwa, pobrany, znajomi)
                    VALUES (?, 0, 1)''', (friend, ))
        countnew = countnew + 1
print('Nowe konta=', countnew, ' widziane ponownie=', countold)
conn.commit()
```

Gdy kursor wykona instrukcję `SELECT`, musimy pobrać wiersze. Możemy to zrobić za pomocą pętli `for`, ale ponieważ pobieramy tylko jeden wiersz (`LIMIT 1`), możemy użyć metody `fetchone()` do pobrania pierwszego (i jedynego) wiersza, który jest wynikiem operacji `SELECT`. Ponieważ metoda `fetchone()` zwraca wiersz jako *krotkę* (nawet jeśli jest tylko jedno pole), bierzemy pierwszą wartość z krotki by uzyskać bieżące zliczenie znajomych i wstawić tę wartość do zmiennej `count`.

Jeśli to pobranie się powiedzie, używamy polecenia SQL `UPDATE` z klauzulą `WHERE`, tak aby dodać 1 do kolumny `friends` w wierszu, który pasuje do konta analizowanego znajomego. Zauważ, że w SQL są dwa symbole zastępcze (tzn. znaki zapytania), a drugi parametr `execute()` jest dwuelementową krotką, która przechowuje wartości, które mają być zastąpione w zapytaniu SQL zamiast znaków zapytania.

Jeśli kod w bloku `try` się nie powiedzie, to prawdopodobnie dlatego, że w instrukcji `SELECT` żaden rekord nie pasuje do klauzuli `WHERE name = ?`. Tak więc w bloku `except` używamy polecenia `INSERT` aby dodać do tabeli atrybut znajomego opisujący jego nazwę ekranową, tj. `screen_name`em ze wskazaniem, że nie pobraliśmy jeszcze `screen_name` i ustawiliśmy liczbę znajomych na jeden.

Podsumowując, przy pierwszym uruchomieniu programu i podaniu konta Twittera, program działa w następujący sposób:

```
Podaj nazwę konta na Twitterze lub wprowadź 'koniec': drchuck
Pobieranie https://api.twitter.com/1.1/friends ...
(...)
Nowe konta= 20  widziane ponownie= 0
Podaj nazwę konta na Twitterze lub wprowadź 'koniec': koniec
```

Ponieważ jest to pierwsze uruchomienie programu, baza danych jest pusta (a w zasadzie jej nie ma), więc tworzymy bazę danych w pliku `spider.sqlite` i dodajemy do niej tabelę o nazwie `Twitter`. Następnie pobieramy kilku znajomych i dodajemy ich wszystkich do pustej bazy danych.

W tym momencie moglibyśmy napisać prosty program do wykonywania zrzutu bazy danych, tak aby przyjrzeć się temu, co znajduje się w naszym pliku `spider.sqlite`:

```
import sqlite3

conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()
cur.execute('SELECT * FROM Twitter')
count = 0
for row in cur:
    print(row)
    count = count + 1
print(count, 'wierszy.')
cur.close()

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

Program ten po prostu otwiera bazę danych i pobiera wszystkie kolumny ze wszystkich wierszy znajdujących się w tabeli `Twitter`, a następnie w pętli przechodzi przez przez wszystkie wiersze i wypisuje każdy z nich.

Jeśli uruchomimy powyższy program po pierwszym uruchomieniu naszego robota internetowego, jego wyjście będzie wyglądało następująco:

```
('opencontent', 0, 1)
('lhawthorn', 0, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
20 wierszy.
```

W każdym wierszu widzimy nazwę ekranową `screen_name` użytkownika, informację, że jeszcze nie pobraliśmy danych dla tego użytkownika, oraz że każdy w bazie danych ma jednego znajomego.

W tym momencie nasza baza danych odzwierciedla pobieranie danych o znajomych z naszego pierwszego konta na Twitterze (*drchuck*). Możemy uruchomić program ponownie i wskazać mu, by pobrał znajomych z kolejnego "nieprzetworzonego" jeszcze konta, naciskając po prostu `<Enter>` (zamiast podawać nazwę konta na Twitterze). Wynik może być mniej więcej taki (uwaga: możliwe jest, że będziesz musiał kilkukrotnie wcisnąć `<Enter>` by natrafić na sytuację gdy pojawią się konta, które już były widziane wcześniej):

```
Podaj nazwę konta na Twitterze lub wprowadź 'koniec':
Pobieranie https://api.twitter.com/1.1/friends ...
(...)
```

```
Nowe konta= 18  widziane ponownie= 2
Podaj nazwę konta na Twitterze lub wprowadź 'koniec':
Pobieranie https://api.twitter.com/1.1/friends ...
(...)
Nowe konta= 17  widziane ponownie= 3
Podaj nazwę konta na Twitterze lub wprowadź 'koniec': koniec
```

Ponieważ wcisnęliśmy `<Enter>` (tzn. nie określiliśmy nazwy konta na Twitterze), wykonywany jest następujący kod:

```
if (len(acct) < 1):
    cur.execute('SELECT nazwa FROM Twitter WHERE pobrany = 0 LIMIT
↪  1')
    try:
        acct = cur.fetchone()[0]
    except:
        print('Nie znaleziono niepobranych kont Twittera')
        continue
```

Używamy polecenia SQL `SELECT`, aby pobrać nazwę pierwszego (`LIMIT 1`) użytkownika, który nadal ma ustawioną wartość "czy przetworzyliśmy już tego użytkownika?" na zero. Używamy także formuły `fetchone()[0]` w obrębie bloku `try/except` albo by wydobyć z pobranych danych `screen_name`, albo by wyświetlić informację o błędzie i wykonać ponownie pętlę.

Jeśli udało nam się uzyskać z `screen_name` nieprzetworzone jeszcze konto, to pobieramy dane z tego konta w następujący sposób:

```
url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count':
↪  '20'})
print('Pobieranie', url)
connection = urlopen(url, context=ctx)
data = connection.read().decode()
# ...
js = json.loads(data)
# ...
cur.execute('UPDATE Twitter SET pobrany=1 WHERE nazwa = ?', (acct,
↪  ))
```

Aby nie przetwarzać wszystkich znajomych danego użytkownika i zachować pewną czytelność w wynikach, zastosujemy pewne uproszczenie, tj. ograniczymy się tylko do pierwszych 20 osób (`'count': '20'`) i zignorujemy pozostałych znajomych.

Po pomyślnym pobraniu danych, używamy instrukcji `UPDATE`, aby ustawić kolumnę `pobrany` na 1, po to by wskazać, że zakończyliśmy pobieranie listy znajomych tego konta. Dzięki temu chroni nas to przed wielokrotnym pobieraniem tych samych danych i sprawia, że robimy postępy w budowie sieci znajomych kont Twittera.

Jeśli uruchomimy pierwszy program do pobierania listy znajomych, naciśniemy kilka razy `<Enter>`, aby uzyskać kolejnych nieprzetworzonych jeszcze znajomych jakiegoś znajomego, a następnie uruchomimy program do zrzucania danych z bazy, to uzyskamy mniej więcej podobny po poniższego wynik (tutaj wciśnięto `<Enter>` dwa razy, ale rzeczywisty aktualny wynik może się różnić):

```
('opencontent', 1, 1)
('lhawthorn', 1, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
('cnxorg', 0, 2)
('knoop', 0, 1)
('kthanos', 0, 2)
('LectureTools', 0, 1)
...
55 wierszy.
```

Widzimy, że poprawnie zapisaliśmy informację, że odwiedziliśmy konta `lhawthorn` i `opencontent`. Również konta `cnxorg` i `kthanos` mają już dwie osoby śledzące. Odkąd pozyskaliśmy znajomych trzech osób (`drchuck`, `opencontent` i `lhawthorn`), nasza tabela ma 55 wierszy dotyczących pobranych użytkowników.

Za każdym razem, gdy uruchomimy program i wciśniemy przycisk `<Enter>`, wybierze on następne nieprzetworzone konto (np. następnym kontem będzie `steve_coppin`), pobierze jego znajomych, oznaczy go jako pobranego, a dla każdego znajomego użytkownika `steve_coppin` albo doda go na końcu tabeli, albo zaktualizuje jego liczbę `znajomi` jeśli już jest w bazie.

Ponieważ wszystkie dane programu są przechowywane na dysku w bazie danych, aktywność robota internetowego może być zawieszana i wznawiana tyle razy, ile chcesz, bez utraty danych.

## 15.7. Podstawy modelowania danych

Prawdziwa siła relacyjnej bazy danych ujawnia się w momencie utworzenia wielu tabel i połączeń pomiędzy nimi. Czynność decydująca o tym jak rozbić dane aplikacji na wiele tabel i ustalić związki między nimi nazywamy *modelowaniem danych*. Dokument projektowy, który pokazuje tabele i ich związki, nazywa się *modelem danych*.

Modelowanie danych jest stosunkowo wyrafinowaną umiejętnością i w tej części wprowadzimy tylko najbardziej podstawowe pojęcia z zakresu modelowania danych relacyjnych. Aby uzyskać więcej szczegółów na temat modelowania danych, możesz zacząć od poniższej strony:

https://pl.wikipedia.org/wiki/Model_relacyjny

Powiedzmy, że w aplikacji naszego robota internetowego chodzącego po Twitterze, zamiast po prostu zliczać znajomych danej osoby, chcielibyśmy prowadzić listę wszystkich związków "przychodzących", tak abyśmy mogli znaleźć listę wszystkich osób, które śledzą dane konto.

Ponieważ każdy użytkownik potencjalnie będzie miał wiele kont, które go śledzą, nie możemy po prostu dodać jednej kolumny do naszej tabeli `Twitter`. Tworzymy więc nową tabelę, która śledzi pary znajomych. Poniżej znajduje się prosty sposób na utworzenie takiej tabeli:

```
CREATE TABLE Znajomości (znajomy_od TEXT, znajomy_do TEXT)
```

Za każdym razem, gdy napotykamy osobę, którą śledzi użytkownik `drchuck`, wstawiamy do tabeli wiersz w postaci:

```
INSERT INTO Znajomości (znajomy_od, znajomy_do) VALUES ('drchuck',
↪  'lhawthorn')
```

Ponieważ przetwarzamy na Twitterze 20 znajomych z kanału użytkownika `drchuck`, wstawimy 20 rekordów z "drchuckiem" jako pierwszym parametrem, przez co będziemy wielokrotnie duplikować ciąg w bazie danych.

Tego typu powielanie danych ciągów znaków narusza jedną z najlepszych praktyk *normalizacji baz danych*, która w zasadzie stwierdza, że nigdy nie powinniśmy umieszczać w bazie danych więcej niż jeden raz tych samych danych ciągów znaków. Jeżeli potrzebujemy tych danych więcej niż jeden raz, tworzymy dla tych danych numeryczny *klucz* i odwołujemy się do rzeczywistych danych za pomocą tego klucza.

W praktyce ciąg znaków zajmuje na dysku i w pamięci naszego komputera dużo więcej miejsca niż liczba całkowita, a porównywanie i sortowanie zajmuje więcej czasu procesora w przypadku ciągów znaków. Jeśli mamy tylko kilkaset wpisów, to czas związany z dostępem i przetwarzaniem danych nie ma większego znaczenia. Ale jeśli mamy milion osób w naszej bazie danych i możliwość 100 milionów powiązań pomiędzy znajomymi, to ważne jest, aby móc jak najszybciej przetworzyć tego typu dane.

Będziemy przechowywać nasze konta na Twitterze w tabeli o nazwie `Osoby` zamiast używanej w poprzednim przykładzie tabeli `Twitter`. Tabela `Osoby` posiada dodatkową kolumnę do przechowywania klucza numerycznego powiązanego z wierszem dla tego użytkownika Twittera. SQLite posiada możliwość automatycznego dodawania wartości klucza do każdego wiersza, który wstawiamy do tabeli, osiąganą za pomocą specjalnego typu danych kolumny (`INTEGER PRIMARY KEY`).

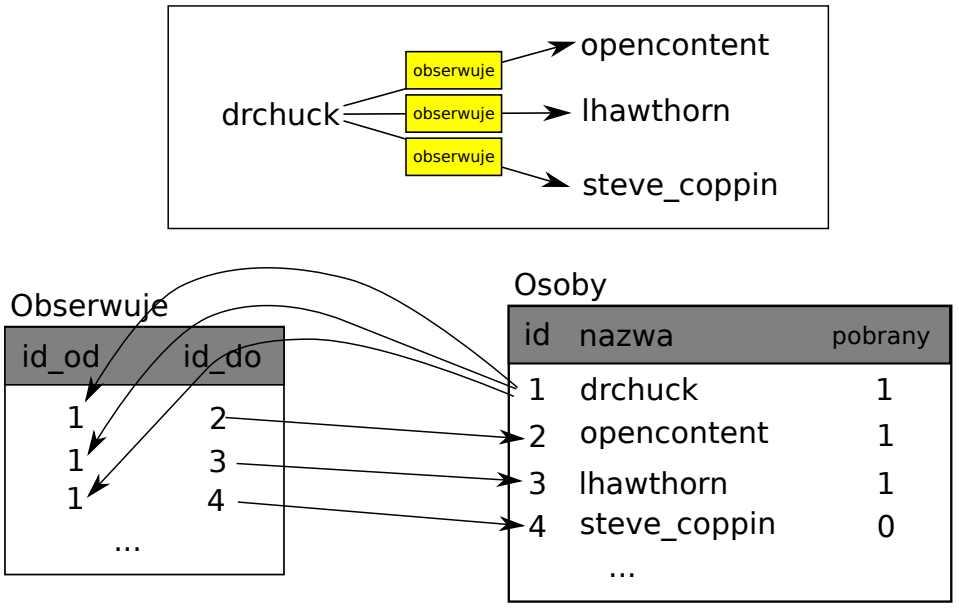
Możemy utworzyć tabelę `Osoby` z dodatkową kolumną `id` w następujący sposób:

```
CREATE TABLE Osoby
    (id INTEGER PRIMARY KEY, nazwa TEXT UNIQUE, pobrana INTEGER)
```

Zauważ, że w każdym wierszu tabeli `Osoby` nie zapisujemy już liczby znajomych. Kiedy jako typ naszej kolumny `id` wybierzemy `INTEGER PRIMARY KEY`, to wskazujemy, że chcielibyśmy by SQLite zarządzał tą kolumną i automatycznie przypisywał unikalny klucz numeryczny każdemu wstawianemu wierszowi. Dodajemy również słowo kluczowe `UNIQUE`, aby wskazać, że nie pozwolimy SQLite na wstawienie dwóch wierszy o tej samej wartości dla kolumny `nazwa`.

Teraz zamiast tworzyć wspomnianą wyżej tabelę `Znajomości`, utworzymy tabelę o nazwie `Obserwuje` z dwiema kolumnami całkowitoliczbowymi `id_od` i `id_do` oraz ograniczeniem, że w tej tabeli *kombinacja/złożenie* `id_od` i `id_do` musi być unikalna (tzn. nie możemy wstawiać duplikatów wierszy).

```
CREATE TABLE Obserwuje
    (id_od INTEGER, id_do INTEGER, UNIQUE(id_od, id_do) )
```




Rysunek 15.4: Związki między tabelami

Kiedy dodajemy do naszych tabel klauzule `UNIQUE`, w rzeczywistości przekazujemy zestaw reguł, o których egzekwowanie prosimy bazę danych przy próbie wstawiania nowych rekordów. Reguły te tworzymy jako pewne udogodnienie w naszych programach, co zobaczymy za chwilę. Reguły te powstrzymują nas przed popełnianiem błędów i ułatwiają napisanie niektórych części naszego kodu.

W istocie, tworząc tabelę `Obserwuje` modelujemy "związek", w którym jedna osoba "obserwuje" drugą, i reprezentujemy ten związek parą liczb (a) wskazując, że jakaś para ludzi jest w jakiś sposób powiązana i (b) wskazując na kierunek tego związku.

## 15.8. Programowanie z użyciem wielu tabel

index{klucz główny}

Napiszemy jeszcze raz kod robota internetowego do chodzenia po kontach użytkowników Twittera, ale tym razem używając dwóch tabel i kluczy głównych i odniesień między tabelami. Dodatkowo w tej wersji programu zwiększymy liczbę pobieranych znajomych z 20 do 100. Poniżej znajduje się nowa wersja programu:

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
import sqlite3
import ssl

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'
```

```
conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()

cur.execute('''CREATE TABLE IF NOT EXISTS Osoby
            (id INTEGER PRIMARY KEY, nazwa TEXT UNIQUE, pobrana
↪  INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Obserwuje
            (id_od INTEGER, id_do INTEGER, UNIQUE(id_od, id_do))''')

# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    acct = input("Podaj nazwę konta na Twitterze lub wprowadź
↪  'koniec': ")
    if (acct == 'koniec'): break
    if (len(acct) < 1):
        cur.execute('SELECT id, nazwa FROM Osoby WHERE pobrana=0
↪  LIMIT 1')
        try:
            (id, acct) = cur.fetchone()
        except:
            print('Nie znaleziono niepobranych kont Twittera')
            continue
    else:
        cur.execute('SELECT id FROM Osoby WHERE nazwa = ? LIMIT 1',
                    (acct, ))
        try:
            id = cur.fetchone()[0]
        except:
            cur.execute('''INSERT OR IGNORE INTO Osoby
                        (nazwa, pobrana) VALUES (?, 0)''', (acct, ))
            conn.commit()
            if cur.rowcount != 1:
                print('Błąd podczas wstawiania konta:', acct)
                continue
            id = cur.lastrowid

    url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count':
↪  '100'})
    print('Pobieranie konta', acct)
    try:
        connection = urllib.request.urlopen(url, context=ctx)
    except Exception as err:
        print('Błąd pobierania', err)
        break

    data = connection.read().decode()
    headers = dict(connection.getheaders())
```

```
    print('Pozostało', headers['x-rate-limit-remaining'])

    try:
        js = json.loads(data)
    except:
        print('Błąd parsowania JSONa')
        print(data)
        break

    # Debugowanie
    # print(json.dumps(js, indent=4))

    if 'users' not in js:
        print('Otrzymano nieprawidłowy JSON')
        print(json.dumps(js, indent=4))
        continue

    cur.execute('UPDATE Osoby SET pobrana=1 WHERE nazwa = ?', (acct,
↪   ))

    countnew = 0
    countold = 0
    for u in js['users']:
        friend = u['screen_name']
        print(friend)
        cur.execute('SELECT id FROM Osoby WHERE nazwa = ? LIMIT 1',
                    (friend, ))
        try:
            friend_id = cur.fetchone()[0]
            countold = countold + 1
        except:
            cur.execute('''INSERT OR IGNORE INTO Osoby (nazwa,
↪   pobrana)
                        VALUES (?, 0)''', (friend, ))
            conn.commit()
            if cur.rowcount != 1:
                print('Błąd podczas wstawiania konta:', friend)
                continue
            friend_id = cur.lastrowid
            countnew = countnew + 1
        cur.execute('''INSERT OR IGNORE INTO Obserwuje (id_od,
↪   id_do)
                    VALUES (?, ?)''', (id, friend_id))
    print('Nowe konta=', countnew, ' widziane ponownie=', countold)
    print('Pozostało', headers['x-rate-limit-remaining'])
    conn.commit()
cur.close()

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

Nasz program zaczyna się komplikować, ale ilustruje schematy programowania, których musimy użyć podczas korzystania z kluczy całkowitoliczbowych do łączenia tabel. Podstawowymi schematami są:

1. Tworzenie tabel z kluczami głównymi i ograniczeniami.
2. Gdy dla danej osoby mamy klucz logiczny (tj. nazwę konta) i potrzebujemy wartości `id` dla tej osoby, to w zależności od tego czy osoba ta znajduje się już w tabeli `Osoby` czy też jej tam nie ma: (1) szukamy osoby w tabeli `Osoby` i pobieramy dla niej wartość `id` lub (2) dodajemy osobę do tabeli `Osoby` i pobieramy wartość `id` dla nowo dodanego wiersza.
3. Wstawienie wiersza, który jest w stanie uchwycić związek "obserwuje".

Po kolei zajmiemy się każdym z tych schematów.

### 15.8.1. Ograniczenia w tabelach bazy danych

Projektując struktury naszych tabel możemy powiedzieć systemowi bazy danych, że chcielibyśmy aby egzekwował on od nas kilka zasad. Reguły te pomagają nam uniknąć popełniania błędów i wprowadzania do naszych tabel błędnych danych. Kiedy tworzymy nasze tabele:

```
cur.execute('''CREATE TABLE IF NOT EXISTS Osoby
    (id INTEGER PRIMARY KEY, nazwa TEXT UNIQUE, pobrana INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Obserwuje
    (id_od INTEGER, id_do INTEGER, UNIQUE(id_od, id_do))''')
```

wskazujemy, że kolumna `nazwa` w tabeli `Osoby` musi być unikalna (`UNIQUE`). Wskazujemy również, że kombinacja dwóch liczb w każdym wierszu tabeli `Obserwuje` musi być unikalna. Te ograniczenia powstrzymują nas przed popełnianiem błędów, takich jak dodawanie tego samego związku więcej niż jeden raz.

Możemy skorzystać z tych ograniczeń w poniższym kodzie:

```
cur.execute('''INSERT OR IGNORE INTO Osoby (nazwa, pobrana)
    VALUES ( ?, 0)''', ( friend, ) )
```

Dodajemy klauzulę `OR IGNORE` do naszej instrukcji `INSERT`, tak aby wskazać, że jeśli ten konkretny `INSERT` spowodowałby naruszenie zasady mówiącej, że "`nazwa` musi być unikalna", to system bazy danych może zignorować to polecenie `INSERT`. Używamy ograniczenia bazy danych jako siatki asekuracyjnej po to by upewnić się, że nie zrobimy przez przypadek czegoś nieprawidłowego.

Podobnie, poniższy kod zapewnia, że nie dodamy dw razy dokładnie tego samego związku w `Obserwuje`.

```
cur.execute('''INSERT OR IGNORE INTO Obserwuje
    (id_od, id_Do) VALUES (?, ?)''', (id, friend_id) )
```

Po prostu mówimy naszej bazie danych, by zignorowała naszą próbę wstawienia danych (`INSERT`), jeśli naruszałaby ona ograniczenie unikalności, które określiliśmy dla wierszy w `Obserwuje`.

### 15.8.2. Pobieranie i/lub wstawianie wiersza

Kiedy zachęcamy użytkownika do podania nazwy konta na Twitterze i jeśli podane przez niego konto istnieje, to musimy sprawdzić jego wartość `id`. Jeśli konto jeszcze nie istnieje w tabeli `Osoby`, to musimy wstawić rekord i uzyskać wartość `id` z wstawionego wiersza.

Jest to bardzo powszechny schemat postępowania i jest używany dwa razy w naszym programie. Kod ten pokazuje sposób w jaki szukamy `id` dla konta znajomego, gdy wyciągnęliśmy nazwę ekranową `screen_name` z węzła `user`.

Ponieważ z czasem będzie coraz bardziej prawdopodobne, że konto znajduje się już w naszej bazie danych, najpierw sprawdzamy, czy istnieje wpis w `Osoby` pomocą polecenia `SELECT`.

Jeśli wszystko pójdzie dobrze[2] w sekcji `try`, to pobieramy z tabeli wiersz za pomocą funkcji `fetchone()`, a następnie pobieramy pierwszy (i jedyny) element zwróconej krotki i zapisujemy go w zmiennej `friend_id`.

Jeśli `SELECT` się nie powiedzie, to kod `fetchone()[0]` też się nie powiedzie i działanie programu zostanie przeniesione do sekcji `except`.

```
friend = u['screen_name']
print(friend)
cur.execute('SELECT id FROM Osoby WHERE nazwa = ? LIMIT 1',
            (friend, ))
try:
    friend_id = cur.fetchone()[0]
    countold = countold + 1
except:
    cur.execute('''INSERT OR IGNORE INTO Osoby (nazwa, pobrana)
                VALUES (?, 0)''', (friend, ))
    conn.commit()
    if cur.rowcount != 1:
        print('Błąd podczas wstawiania konta:', friend)
        continue
    friend_id = cur.lastrowid
    countnew = countnew + 1
```

Jeśli znajdziemy się w kodzie sekcji `except`, to oznacza to po prostu, że wiersz nie został znaleziony w tabeli, więc musimy go tam wstawić. Używamy `INSERT OR IGNORE` tylko po to, by uniknąć błędów, a następnie wywołujemy `commit()`, by zmusić bazę danych do faktycznej aktualizacji danych. Po zakończeniu zapisywania danych, możemy sprawdzić zmienną `cur.rowcount` by sprawdzić ile wierszy zostało zmienionych. Staramy się wstawić pojedynczy wiersz, więc jeśli liczba zmienionych wierszy jest inna niż 1, to mamy do czynienia z błędem.

Jeśli polecenie `INSERT` się powiedzie, to możemy spojrzeć na zmienną `cur.lastrowid` aby dowiedzieć się jaką wartość baza danych przypisała do kolumny `id` w naszym nowo utworzonym wierszu.

[2] Ogólnie rzecz biorąc, gdy zdanie zaczyna się od "jeśli wszystko idzie dobrze", to zauważysz, że kod musi używać `try/except`.

### 15.8.3. Przechowywanie związku dotyczącego znajomości

Kiedy już poznamy wartość klucza zarówno dla danego użytkownika Twittera, jak i jego znajomego, to łatwo będzie nam wpisać dwie liczby całkowite do tabeli `Obserwuje`:

```
cur.execute('INSERT OR IGNORE INTO Obserwuje (id_od, id_do) VALUES
↪  (?, ?)',
    (id, friend_id) )
```

Zauważ, że pozwalamy bazie danych zadbać o to, by nie dopuścić do "podwójnego wstawiania" związku, tworząc tabelę z ograniczeniem unikalności, a następnie dodając `OR IGNORE` do naszej instrukcji `INSERT`.

Poniżej znajduje się przykładowe wynik naszego programu:

```
Podaj nazwę konta na Twitterze lub wprowadź 'koniec':
Nie znaleziono niepobranych kont Twittera
Podaj nazwę konta na Twitterze lub wprowadź 'koniec': drchuck
Pobieranie konta drchuck
(...)
Nowe konta= 100  widziane ponownie= 0
Pozostało 13
Podaj nazwę konta na Twitterze lub wprowadź 'koniec':
Pobieranie konta jimcollins
(...)
Nowe konta= 99  widziane ponownie= 1
Pozostało 12
Podaj nazwę konta na Twitterze lub wprowadź 'koniec':
Pobieranie konta jczetta
(...)
Nowe konta= 97  widziane ponownie= 3
Pozostało 11
Podaj nazwę konta na Twitterze lub wprowadź 'koniec': koniec
```

Zaczęliśmy od konta `drchuck`, a następnie pozwoliliśmy programowi automatycznie wybrać dwa następne konta do pobrania i dodania do naszej bazy danych.

Poniżej znajduje się wynik `twdump2.py`, który wyświetla wiersze tabel `Osoby` i `Obserwuje`:

```
Osoby:
(1, 'drchuck', 1)
(2, 'jimcollins', 1)
(3, 'jczetta', 1)
(4, 'fsf', 0)
(5, 'ubuntourist', 0)
...
297 wierszy.

Obserwuje:
(1, 2)
(1, 3)
```

```
(1, 4)
...
(2, 121)
(2, 122)
(2, 1)
(2, 123)
...
300 wierszy.
```

Możesz zauważyć pola `id`, `nazwa` i `pobrana` z tabeli `Osoby` oraz numery opisujące związki z tabeli `Obserwuje`. W tabeli `Osoby` widzimy, że pierwsze trzy osoby zostały już odwiedzone i ich dane zostały pobrane. Dane w tabeli `Obserwuje` wskazują, że `drchuck` (użytkownik 1) jest znajomym wszystkich osób pokazanych w pierwszych trzech wierszach. Ma to sens, ponieważ pierwszymi danymi, które pobraliśmy i przechowywaliśmy byli znajomi użytkownika `drchuck`. Jeśli wyświetlisz u siebie wszystkie wiersze tabeli `Obserwuje`, to zobaczysz również znajomych użytkowników o `id` równych 2 i 3.

## 15.9. Trzy rodzaje kluczy

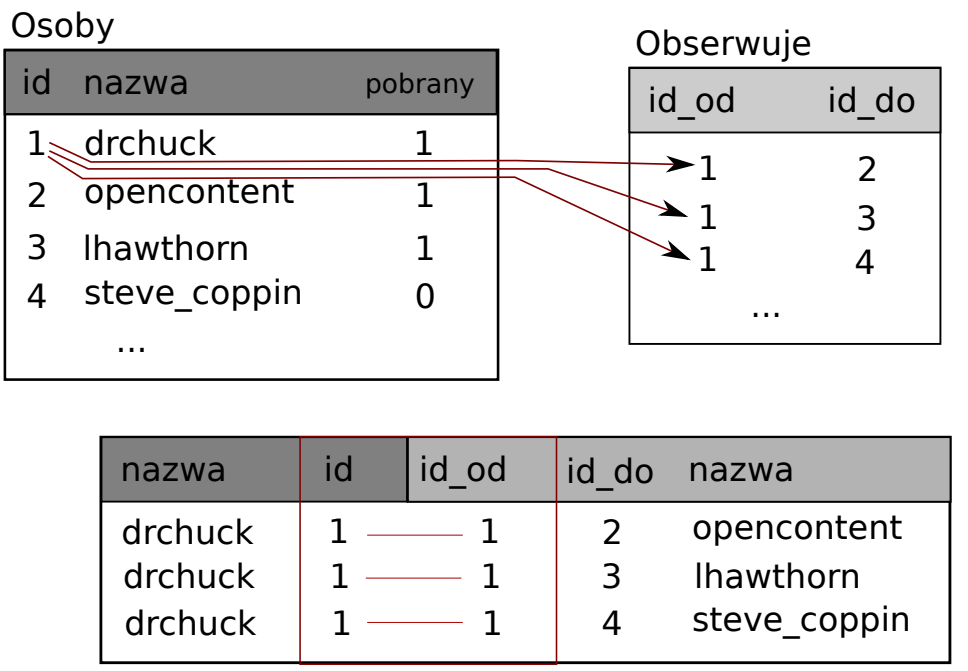
Teraz, gdy zaczęliśmy budować model danych umieszczając nasze dane w wielu połączonych ze sobą tabelach i łącząc wiersze w tych tabelach za pomocą *kluczy*, musimy przyjrzeć się trochę terminologii dotyczącej kluczy. Ogólnie rzecz biorąc, istnieją trzy rodzaje kluczy używanych w modelu bazy danych.

- *Klucz logiczny* to klucz, którego "rzeczywisty świat" może użyć do wyszukiwania wierszy. W naszym przykładowym modelu danych, pole `nazwa` jest kluczem logicznym. Jest to nazwa ekranowa użytkownika i rzeczywiście szukamy wiersza użytkownika kilka razy w programie używając pola `nazwa`. Często okazuje się, że sensowne jest dodanie ograniczenia `UNIQUE` do klucza logicznego. Ponieważ klucz logiczny jest tym, w jaki sposób z zewnętrznego świata szukamy danego wiersza, nie ma sensu zezwalać w tabeli na wiele wierszy o tej samej wartości.

- *Klucz główny* jest zazwyczaj liczbą, która jest przypisywana automatycznie przez bazę danych. Na ogół nie ma on żadnego znaczenia poza programem i jest używany tylko do łączenia razem wierszy z różnych tabel. Gdy chcemy odnaleźć wiersz w tabeli, zazwyczaj najszybszym sposobem na odnalezienie wiersza jest wyszukiwanie go za pomocą klucza głównego. Ponieważ klucze główne są liczbami całkowitymi, zajmują bardzo mało miejsca w pamięci i mogą być bardzo szybko porównywane lub sortowane. W naszym modelu danych, pole `id` jest przykładem klucza głównego.

- *Klucz obcy* jest zazwyczaj liczbą, która wskazuje na klucz główny powiązanego wiersza w innej tabeli. Przykładem klucza obcego w naszym modelu danych jest `id_od`.

Używamy konwencji nazewniczej polegającej na nazywaniu nazwy pola klucza głównego `id` i dodawaniu przedrostka `id_` do każdej nazwy pola, które jest kluczem obcym.




Rysunek 15.5: Łączenie tabel przy pomocy JOIN

## 15.10. Używanie operacji JOIN do pozyskiwania danych

Teraz, gdy zastosowaliśmy się do zasad normalizacji baz danych i mamy dane rozdzielone na dwie tabele połączone kluczami głównymi i obcymi, musimy być w stanie skonstruować polecenie `SELECT`, które ponownie złoży dane zawarte w tabelach.

SQL używa klauzuli `JOIN` do ponownego połączenia tabel. W klauzuli `JOIN` określasz pola, które są używane do ponownego połączenia wierszy między tabelami.

Poniżej znajduje się przykład polecenia `SELECT` z klauzulą `JOIN`:

```
SELECT * FROM Obserwuje JOIN Osoby
    ON Obserwuje.id_od = Osoby.id WHERE Osoby.id = 1
```

Klauzula `JOIN` wskazuje, że wybrane przez nas pola odpowiadają sobie w tabelach `Obserwuje` i `Osoby`. Klauzula `ON` wskazuje jak te dwie tabele mają zostać połączone: weź wiersze z `Obserwuje` i dodaj wiersz z `Osoby` tam, gdzie pole `id_od` w `Obserwuje` ma tę samą wartość co `id` w `Osoby`.

Wynikiem operacji `JOIN` jest stworzenie bardzo długich "meta-wierszy", które mają zarówno pola z tabeli `Osoby`, jak i pasujące pola z `Obserwuje`. Jeśli jest więcej niż jedno dopasowanie pomiędzy polem `id` z `Osoby` i `id_od` z `Obserwuje`, wtedy `JOIN` tworzy taki meta-wiersz dla *każdej* pasującej pary wierszy, dublując przy tym dane w razie potrzeby.

Poniższy kod pokazuje dane, które uzyskamy po kilkukrotnym uruchomieniu powyższego robota internetowego, działającego na kilku tabelach.

```
import sqlite3

conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()

cur.execute('SELECT * FROM Osoby')
count = 0
print('Osoby:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'wierszy.')

cur.execute('SELECT * FROM Obserwuje')
count = 0
print('\nObserwuje:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'wierszy.')

cur.execute('''SELECT * FROM Obserwuje JOIN Osoby
            ON Obserwuje.id_do = Osoby.id
            WHERE Obserwuje.id_od = 2''')
count = 0
print('\nPołączenia dla id=2:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'wierszy.')

cur.close()

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

W powyższym programie najpierw wyświetlamy po 5 wierszy z tabel `Osoby` i `Obserwuje`, a następnie wyświetlamy podzbiór danych z połączonych ze sobą tabel.

Wynik działania programu jest następujący:

```
Osoby:
(1, 'drchuck', 1)
(2, 'jimcollins', 1)
(3, 'jczetta', 1)
(4, 'fsf', 0)
(5, 'ubuntourist', 0)
297 wierszy.
```

```
Obserwuje:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
300 wierszy.

Połączenia dla id=2:
(2, 1, 1, 'drchuck', 1)
(2, 102, 102, 'CherieInDC', 0)
(2, 103, 103, 'optionslion8', 0)
(2, 104, 104, 'ForgottenDC', 0)
(2, 105, 105, 'michael_saylor', 0)
100 wierszy.
```

Widzimy kolumny z tabel `Osoby` i `Obserwuje`, a ostatni zestaw wierszy jest wynikiem polecenia `SELECT` z klauzulą `JOIN`.

W ostatniej operacji `SELECT` szukamy kont, które są znajomymi "jimcollins" (tj. `Osoby.id=2`).

W każdym z "meta-wierszy" ostatniej operacji `SELECT`, pierwsze dwie kolumny są z tabeli `Obserwuje`, po których następują trzy kolumny z tabeli `Osoby`. Możesz również zobaczyć, że druga kolumna (`Obserwuje.id_do`) pasuje do trzeciej kolumny (`Osoby.id`) w każdym z połączonych "meta-wierszy".

## 15.11. Podsumowanie

Powyższy rozdział zawiera wiele informacji na temat podstaw korzystania z bazy danych w Pythonie. Pisanie kodu do wykorzystującego bazy danych do przechowywania informacji jest bardziej skomplikowane niż używanie słowników Pythona lub zwykłych pliki, więc nie ma większego powodu do korzystania z bazy danych, no chyba że Twoja aplikacja naprawdę potrzebuje możliwości jakie dają bazy. Są sytuacje, w których baza danych może być całkiem użyteczna: (1) gdy Twoja aplikacja musi dokonać małej liczby losowych aktualizacji w ramach dużego zbioru danych, (2) gdy Twoje dane są tak duże, że nie mieszczą się w słowniku i musisz wielokrotnie wyszukiwać dane z tego zbioru, lub (3) gdy masz długotrwały proces, który chcesz zatrzymać i ponownie uruchomić oraz jednocześnie zachować dane z jednego uruchomienia tak, aby były dostępne w drugim.

Możesz zbudować prostą bazę danych z pojedynczą tabelą pasującą do wielu potrzeb Twojej aplikacji, ale większość problemów będzie jednak wymagała kilku tabel i połączeń/powiązań między wierszami różnych tabel. Kiedy zaczynasz tworzyć połączenia między tabelami, ważne jest by wcześniej zrobić jakiś przemyślany projekt tabel i postępować zgodnie z zasadami normalizacji bazy danych, tak aby jak najlepiej wykorzystać jej możliwości. Ponieważ główną motywacją do korzystania z bazy danych jest to, że masz do czynienia z dużą ilością danych, ważne jest też by modelować swoje dane efektywnie, tak aby Twoje programy działały jak najszybciej.

## 15.12. Debugowanie

Jednym z powszechnych schematów postępowania podczas tworzenia w Pythonie programu do łączenia się z bazą danych SQLite jest uruchomienie programu i sprawdzenie wyników za pomocą przeglądarki bazy danych SQLite. Przeglądarka pozwala na szybkie sprawdzenie, czy Twój program działa poprawnie.

Musisz być tutaj ostrożny, ponieważ SQLite dba o to, by dwa programy nie zmieniały jednocześnie tych samych danych. Na przykład, jeśli otworzysz bazę danych w przeglądarce i dokonasz zmiany w bazie danych, ale w przeglądarce nie naciśniesz jeszcze przycisku "zapisz", to przeglądarka "zablokuje" plik bazy danych i uniemożliwi dostęp do niego innym programom. W szczególności, Twój program napisany Pythonie nie będzie w stanie uzyskać dostępu do zablokowanego pliku.

Rozwiązaniem jest więc zamknięcie przeglądarki bazy danych lub skorzystanie z menu *File*, tak aby zamknąć bazę danych w przeglądarce przed próbą dostępu do bazy danych z Pythona. Dzięki temu można uniknąć problemu niepowodzenia kodu Pythona, ponieważ baza danych będzie wtedy odblokowana.

## 15.13. Słowniczek

**atrybut** Jedna z wartości w krotce. Częściej nazywany "kolumną" lub "polem".

**indeks** Dodatkowe dane, które oprogramowanie bazy danych przechowuje w postaci wierszy i wstawia dodatkowo do tabeli, dzięki czemu wyszukiwanie informacji odbywa się bardzo szybko.

**klucz główny** Wartość numeryczna przypisana do każdego wiersza tabeli, która jest używana do odwołania się z innej tabeli do tego konkretnego wiersza. Często baza danych jest skonfigurowana tak, aby automatycznie przydzielać klucze główne w miarę wstawiania kolejnych wierszy.

**klucz logiczny** Klucz, którego "świat zewnętrzny" używa do wyszukiwania konkretnego wiersza. Na przykład w tabeli z kontami użytkowników, adres email danej osoby może być dobrym kandydatem na klucz logiczny dla danych o użytkowniku.

**klucz obcy** Klucz numeryczny, który wskazuje na klucz główny jakiegoś wiersza w innej tabeli. Klucze obce ustanawiają związki między wierszami przechowywanymi w różnych tabelach.

**krotka** Pojedynczy wpis w tabeli bazy danych, który jest zbiorem atrybutów. Częściej używa się określenia "wiersz".

**kursor** Kursor pozwala na programistyczne wykonywanie poleceń SQL w bazie danych i pobieranie danych z bazy. Kursor jest podobny do gniazda połączeń sieciowych lub uchwytu pliku.

**normalizacja** Projektowanie modelu danych w taki sposób, aby żadne dane nie były powielone. Każdą porcję danych przechowujemy w bazie tylko w jednym miejscu i z innych miejsc odwołujemy się do niej za pomocą klucza obcego.

**ograniczenie** Gdy mówimy bazie danych, by egzekwowała jakąś regułę/zasadę na kolumnie lub wierszu tabeli. Częstym ograniczeniem jest wymuszenie by w danej kolumnie nie było zduplikowanych wartości (tzn. by wszystkie wartości były unikalne).

**przeglądarka bazy danych** Oprogramowanie, które pozwala na bezpośrednie połączenie z bazą danych i bezpośrednie zarządzanie bazą danych, bez konieczności pisania osobnego programu.

**relacja** Obszar w obrębie bazy danych, który zawiera krotki i atrybuty. Częściej używa się określenia "tabela".

# Rozdział 16

# Wizualizacja danych

Nauczyliśmy się pewnych aspektów Pythona, sprawdzaliśmy jak go używać Pythona oraz jak korzystać z sieci i baz danych w celu zarządzania danymi.

W poniższym rozdziale przyjrzymy się trzem kompletnym aplikacjom, które łączą wszystkie te rzeczy razem, tak aby zarządzać danymi i je wizualizować. Możesz później użyć tych aplikacji jako przykładowego kodu, który pomoże Ci rozpocząć rozwiązywanie Twojego problemu.

Każda z tych aplikacji jest plikiem ZIP, który możesz pobrać na swój komputer, rozpakować i uruchomić.

## 16.1. Budowanie mapy OpenStreetMap z geokodowanych danych

W tym projekcie wykorzystamy API geokodowania OpenStreetMap (usługa Nominatim), po to by zamienić wpisane przez użytkowników nazwy uczelni na lokalizacje geograficzne[1], a następnie umieścimy przetworzone dane na mapie OpenStreetMap.

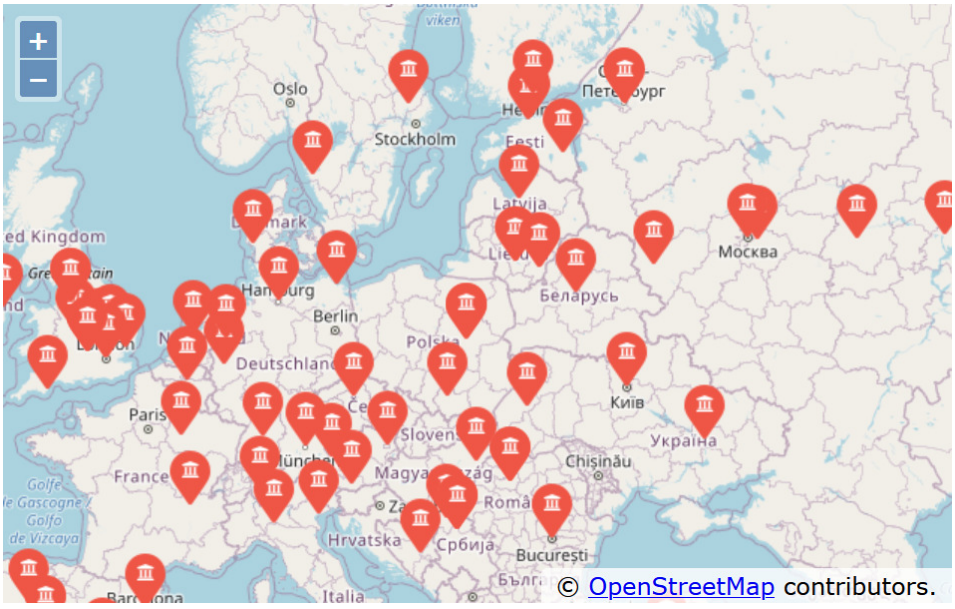
Aby rozpocząć, pobierz aplikację z poniższego adresu:

pl.py4e.com/code3/geodata.zip

W warunkach korzystania z usługi Nominatim jest wskazanie by ogarniczyć się do maksymalnie jednego zapytania na sekundę (usługa jest darmowa, stąd też gdybyśmy generowali bardzo dużą liczbę zapytań w krótkim czasie, to prawdopodobnie szybko zablokowano by nam dostęp do API). Nasze zadanie dzielimy na dwie fazy.

W pierwszej fazie bierzemy nasze dane wejściowe z pliku *where.data* i odczytujemy je wiersz po wierszu, odczytując przy tym zgeokodowaną odpowiedź serwera Nominatim i przechowujemy ją w bazie danych (plik *geodata.sqlite*). Zanim użyjemy API geokodowania, po prostu sprawdzamy, czy w naszej bazie ("pamięci podręcznej") mamy już dane dla tego konkretnego wiersza, dzięki czemu w przypadku ponownego uruchomienia programu nie będziemy musieli drugi razy wysyłać zapytania do API.

[1] Należy mieć na uwadze, że taka zamiana tekstu na dane geolokalizacyjne nie zawsze daje precyzyjne i poprawne wyniki, co będzie można też zauważyć w części wyników tej aplikacji.




Rysunek 16.1: Mapa OpenStreetMap

W dowolnym momencie możesz uruchomić cały proces od początku, po prostu usuwając wygenerowany plik *geodata.sqlite*.

Uruchom program *geoload.py*. Program ten odczyta wiersze wejściowe z pliku *where.data* i dla każdego wiersza sprawdzi, czy jest on już w bazie danych, a jeśli nie mamy danych dla przetwarzanej lokalizacji, to wywoła on zapytanie API geokodowania aby pobrać dane i przechowywać je w bazie SQLite.

Oto przykładowe uruchomienie po tym, jak w bazie danych znajdują się już jakieś dane:

```
Znaleziono w bazie  AGH University of Science and Technology

Znaleziono w bazie  Academy of Fine Arts Warsaw Poland

Znaleziono w bazie  American University in Cairo

Znaleziono w bazie  Arizona State University

Znaleziono w bazie  Athens Information Technology

Pobieranie https://nominatim.openstreetmap.org/search.php?q=University+of+Pretoria&format=geojs
Pobrano 954 znaków {"type":"FeatureColl

Pobieranie https://nominatim.openstreetmap.org/search.php?q=University+of+Salamanca&format=geoj
Pobrano 822 znaków {"type":"FeatureColl

...
```

Pierwsze pięć lokalizacji znajduje się już w bazie danych, a więc są one pomijane.

Program przetwarza dane do momentu, w którym znajdzie niezapisane lokalizacje i zaczyna o nie odpytywać API.

Plik *geoload.py* może zostać zatrzymany w dowolnym momencie, a ponadto kod zawiera licznik (zmienna *count*), którego można użyć do ograniczenia liczby połączeń do API geokodowania w danym uruchomieniu programu.

Po załadowaniu danych do *geodata.sqlite*, możesz je zwizualizować za pomocą programu *geodump.py*. Program ten odczytuje bazę danych i zapisuje plik *where.js* zawierający lokalizacje, szerokości i długości geograficzne w postaci wykonywalnego kodu JavaScript. Pobrany przez Ciebie plik ZIP zawiera już wygenerowany *where.js*, ale możesz go wygenerować jeszcze raz aby sprawdzić działanie programu *geodump.py*.

Uruchomienie programu *geodump.py* odbywa się w następujący sposób:

```
Akademia Górniczo-Hutnicza ... Polska 50.065703299999996 19.918958667058632
Akademia Sztuk Pięknych ... Polska 52.2397515 21.015564130658333
...
260 wierszy zapisano do where.js
Otwórz w przeglądarce internetowej plik where.html aby obejrzeć dane.
```

Plik *where.html* składa się z kodu HTML i JavaScript, które służą do wizualizacji mapy OpenStreetMap przy pomocy biblioteki OpenLayers. Strona odczytuje najświeższe dane z pliku *where.js* po to by uzyskać dane niezbędne do wizualizacji. Oto format pliku *where.js*:

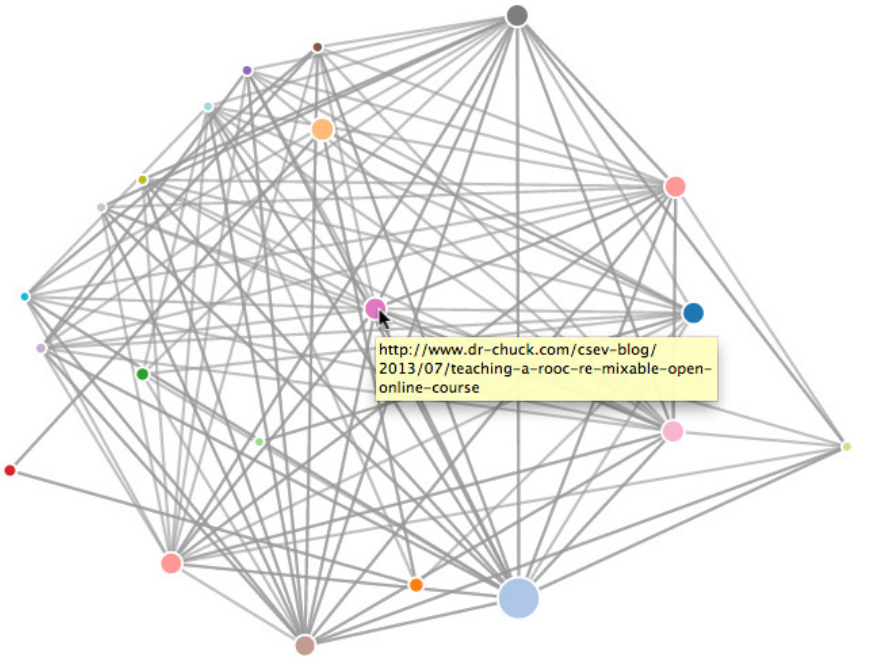
```
myData = [
[50.065703299999996,19.918958667058632, 'Akademia Górniczo-Hutnicza,
↪  ... , Polska'],
[52.2397515,21.015564130658333, 'Akademia Sztuk Pięknych, ... ,
↪  Polska'],
   ...
];
```

Jest to lista list zapisana w języku JavaScript. Składnia listy w języku JavaScript jest bardzo podobna do składni Pythona.

By zobaczyć lokalizacje na mapie, otwórz plik *where.html* w przeglądarce internetowej. Możesz najechać kursorem na każdą pinezkę mapy i na nią kliknąć, tak aby znaleźć lokalizację, którą zwróciło API kodowania dla danych wejściowych wprowadzonych przez użytkownika. Jeżeli po otwarciu pliku *where.html* nie widzisz żadnych danych, sprawdź czy w przeglądarce jest włączony JavaScript lub w konsoli deweloperskiej swojej przeglądarki sprawdź czy są jakieś błędy.

## 16.2. Wizualizacja sieci i połączeń

W poniższej aplikacji będziemy wykonywać niektóre funkcje znajdujące się w mechaniźmie wyszukiwarki internetowej. Najpierw przeczeszemy mały fragment sieci internetowej, uruchomimy uproszczoną wersję algorytmu Google PageRank by




Rysunek 16.2: Algorytm PageRank

określić strony, do których jest najwięcej odniesień z innych stron (czyli by wskazać które strony są potencjalnie najistotniejsze), a następnie zwizualizujemy rangę strony i połączenia w naszym małym zakątku sieci. Aby utworzyć wizualizacje wykorzystamy bibliotekę JavaScript D3 https://d3js.org/.

Możesz pobrać i rozpakować poniższą aplikację:

pl.py4e.com/code3/pagerank.zip

Pierwszy program (*spider.py*) wczytuje stronę internetową i umieszcza serię stron do bazy danych (*spider.sqlite*), rejestrując przy tym odnośniki (linki) pomiędzy stronami. W każdej chwili możesz zrestartować cały proces, usuwając plik *spider.sqlite* i uruchamiając ponownie *spider.py*.

```
Wpisz adres internetowy lub wciśnij Enter:
['https://www.dr-chuck.com']
Ile stron: 25
1 https://www.dr-chuck.com (8386) 4
4 https://www.dr-chuck.com/dr-chuck/resume/index.htm (1855) 9
12 https://www.dr-chuck.com/dr-chuck/resume/pictures/index.htm (1827) 5
...
Ile stron:
```

W powyższym przykładowym uruchomieniu wskazaliśmy naszemu robotowi, by sprawdził domyślną stronę i pobrał 25 stron. Jeśli zrestartujesz program i wskażesz by przeszukał więcej stron, to nie będzie on ponownie przeszukiwał stron już znajdujących się w bazie danych. Po ponownym uruchomieniu robot przechodzi do

losowej, niesprawdzonej jeszcze strony i zaczyna tam swoją pracę. Tak więc każde kolejne uruchomienie *spider.py* jest dodaje tylko nowe strony.

```
Wpisz adres internetowy lub wciśnij Enter:
Kontynuowanie istniejącego indeksowania stron. Usuń spider.sqlite, aby rozpocząć nowe
['https://www.dr-chuck.com']
Ile stron: 3
22 https://www.dr-chuck.com/csev-blog/category/uncategorized (91096) 61
27 https://www.dr-chuck.com/csev-blog/2014/09/how-to-achieve-vendor-lock-in-with-a-le
30 https://www.dr-chuck.com/csev-blog/2020/08/styling-tsugi-koseu-lessons (33522) 21
Ile stron:
```

Możesz mieć wiele punktów startowych w tej samej bazie danych - w programie nazywane są one "witrynami". Robot internetowy jako następną stronę do sprawdzenia wybiera losową stronę spośród wszystkich nieodwiedzonych linków na wszystkich witrynach.

Jeżeli chcesz wyświetlić zawartość bazy *spider.sqlite*, możesz uruchomić *spdump.py*:

```
(16, None, 1.0, 2, 'https://www.dr-chuck.com/csev-blog')
(15, None, 1.0, 30, 'https://www.dr-chuck.com/csev-blog/2020/08/styling-tsugi-koseu-l
(15, None, 1.0, 22, 'https://www.dr-chuck.com/csev-blog/category/uncategorized')
...
22 wierszy.
```

Program dla danej strony pokazuje liczbę przychodzących linków, stary współczynnik PageRank, nowy współczynnik PageRank, id strony oraz adres URL. Program *spdump.py* pokazuje tylko te strony, które mają co najmniej jedno odniesienie z innych stron.

Gdy będziesz już miał kilka stron w swojej bazie danych, to za pomocą programu *sprank.py* możesz uruchomić algorytm obliczania współczynnika PageRank. Musisz tylko podać ile iteracji algorytmu program ma wykonać.

```
Ile iteracji: 2
1 0.720326643053916
2 0.34992366601870745
[(1, 0.6196280991735535), (2, 2.4944679374657728), (3, 0.6923553719008263), (4, 1.101
```

Możesz ponownie wyświetlić zawartość bazy aby zobaczyć, że współczynnik PageRank dla stron został zaktualizowany:

```
(16, 1.0, 2.4944679374657728, 2, 'https://www.dr-chuck.com/csev-blog')
(15, 1.0, 2.1360764832409846, 30, 'https://www.dr-chuck.com/csev-blog/2020/08/styling
(15, 1.0, 2.449518442516278, 22, 'https://www.dr-chuck.com/csev-blog/category/uncateg
...
22 wierszy.
```

Możesz uruchamiać *sprank.py* tyle razy, ile chcesz, a program po prostu poprawi obliczenie współczynnika PageRank za każdym razem, gdy go uruchomisz. Możesz nawet uruchomić *sprank.py* kilka razy, a następnie dodać kilka kolejnych stron

poprzez *spider.py*, a następnie znów uruchomić *sprank.py*, by dalej przeliczyć wartości PageRank. Wyszukiwarka internetowa zazwyczaj przez cały czas uruchamia zarówno programy do indeksowania stron, jak i do tworzenia rankingu.

Jeżeli chcesz uruchomić obliczenia PageRank od początku bez ponownego przejścia robotem po stronach, to możesz użyć programu *spreset.py*, a następnie możesz uruchomić ponownie *sprank.py*.

Wynik uruchomienia *spreset.py*:

```
Wszystkie strony mają ustawiony współczynnik PageRank na 1.0
```

Ponowne uruchomienie *sprank.py*:

```
Ile iteracji: 50
1 0.720326643053916
2 0.34992366601870745
3 0.17895552923503424
4 0.11665048143652895
...
46 3.579258334100184e-05
47 3.0035450290624035e-05
48 2.520367345856324e-05
49 2.114963873141301e-05
50 1.7747381915049988e-05
[(1, 9.945248881666563e-05), (2, 3.205252622657907), (3, 0.00012907931109952867), (4, 0.000371
```

Dla każdej iteracji algorytmu PageRank program wypisuje średnią zmianę współczynnika na stronę. Początkowo sieć jest dość niezbalansowana, więc poszczególne wartości rankingu bardzo się zmieniają pomiędzy kolejnymi iteracjami. Jednak w kilku kolejnych iteracjach algorytm zbiega szybko do końcowego wyniku. Powinieneś uruchomić *sprank.py* na tyle długo, by kolejne wartości generowane przez aglorytm nie miały już zbyt dużych różnic.

Jeśli chcesz zwizualizować strony, które aktualnie najwyżej znajdują się w rankingu, uruchom program *spjson.py*. Odczytuje on bazę danych i zapisuje dane dotyczące najbardziej linkowanych stron w formacie JSON, który może być obejrzany w przeglądarce internetowej.

```
Tworzenie JSONa w pliku spider.js...
Ile węzłów? 30
Otwórz force.html w przeglądarce internetowej by zobaczyć wizualizację
```

Możesz obejrzeć wynik otwierając plik *force.html* w swojej przeglądarce internetowej. Pokazuje on automatyczny układ węzłów (stron) i połączeń między nimi. Możesz kliknąć i przeciągnąć dowolny węzeł, a także dwukrotnie kliknąć na węzeł, by wyświetlić adres URL, który jest reprezentowany przez ten węzeł. Wielkość węzła reprezentuje jego istotność, tzn. że wiele innych stron do linkuje do tej strony.

Jeżeli uruchomisz ponownie inne narzędzia tej aplikacji, uruchom ponownie *spjson.py* i ośwież stronę *force.html* w przeglądarce, tak aby uzyskać nowe dane umieszczone w *spider.json*.


Rysunek 16.3: Chmura wyrazów z listy mailingowej deweloperów projektu Sakai

## 16.3. Wizualizacja danych z e-maili

W niektórych rozdziałach książki i ćwiczeniach pojawiały się pliki *mbox-short.txt* i *mbox.txt*, które zawierały wiadomości mailowe. Nadszedł czas by naszą analizę danych dotyczącą poczty elektronicznej na przenieść kolejny poziom.

Zdarza się, że trzeba pobrać z serwerów dane dotyczące poczty e-mail. Może to zająć sporo czasu, a dane mogą być niespójne, pełne błędów i wymagać wielu poprawek lub czyszczenia. W tej sekcji będziemy pracować z najbardziej złożoną jak dotąd aplikacją, pobierzemy prawie gigabajt danych i zwizualizujemy te dane.

Możesz pobrać aplikację z:

pl.py4e.com/code3/gmane.zip

Będziemy korzystać z danych umieszczonych na bezpłatnej usłudze archiwizacji listy mailingowej o nazwie Gmane. Usługa była bardzo popularna wśród projektów open source, ponieważ zapewniała ładne i łatwe do przeszukiwania archiwum ich aktywności emailowej. Usługa posiadała również bardzo liberalną politykę dostępu do swoich danych poprzez swoje API.

Aby nie obciążać innych serwerów, moja własna kopia wiadomości dostępna jest pod adresem:

https://mbox.dr-chuck.net/

Gdy dane poczty elektronicznej Sakai były pobierane za pomocą tej aplikacji, to wytworzyło to prawie gigabajt danych i cały proces pobierania trwał kilka dni. Plik *README.txt* w powyższym pliku ZIP zawiera instrukcje opisujące pobranie

archiwalnej kopii *content.sqlite*, która zawiera większość korpusu tekstowego poczty elektronicznej projektu Sakai (do marca 2015 r.). W związku z tym nie musisz przy pomocy robota internetowego ściągać danych przez pięć dni. Jeśli pobierzesz ww. archiwalną wersję bazy SQLite, to możesz dodatkowo uruchomić robota internetowego by uzupełnić dane o najnowsze wiadomości mailowe.

Pierwszym krokiem jest przeskanowanie repozytorium Gmane. Główny adres URL jest zapisany w *gmane.py* wzmiennej *baseurl* i wskauzje na listę mailingową deweloperów projektu Sakai. Twój robot może skanować inne repozytorium - wystarczy, że zmienisz wspomniany adres adres URL. Jeśli zmienisz adres URL, to skasuj również plik *content.sqlite*.

Plik *gmane.py* działa jako odpowiedzialny robot internetowy, zapisujący w pamięci pobrane dane i odczekujący sekundę po pobraniu stu wiadomości. Program przechowuje wszystkie swoje dane w bazie, może być przerywany i uruchamiany ponownie tak często, jak to konieczne. Pobranie wszystkich danych może potrwać wiele godzin. Może być więc konieczne kilkukrotne ponowne uruchomienie tego programu.

Oto wynik uruchomienia *gmane.py*, który pobiera dziesięć ostatnich wiadomości z listy deweloperów projektu Sakai:

```
Ile wiadomości: 10
http://mbox.dr-chuck.net/sakai.devel/59643/59644 17553
    matthew@longsight.com 2015-03-20T16:27:12-04:00 re: [building sakai] sakai 10 bulding error
http://mbox.dr-chuck.net/sakai.devel/59644/59645 13128
    alberto.olivamolina@gmail.com 2015-03-20T16:36:12+01:00 re: [building sakai] sakai 10 buldi
http://mbox.dr-chuck.net/sakai.devel/59645/59646 7557
    eric.duquenoy@univ-littoral.fr 2015-03-20T16:52:24+01:00 [building sakai] lti and sakai gro
http://mbox.dr-chuck.net/sakai.devel/59646/59647 1
...
```

Program skanuje zawartość *content.sqlite* w poszukiwaniu numeru pierwszej niepobranej jeszcze wiadomości. Robot ściąga dane tak długo aż nie pobierze pożądanej liczby wiadomości lub gdy dotrze do strony, która nie wydaje się być odpowiednio sformatowaną wiadomością.

Czasami może brakować pewnych wiadomości. Być może administratorzy Gmane mogli usuwać wiadomości, a może się te wiadomości gdzieś przepadły. Jeżeli Twój robot się zatrzyma, a wydaje się, że trafił na właśnie taką brakującą wiadomość, to przejdź do przeglądarki bazy SQLite i dodaj wiersz z brakującym id, pozostawiając wszystkie pozostałe pola puste, a następnie uruchom ponownie *gmane.py*. Dzięki temu robot będzie mógł kontynuować pracę. Wstawione ręcznie puste wiadomości będą ignorowane w następnej fazie procesu.

Jedną z przyjemnych w tym wszystkim rzeczy jest to, że gdy już pobierzesz wszystkie wiadomości i będziesz je miał w *content.sqlite*, to po jakimś czasie będziesz mógł ponownie uruchomić *gmane.py*, po to by uzyskać tylko nowe wiadomości, które zostały ostatnio wysłane na listę mailingową.

Dane zawarte w bazie *content.sqlite* są dość surowe, z niewydajnym modelem danych, a ponadto nie są skompresowane. Jest to celowe, ponieważ pozwala Ci oberzeć plik *content.sqlite* w przeglądarce bazy SQLite by usunąć problemy związane procesem

pobierania danych. Generalnie byłby to zły pomysł aby uruchomić jakiekolwiek zapytanie SQL na tej bazie danych, ponieważ jego wykonanie byłyby dość powolne.

Druga faza polega na uruchomieniu programu *gmodel.py*. Program ten odczytuje surowe dane z *content.sqlite* i tworzy w pliku *index.sqlite* oczyszczoną i dobrze zamodelowaną wersję danych. Plik ten będzie znacznie mniejszy (często ok. 10 razy mniejszy) niż *content.sqlite*, ponieważ kompresuje on również nagłówek i treść wiadomości.

Za każdym razem gdy uruchomimy *gmodel.py*, program usuwa i przebudowuje plik *index.sqlite*, pozwalając Ci dostosować jego parametry i edytować tabele mapowania w *content.sqlite*, tak aby dopasować odpowiednio proces czyszczenia danych. Poniżej mamy przykładowe uruchomienie *gmodel.py*. Wypisuje on linię za każdym razem, gdy przetwarzanych jest 250 wiadomości mailowych, dzięki czemu widzisz postęp pracy. Jest to o tyle istotne, ponieważ program ten może działać przez pewien dłuższy czas przetwarzając prawie gigabajt danych wejściowych.

```
Załadowano nadawców: 1588 , mapowania: 29 , mapowania dns: 1
1 2005-12-08T23:34:30-06:00 ggolden22@mac.com
251 2005-12-22T10:03:20-08:00 tpamsler@ucdavis.edu
501 2006-01-12T11:17:34-05:00 lance@indiana.edu
751 2006-01-24T11:13:28-08:00 vrajgopalan@ucmerced.edu
...
```

Program *gmodel.py* realizuje szereg zadań związanych z czyszczeniem danych.

Nazwy domen są obcięte do dwóch poziomów dla domen *.com*, *.org*, *.edu* i *.net*. Inne nazwy domen są obcięte do trzech poziomów. Tak więc *si.umich.edu* staje się *umich.edu*, a *caret.cam.ac.uk* staje się *cam.ac.uk*. Adresy e-mail są również wymuszone na zapis małymi literami, a niektóre z adresów *@gmane.org* jak np.

```
arwhyte-63aXycvo3TyHXe+LvDLADg@public.gmane.org
```

są konwertowane na rzeczywisty adres za każdym razem, gdy w korpusie wiadomości znajduje się odpowiadający mu rzeczywisty adres e-mail.

W bazie danych *mapping.sqlite* znajdują się dwie tabele, które pozwalają na mapowanie zarówno nazw domen, jak i poszczególnych adresów e-mail, które zmieniają się w ciągu całego okresu życia listy mailingowej. Na przykład, Steve Githens użył następujących adresów e-mail, ponieważ zmieniał pracę w ciągu całego życia listy mailingowej deweloperów projektu Sakai:

```
s-githens@northwestern.edu
sgithens@cam.ac.uk
swgithen@mtu.edu
```

Jeśli chcemy, to możemy w *mapping.sqlite* do tabeli mapowania nadawców *Mapping* dodać dwa wpisy, tak by *gmodel.py* mapował wszystkie trzy adresy na jeden adres:

```
s-githens@northwestern.edu -> swgithen@mtu.edu
sgithens@cam.ac.uk -> swgithen@mtu.edu
```

Jeżeli istnieje wiele nazw DNS, które chcesz zmapować na jeden DNS, to możesz również dokonać podobnych wpisów w tabeli *DNSMapping*. Przykładowo:

```
iupui.edu -> indiana.edu
```

dzięki czemu wszystkie konta z różnych kampusów Uniwersytetu Indiany są śledzone razem.

Możesz uruchamiać *gmodel.py* wielokrotnie, za każdym razem gdy spojrzysz na dane i dodasz mapowania, tak by dane do analizy były czystsze i bardziej przejrzyste. Kiedy skończysz, w pliku *index.sqlite* będziesz miał ładnie zaindeksowaną wersję e-maili. Jest to plik, którego możesz użyć do dalszej analizy danych. Z tym plikiem analiza będzie naprawdę szybka.

Pierwszą, najprostszą analizą jest określenie "kto wysłał najwięcej wiadomości?" i "która organizacja wysłała najwięcej listów"? Odpowiedzi na te pytania uzyskamy za pomocą progrmau *gbasic.py*:

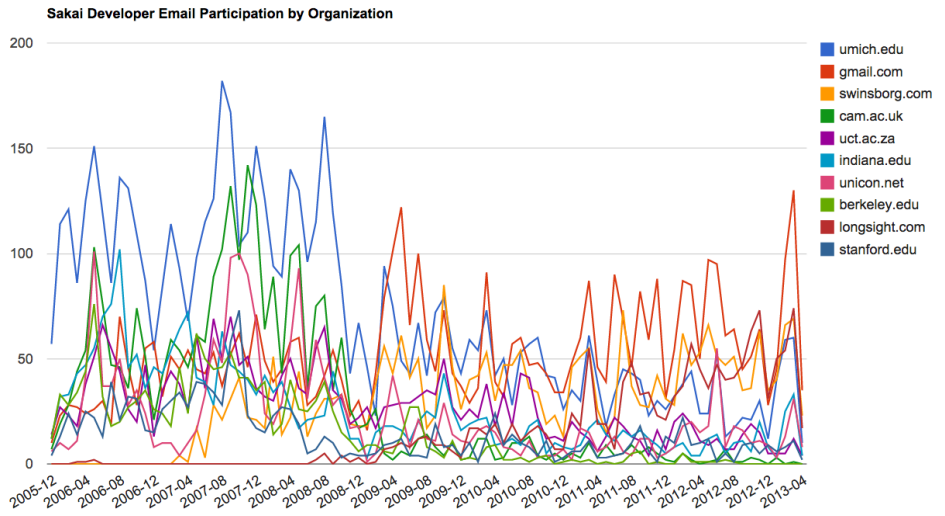
```
Ile wyświetlić? 5
Załadowanych wiadomości= 51330 tematów= 25033 nadawców= 1584

Lista 5 najczęstszych użytkowników
steve.swinsburg@gmail.com 2657
azeckoski@unicon.net 1742
ieb@tfd.co.uk 1591
csev@umich.edu 1304
david.horwitz@uct.ac.za 1184

Lista 5 najczęstszych organizacji
gmail.com 7339
umich.edu 6243
uct.ac.za 2451
indiana.edu 2258
unicon.net 2055
```

Zauważ jak szybko działa *gbasic.py* w porównaniu z *gmane.py* czy nawet *gmodel.py*. Wszystkie pracują na tych samych danych, ale *gbasic.py* używa skompresowanych i znormalizowanych danych znajdujących się w *index.sqlite*. Jeśli masz dużo danych do przetworzenia, to wieloetapowy proces tworzenia narzędzi do analizy (taki jak ten w tej aplikacji) może zająć Ci trochę więcej czasu, ale z drugiej strony zaoszczędzi Ci bardzo dużo czasu gdy naprawdę zaczniesz eksplorować i wizualizować swoje dane.

Poprzez program *gword.py* możesz stworzyć prostą wizualizację częstości występowania słów w tematach wiadomości:

```
Zakres częstości: 33229 129
Wynik zapisano w gword.js
Otwórz gword.htm w przeglądarce internetowej by zobaczyć wizualizcję
```

W ten sposób powstaje plik *gword.js*, który możesz zwizualizować za pomocą *gword.htm*. Mamy utworzoną chmurę wyrazów podobną do tej, która znajduje się w grafice z początku tej sekcji.




Rysunek 16.4: Aktywność mailowa w projekcie Sakai z podziałem na organizacje

Druga wizualizacja jest tworzona przez program *gline.py*. Oblicza ona udział organizacji w wiadomościach mailowych w danym czasie.

```
Loaded messages= 51330 subjects= 25033 senders= 1584
Top 10 organizacji
['gmail.com', 'umich.edu', 'uct.ac.za', 'indiana.edu',
'unicon.net', 'tfd.co.uk', 'berkeley.edu', 'longsight.com',
'stanford.edu', 'ox.ac.uk']
Wynik zapisano w gline.js
Otwórz gline.htm by zwizualizować dane
```

Wynik jest zapisywany w pliku *gline.js*, który można zwizualizować przy użyciu strony *gline.htm*.

Podsumowując, jest to stosunkowo złożona i wyrafinowana aplikacja, posiadająca funkcje pozwalające na wyszukiwanie, czyszczenie i wizualizację prawdziwych danych.

# Dodatek A

# Wsparcie i współpraca nad książką

## A.1. Lista współtwórców "Python dla wszystkich"

Andrzej Wójtowicz

Szczegóły dotyczące poszczególnego wkładu w tworzenie książki możesz znaleźć na stronie:

https://github.com/andre-wojtowicz/py4e-pl/graphs/contributors

## A.2. Lista współtwórców "Python for Everybody"

Elliott Hauser, Stephen Catto, Sue Blumenberg, Tamara Brunnock, Mihaela Mack, Chris Kolosiwsky, Dustin Farley, Jens Leerssen, Naveen KT, Mirza Ibrahimovic, Naveen (@togarnk), Zhou Fangyi, Alistair Walsh, Erica Brody, Jih-Sheng Huang, Louis Luangkesorn, oraz Michael Fudge

Szczegóły dotyczące poszczególnego wkładu w tworzenie książki możesz znaleźć na stronie:

https://github.com/csev/py4e/graphs/contributors

## A.3. Lista współtwórców "Python for Informatics"

Bruce Shields w zakresie pierwszy wersji roboczych, Sarah Hegge, Steven Cherry, Sarah Kathleen Barbarow, Andrea Parker, Radaphat Chongthammakun, Megan

Hixon, Kirby Urner, Sarah Kathleen Barbrow, Katie Kujala, Noah Botimer, Emily Alinder, Mark Thompson-Kular, James Perry, Eric Hofer, Eytan Adar, Peter Robinson, Deborah J. Nelson, Jonathan C. Anthony, Eden Rassette, Jeannette Schroeder, Justin Feezell, Chuanqi Li, Gerald Gordinier, Gavin Thomas Strassel, Ryan Clement, Alissa Talley, Caitlin Holman, Yong-Mi Kim, Karen Stover, Cherie Edmonds, Maria Seiferle, Romer Kristi D. Aranas (RK), Grant Boyer, Hedemarrie Dussan

## A.4. Przedmowa do "Think Python"

### A.4.1. Dziwna historia "Think Python"

(Allen B. Downey)

W styczniu 1999 roku przygotowywałem się do prowadzenia wstępnych zajęć z programowania w języku Java. Miałem te zajęcia trzy razy i byłem sfrustrowany. Wskaźnik braku zaliczenia przedmiotu był zbyt wysoki i nawet dla studentów, którym się udało zdać, ogólny poziom uzyskanych wyników był zbyt niski.

Jednym z problemów, które widziałem, były książki. Były one zbyt grube, zawierały za dużo zbędnych szczegółów na temat Javy i nie zawierały wystarczająco dużo wskazówek na temat programowania wysokopoziomowego. Na dodatek wszystkie cierpiały z powodu efektu zapadni: zaczynały się łatwo, postępowały stopniowo, a potem gdzieś około rozdziału 5 wszyscy wpadali na dno. Studenci dostawaliby nowy materiał za szybko i w zbyt dużych ilościach, a ja spędziłbym resztę semestru zbierając pozostałości.

Dwa tygodnie przed pierwszym dniem zajęć postanowiłem napisać własną książkę. Moje cele były następujące:

- Trzymać zwięzłość. Lepiej, żeby studenci przeczytali 10 stron, niż nie przeczytali 50.
- Uważać na słownictwo. Starałem się minimalizować specjalistyczny żargon i definiować każdy termin przy pierwszym użyciu.
- Budować stopniowo. Aby uniknąć efektu zapadni, brałem najtrudniejsze tematy i dzieliłem je na serię małych kroków.
- Skupić się na programowaniu, a nie na języku programowania. Włączyłem do książki minimalny użyteczny podzbiór języka Java i pominąłem resztę.

Potrzebowałem tytułu, więc kapryśnie wybrałem *How to Think Like a Computer Scientist*.

Moja pierwsza wersja książki była surowa, ale działała. Studenci czytali i zrozumieli temat na tyle, że podczas zajęć mogłem spędzać czas na analizie trudniejszych i ciekawych tematów oraz (co najważniejsze) pozwalając studentom ćwiczyć.

Wydałem tę książkę na licencji GNU Free Documentation License, która pozwala jej użytkownikom ją kopiować, modyfikować i rozpowszechniać.

A teraz najfajniejsza część tej historii. Jeff Elkner, nauczyciel z liceum w Virginii, włączył moją książkę do swojego programu nauczania i przetłumaczył ją na Pythona. Wysłał mi kopię swojego tłumaczenia, a ja doznałem niezwykłego doświadczenia w nauce Pythona, czytając moją własną książkę.

Zrewidowaliśmy z Jeffem książkę, włączyliśmy do niej studium przypadku Chrisa Meyersa, a w 2001 roku wydaliśmy *How to Think Like a Computer Scientist: Learning with Python*, także na licencji GNU Free Documentation License. Już jako wydawnictwo Green Tea Press opublikowałem książkę i zacząłem ją sprzedawać w formie papierowej za pośrednictwem Amazon.com i księgarni uniwersyteckich. Inne książki z Green Tea Press są dostępne na stronie greenteapress.com.

W 2003 roku zacząłem uczyć w Olin College, gdzie po raz pierwszy musiałem uczyć Pythona. Kontrast w zestawieniu z Javą był uderzający. Studenci mieli mniej problemów, uczyli się więcej, pracowali nad bardziej interesującymi projektami i ogólnie mieli dużo więcej zabawy.

W ciągu ostatnich pięciu lat kontynuowałem rozwój książki, poprawiając przy tym błędy i niektóre przykłady oraz dodając nowe materiały, zwłaszcza w sekcji ćwiczeń. W 2008 roku rozpocząłem pracę nad dużą korektą - w tym samym czasie skontaktował się ze mną redaktor Cambridge University Press, który był zainteresowany wydaniem kolejnej edycji książki. Dobre wyczucie czasu!

Mam nadzieję, że praca z tą książką sprawi Ci przyjemność oraz pomoże Ci nauczyć się programować i myśleć, przynajmniej trochę, jak informatyk.

### A.4.2. Podziękowania za wkład w “Think Python”

(Allen B. Downey)

Po pierwsze i najważniejsze, dziękuję Jeffowi Elknerowi, który przetłumaczył moją książkę z Javy na Pythona, co zapoczątkowało ten projekt i wprowadziło mnie w to, co okazało się być moim ulubionym językiem.

Dziękuję również Chrisowi Meyersowi, który przyczynił się do powstania kilku sekcji w *How to Think Like a Computer Scientist.*

I dziękuję Fundacji Wolnego Oprogramowania za opracowanie GNU Free Documentation License, co pomogło mi w umożliwieniu współpracy z Jeffem i Chrisem.

Dziękuję również redaktorom Lulu, którzy pracowali nad *How to Think Like a Computer Scientist.*

Dziękuję wszystkim studentom, którzy pracowali z wcześniejszymi wersjami tej książki oraz wszystkim współtwórcom (wymienionym w załączniku), którzy przesłali poprawki i sugestie.

Dziękuję również mojej żonie, Lisie, za jej pracę nad tą książką, oraz Green Tea Press i wszystkim innym.

Allen B. Downey
Needham MA

Allen Downey zajmuje stanowisko Associate Professor of Computer Science w Franklin W. Olin College of Engineering.

## A.5. Lista współtwórców “Think Python”

(Allen B. Downey)

W ciągu ostatnich kilku lat ponad 100 uważnych i skupionych czytelników przesłało sugestie i poprawki. Ich wkład i entuzjazm dla tego projektu był ogromną pomocą.

Szczegółowe informacje na temat charakteru każdego wkładu w rozwój książki można znaleźć w tekście “Think Python”.

Lloyd Hugh Allen, Yvon Boulianne, Fred Bremmer, Jonah Cohen, Michael Conlon, Benoit Girard, Courtney Gleason oraz Katherine Smith, Lee Harr, James Kaylin, David Kershaw, Eddie Lam, Man-Yong Lee, David Mayo, Chris McAloon, Matthew J. Moelter, Simon Dicon Montford, John Ouzts, Kevin Parks, David Pool, Michael Schmitt, Robin Shaw, Paul Sleigh, Craig T. Snydal, Ian Thomas, Keith Verheyden, Peter Winstanley, Chris Wrobel, Moshe Zadka, Christoph Zwerschke, James Mayer, Hayden McAfee, Angel Arnal, Tauhidul Hoque oraz Lex Berezhny, Dr. Michele Alzetta, Andy Mitchell, Kalin Harvey, Christopher P. Smith, David Hutchins, Gregor Lingl, Julie Peters, Florin Oprina, D. J. Webre, Ken, Ivo Wever, Curtis Yanko, Ben Logan, Jason Armstrong, Louis Cordier, Brian Cain, Rob Black, Jean-Philippe Rey z Ecole Centrale Paris, Jason Mader z George Washington University, Jan Gundtofte-Bruun, Abel David oraz Alexis Dinno, Charles Thayer, Roger Sperberg, Sam Bull, Andrew Cheung, C. Corey Capel, Alessandra, Wim Champagne, Douglas Wright, Jared Spindor, Lin Peiheng, Ray Hagtvedt, Torsten Hübsch, Inga Petuhhov, Arne Babenhauserheide, Mark E. Casida, Scott Tyler, Gordon Shephard, Andrew Turner, Adam Hobart, Daryl Hammond oraz Sarah Zimmerman, George Sass, Brian Bingham, Leah Engelbert-Fenton, Joe Funke, Chao-chao Chen, Jeff Paine, Lubos Pintes, Gregg Lind oraz Abigail Heithoff, Max Hailperin, Chotipat Pornavalai, Stanislaw Antol, Eric Pashman, Miguel Azevedo, Jianhua Liu, Nick King, Martin Zuther, Adam Zimmerman, Ratnakar Tiwari, Anurag Goel, Kelli Kratzer, Mark Griffiths, Roydan Ongie, Patryk Wolowiec, Mark Chonofsky, Russell Coleman, Wei Huang, Karen Barber, Nam Nguyen, Stéphane Morin, Fernando Tardio oraz Paul Stoop.

# Dodatek B

# Szczegóły dotyczące praw autorskich



Wolałbym udostępniać książkę na mniej restrykcyjnej licencji CC-BY-SA. Ale niestety istnieje kilka pozbawionych skrupułów organizacji, które szukają i znajdują książki z wolnymi licencjami, a następnie publikują i sprzedają praktycznie niezmienione egzemplarze książek w usłudze druku na żądanie takich jak LuLu czy KDP. KDP (na szczęście) dodało zasady, która daje pierwszeństwo woli rzeczywistego posiadacza praw autorskich przed podmiotami nie posiadającymi praw autorskich, które próbują opublikować wolny utwór objęty licencją. Niestety, istnieje wiele usług druku na żądanie i bardzo niewiele z nich rozważa takie zasady jak przyjęte przez KDP.

Niestety, dodałem element NC (NonCommercial, z ang. użycie niekomercyjne) do licencji tej książki, tak by dać mi możliwość roszczeń regresowych w przypadku, gdy ktoś spróbuje skopiować tę książkę i będzie ją sprzedawać komercyjnie. Niestety, dodanie elementu NC ogranicza wykorzystanie tego materiału w zakresie jakim chciałbym zezwolić. Dodałem więc tę część dokumentu, aby opisać konkretne sytuacje, w których z góry wyrażam zgodę na wykorzystanie materiału zawartego w tej książce w sytuacjach, które niektórzy mogą uznać za komercyjne.

- Jeśli drukujesz ograniczoną liczbę egzemplarzy całości lub części tego podręcznika do wykorzystania podczas zajęć (np. w formie pakietu szkoleniowego), wówczas w tym celu na te materiały otrzymujesz licencję CC-BY.

- Jeśli jesteś nauczycielem na uczelni i tłumaczysz tę książkę na język inny niż angielski i nauczasz za pomocą przetłumaczonej książki, możesz skontaktować się ze mną, a ja udzielę ci licencji CC-BY-SA na te materiały w odniesieniu do publikacji twojego tłumaczenia. W szczególności, będziesz mógł sprzedać przetłumaczoną książkę komercyjnie.

Jeśli zamierzasz przetłumaczyć książkę, możesz skontaktować się ze mną, abyśmy mogli się upewnić, że posiadasz wszystkie związane z nią materiały kursowe, dzięki czemu będziesz mógł je również przetłumaczyć.

Oczywiście, możesz również skontaktować się ze mną i poprosić o zgodę, jeśli powyższe zapisy nie są wystarczające. We wszystkich przypadkach zgoda na ponowne użycie i remiksowanie tego materiału zostanie udzielona pod warunkiem, że pojawi się wyraźna wartość dodana lub korzyść dla studentów lub nauczycieli, która powstanie w wyniku nowej pracy.

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
9 września 2013 r.