Poniższy rozdział pochodzi z książki:

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

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

Rozdział 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 zagadnień analizy danych, po zabawę i pomaganie komuś w rozwiązaniu jakiegoś problemu. Poniższa książka zakłada, że *każdy* powinien wiedzieć, jak się programuje, więc gdy już się dowiesz, jak programować, to zorientujesz się, co chcesz zrobić ze swoją nową umiejętnością.

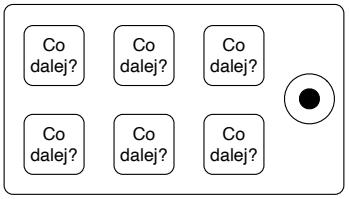
W codziennym życiu jesteśmy otoczeni przez komputery, począwszy od laptopów po smartfony. Możemy myśleć o tych komputerach jak o naszych "osobistych asystentach", którzy w naszym imieniu mogą zająć się wieloma sprawami. Sprzęt we współczesnych komputerach jest zasadniczo zbudowany tak, aby nieustannie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

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

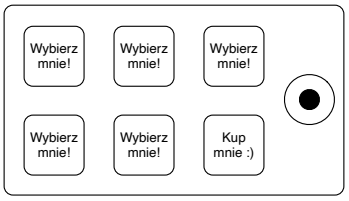
Nasze komputery są szybkie i mają ogromne zasoby pamięci – ten fakt byłby dla nas bardzo pomocny, gdybyśmy tylko znali język do porozumiewania się i wyjaśniania komputerowi, co chcemy, aby dla nas "teraz zrobił". Gdybyśmy znali taki język, to moglibyśmy powiedzieć komputerowi, by wykonał za nas pewne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

Na przykład: spójrz na pierwsze trzy akapity tego rozdziału i wskaż, które słowo było najczęściej użyte oraz ile razy pojawiło się w tekście. W ciągu kilku sekund byłeś w stanie przeczytać i zrozumieć słowa, ale zliczanie ich jest niemalże bolesne, ponieważ nie jest to problem takiego rodzaju, do rozwiązywania którego został zaprojektowany ludzki umysł. W przypadku komputera jest na odwrót – czytanie i rozumienie tekstu z kartki papieru jest trudne do wykonania, ale zliczenie słów i wskazanie, które słowo zostało użyte najczęściej (oraz ile razy), jest bardzo łatwe:

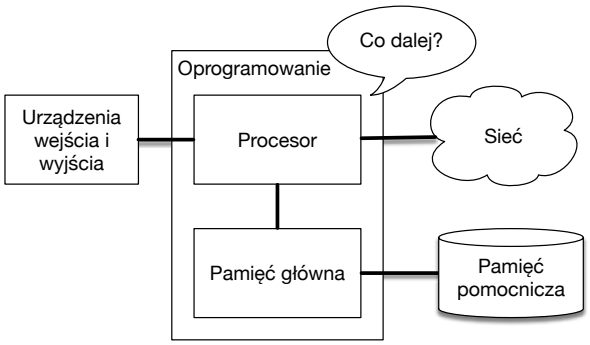
```
$ python3 words.py
Podaj nazwę pliku: words.txt
w 5
```



Rysunek 1.1. Osobisty asystent cyfrowy



Rysunek 1.2. Programiści mówią do Ciebie



Rysunek 1.3. Architektura sprzętowa

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

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 swojemu partnerowi (komputerowi) nieciekawe zadania, pozostawiając sobie więcej czasu na sprawy, do których jesteś przystosowany. Do tego duetu wnosisz kreatywność, intuicję i pomysłowość.

1.1. Kreatywność i motywacja

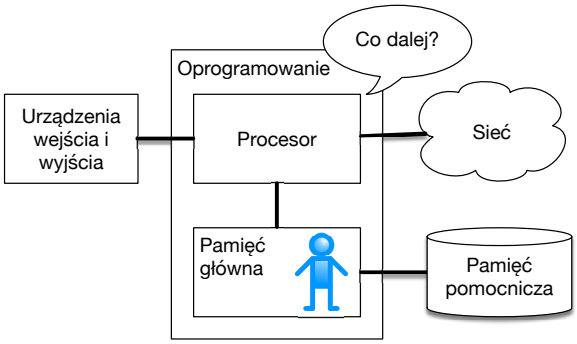
Chociaż ta książka nie jest przeznaczona dla zawodowych programistów, to zawodowe programowanie może być pracą bardzo satysfakcjonującą zarówno pod kątem finansowym, jak i osobistym. Tworzenie przydatnych, eleganckich i sprytnych programów, z których mogą korzystać inni, jest bardzo kreatywnym zajęciem. Twój komputer lub smartfon zwykle zawiera wiele różnych programów dostarczonych przez wiele różnych grup programistów. Aplikacje te ciągle konkurują o Twoją uwagę i zainteresowanie. Robią one wszystko co w ich mocy, aby zaspokoić Twoje potrzeby i zapewnić podczas ich używania najlepsze *user experience* (z ang. doświadczenie użytkownika). W niektórych sytuacjach, gdy wybierasz konkretne oprogramowanie, programiści są bezpośrednio wynagradzani z powodu Twojego wyboru.

Jeśli myślimy o programach jako o twórczym wyniku pracy grup programistów, to zapewne poniższy rysunek będzie przedstawiał bardziej racjonalną wersję naszego smartfona:

Na tę chwilę naszą główną motywacją nie jest zarabianie pieniędzy ani zadowalanie użytkowników końcowych; bardziej będzie nas interesowało produktywne przetwarzanie danych i informacji, które napotkamy w naszym życiu. Na początku będziesz zarówno programistą, jak i końcowym użytkownikiem swoich programów. Gdy zdobędziesz już umiejętności programisty, a programowanie stanie się dla Ciebie bardziej kreatywne, będziesz mógł skierować się w stronę tworzenia programów dla innych osób.

1.2. Architektura sprzętu komputerowego

Zanim zaczniemy uczyć się języka, którym będziemy wydawać polecenia komputerom, aby tworzyły dla nas oprogramowanie, musimy trochę się doszkolić z tego, jak budowane są komputery. Gdybyś rozebrał komputer lub smartfon i wnikliwie się mu przyjrzał, to znalazłbyś takie oto części:



Rysunek 1.4. Gdzie jesteś?

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 pyta "Co dalej?" trzy miliardy razy na sekundę. Będziesz musiał nauczyć się szybkiej komunikacji z procesorem, tak by za nim nadążać.
- *Pamięć główna*[1] 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.
- *Pamięć pomocnicza* jest również używana do przechowywania informacji, ale jest znacznie wolniejsza niż pamięć główna. Zaletą pamięci pomocniczej jest to, że może przechowywać informacje nawet wtedy, gdy komputer nie jest zasilany. Przykładami pamięci pomocniczej są dyski twarde lub pamięć flash (zwykle znajdująca się w pamięciach USB/pendrive i przenośnych odtwarzaczach muzycznych).
- *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 z sieci. 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 znać podstawową terminologię na tyle, abyśmy podczas pisania naszych programów mogli rozmawiać o różnych częściach komputera.

Twoim zadaniem jako programisty jest wykorzystywanie i zarządzanie każdym z tych zasobów po to, by rozwiązać problem i przeanalizować otrzymane dane. Jako programista będziesz głównie "rozmawiać" z procesorem i mówić mu, co ma dalej robić. Czasami powiesz procesorowi, aby użył pamięci głównej, pamięci pomocniczej, sieci lub urządzeń wejścia/wyjścia.

Musisz być osobą, która odpowiada na pytanie procesora "Co dalej?". Zmniejszenie Ciebie do 5 mm wysokości i włożenie do komputera tylko po to, abyś mógł wydawać polecenia trzy miliardy razy na sekundę, to raczej niezbyt komfortowe rozwiązanie. Zamiast tego, w pierwszej kolejności będziesz musiał 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 zamienić Cię w osobę biegłą w sztuce programowania. Na końcu zostaniesz *programistą* – być może niekoniecznie profesjonalnym, ale przynajmniej będziesz posiadał umiejętności spojrzenia na problem analizy danych/informacji oraz opracowania programu do rozwiązania takiego problemu.

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

W pewnym sensie, aby być programistą, potrzebujesz dwóch umiejętności:

- Po pierwsze, musisz znać język programowania (np. 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. Tworzenie historii wymaga pewnej sztuki, ale umiejętność jej pisania poprawia się poprzez pisanie i uzyskiwanie informacji zwrotnych. W programowaniu nasz program jest "historią", a problem, który próbujesz rozwiązać, to "idea" lub "myśl".

Gdy nauczysz się jednego języka programowania (w tym przypadku Pythona), znacznie łatwiej będzie Ci nauczyć się drugiego języka programowania, np. JavaScript lub C++. Nowy język programowania bardzo różni się słownictwem i gramatyką, ale umiejętności rozwiązywania problemów będą takie same we wszystkich językach programowania.

Dość szybko nauczysz się "słownictwa" i "zdań" stosowanych w Pythonie. Napisanie spójnego programu rozwiązującego zupełnie nowy problem zajmie trochę więcej czasu. Uczymy się programowania podobnie jak zwykłego pisania. Zaczynamy czytać i objaśniać programy, potem tworzymy proste programy, a 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 dla Ciebie bardzo przyjemne i kreatywne.

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

1.4. Słowa i zdania

W przeciwieństwie do zwykłych języków stosowanych przez ludzi do codziennej komunikacji, słownictwo Pythona jest w rzeczywistości dość skromne. To "słownictwo" nazywamy "słowami zastrzeżonymi". Są to wyrazy, które mają dla Pythona szczególne znaczenie. Gdy Python widzi te słowa w programie, to mają one dla niego jedno i tylko jedno znaczenie. Później, już podczas pisania programów, stworzysz własne słowa, które dla Ciebie będą miały jakieś specjalne znaczenie – będą to *zmienne*. Będziesz mieć dużą swobodę w wyborze nazw dla swoich zmiennych, za wyjątkiem używania zastrzeżonych słów Pythona.

Gdy trenujemy psa, używamy specjalnych słów, np. "siad", "zostań" i "aport". Kiedy rozmawiasz z psem i nie używasz żadnych specjalnych (zastrzeżonych) słów, to po prostu patrzy na Ciebie z pytającym wyrazem twarzy tak długo, aż nie powiesz zastrzeżonego słowa. Np. 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 specjalnym słowem w języku komunikacji z psami. Natomiast wiele wskazuje na to, że język do komunikacji ludzi z kotami nie ma żadnych zastrzeżonych słów[2].

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
```

I tyle. Dodatkowo, w przeciwieństwie do psa, Python jest już w pełni wyszkolony. Za każdym razem, gdy wypowiesz słowo "try", Python bezbłędnie wykona polecenie "try".

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

W odpowiednim czasie poznamy powyższe zastrzeżone słowa oraz to, kiedy ich używać, ale na razie skupimy się na Pythonowym odpowiedniku słowa "daj głos", które jest używane w komunikacji między człowiekiem a psem. Dobrą rzeczą we wskazywaniu Pythonowi, że ma "dać głos", jest to, że możemy mu nawet wskazać, 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. Napisy używane w funkcjach `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 samym napisie.

1.5. Rozmawianie z Pythonem

Mamy już słowo i proste zdanie, które znamy w Pythonie. Teraz 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 je uruchomić. Jest to zbyt wiele szczegółów jak na ten rozdział, więc *koniecznie* wejdź na stronę https://py4e.pl/install, gdzie umieściłem instrukcje dotyczące konfiguracji i uruchamiania Pythona 3 na systemach Windows i macOS oraz w Linuxowym środowisku chmurowym PythonAnywhere. W każdym wariancie, poza samą instalacją Pythona, zwróć szczególną uwagę na sekcje *Uruchomienie wiersza poleceń*, *Uruchomienie skryptu Pythona* i *Uruchomienie sesji interaktywnej Pythona*, w których jest mowa m.in. o katalogu roboczym, który jest istotny z punktu widzenia uruchamiania skryptów z poziomu wiersza poleceń. Przećwicz opisane tam warianty i przykłady, a następnie wróć do dalszej lektury tej książki.

Zrobione? No to jedziemy dalej!

Niezależnie od wybranego przez Ciebie systemu operacyjnego, w każdym z przypadków w pewnym momencie znajdziesz się w oknie wiersza poleceń lub w terminalu. Następnie wpiszesz `python3`, potwierdzisz klawiszem `<Enter>`, a interpreter Pythona rozpocznie pracę w trybie interaktywnym, pojawiając się i wyglądając mniej więcej tak:

```
Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53)
[MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Znak zachęty >>> jest sposobem, w który interpreter języka Python zadaje Ci pytanie "Co mam teraz zrobić?". Python jest teraz gotowy do rozmowy z Tobą. Teraz musisz się tylko dowiedzieć, jak mówić w języku Python. Będziemy komunikować się z Pythonem wprowadzając komunikaty (instrukcje) *za* wyświetlonym przez Pythona znakiem zachęty >>>.

Powiedzmy na przykład, że nie znałeś nawet najprostszych słów lub zdań w języku Python. Możesz użyć standardowego zwrotu, z którego korzystają astronauci, gdy lądują na odległej planecie i próbują rozmawiać z jej mieszkańcami. Wprowadzamy na klawiaturze nasz komunikat `Przybywam w pokoju, zabierz mnie proszę do swojego przywódcy` i otrzymujemy taki oto efekt:

```
>>> 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 Ci za dobrze. Jeśli szybko czegoś nie wymyślisz, mieszkańcy planety prawdopodobnie zaatakują Cię swoimi włóczniami, nadzieją na rożen, upieką na ogniu i zjedzą na obiad.

Na szczęście na czas podróży zabrałeś kopię tej książki, a następnie otworzyłeś ją na tej stronie i spróbowałeś jeszcze raz, pisząc komunikat `print('Witaj świecie!')`:

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

Wygląda to teraz 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 i Python ponownie pokazał swoje pazury.

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

W pewnym sensie, gdy używasz programu napisanego przez kogoś innego, to rozmowa odbywa się między Tobą a innym programistą, a Python działa jako pośrednik. Python to sposób, dzięki któremu twórcy programów mogą wyrazić, jak ma przebiegać taka rozmowa. W następnych rozdziałach będziesz jednym z tych programistów, którzy używają Pythona do rozmów z użytkownikami Twojego programu.

Zanim opuścimy naszą pierwszą rozmowę z interpreterem Pythona, prawdopodobnie powinieneś poznać właściwy sposób na "pożegnanie się" z mieszkańcami planety Python, skoro ani polski, ani angielski nie działa:

```
>>> 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 wygląda inaczej dla powyższych dwóch niepoprawnych prób zakończenia pracy. Drugi błąd jest inny, ponieważ `if` jest słowem zastrzeżonym, a Python zobaczył to słowo i pomyślał, że próbujemy mu coś powiedzieć, ale składnia zdania była błędna.

Właściwym sposobem "pożegnania się" z Pythonem jest wpisanie za interaktywnym znakiem zachęty >>> instrukcji `quit()`. Zapewne dojście do tego zajęłoby Ci trochę czasu, więc jak widzisz, posiadanie tej książki pod ręką okaże się w najbliższym czasie dość 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, a dla komputerów łatwy do odczytywania i przetwarzania. 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 w nim tylko zera i jedynki (składające się na *kod maszynowy*). Jednakże w porównaniu z Pythonem, składnia języka maszynowego jest jeszcze bardziej złożona i znacznie bardziej skomplikowana. W konsekwencji bardzo niewielu programistów pisze cokolwiek w języku maszynowym. Zamiast tego tworzone są różne translatory umożliwiające programistom pisanie w językach wysokiego poziomu, takich jak Python lub JavaScript, a następnie te translatory konwertują programy na język maszynowy w celu wykonania ich przez procesor.

Ponieważ język maszynowy jest powiązany ze sprzętem komputerowym, nie jest on *przenośny* w przypadku różnych typów sprzętu. Programy napisane w językach wysokiego poziomu można przenosić między różnymi komputerami, używając innego interpretera na nowej maszynie lub ponownie kompilując kod, aby utworzyć wersję 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 Python, a interpreter Pythona natychmiast to przetwarza i jest gotowy do wpisania kolejnej linii w języku Python.

Niektóre wiersze napisane w języku Python mówią interpreterowi Pythona, że chcesz, aby zapamiętał on jakąś wartość na później. Musimy wybrać nazwę dla tej zapamiętywanej wartości, a potem możemy użyć tej umownej nazwy do jej pobrania. Etykiety, których używamy, odnosząc się do przechowywanych danych, nazywamy *zmiennymi*.

```
>>> 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 funkcji `print()`, czy Python faktycznie zapamiętał tę 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. Z kolei *kompilator* najpierw musi otrzymać w pliku cały program, potem uruchamia proces tłumaczenia kodu źródłowego wysokiego poziomu na język maszynowy, a następnie kompilator umieszcza w pliku wynikowy język maszynowy, aby go później wykonać.

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 macOS nie ma przyrostka, który jednoznacznie określałby plik jako wykonywalny.

Gdybyś otworzył 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 już się trochę zastanawiać nad samym interpreterem Pythona. W jakim języku jest on napisany? Czy jest napisany w języku kompilowanym? Gdy wpisujemy w terminalu `python3`, to co się właściwie dzieje?

Interpreter Pythona jest napisany w języku wysokiego poziomu o nazwie "C". Możesz spojrzeć na rzeczywisty kod źródłowy interpretera Pythona, przechodząc na stronę https://www.python.org i sprawdzając jego kod źródłowy. Tak więc Python sam w sobie jest programem i jest skompilowany do kodu maszynowego. Gdy zainstalowałeś na swoim komputerze Pythona (lub zainstalował go dostawca sprzętu), to w rzeczywistości skopiowałeś do swojego systemu kopię programu Python przetłumaczonego na język maszynowy. W systemie Windows wykonywalny kod maszynowy samego Pythona może znajdować się w pliku o następującej ścieżce dostępu:

```
C:\Python38\python.exe
```

Powyższe informacje to wiedza głębsza od tej, która wystarcza do zostania programistą Pythona. Niemniej, czasami warto już na początku odpowiedzieć sobie na intrygujące pytania.

1.7. Pisanie programu

Wpisywanie poleceń do interpretera Pythona to świetny sposób na eksperymentowanie z jego funkcjami, 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`.

Utwórz w katalogu roboczym plik o nazwie `hello.py`, który zawiera jedną linię:

```
print('Witaj świecie!')
```

Wyświetlmy w terminalu zawartość pliku[3]:

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

Symbol `$` to znak zachęty systemu operacyjnego, a `cat hello.py` wyświetla nam zawartość pliku `hello.py`, który składa się z jednej linii[4].

Aby uruchomić skrypt, musimy podać interpreterowi Pythona nazwę pliku:

```
$ python3 hello.py
Witaj świecie!
```

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 na końcu pliku programu instrukcji `quit()`. Gdy Python wczytuje kod źródłowy z pliku, to wie, że ma zakończyć pracę, gdy dojdzie do jego końca.

[3] W przypadku systemu Windows i wiersza poleceń `cmd.exe` wpisalibyśmy `type hello.py`; w zależności od ustawień systemu polecenie to mogłoby wyświetlić nam, zamiast polskich liter, tzw. krzaki, więc konieczna byłaby również uprzednia zmiana aktywnej strony kodowej wiersza poleceń na UTF-8 poprzez wydanie polecenia `chcp 65001`.

[4] W przypadku systemu Windows i wiersza poleceń `cmd.exe` mielibyśmy odpowiednio `>` i `type hello.py`.

1.8. Czym jest program?

W najbardziej podstawowej formie, definicja *programu* to sekwencja instrukcji Pythona, która 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 napisany w języku Python.

Najłatwiej jest zrozumieć pojęcie programu poprzez myślenie o problemie, dla którego można stworzyć rozwiązujący go program, a następnie patrząc właśnie na ten program.

Załóżmy, że prowadzisz badania w dziedzinie socjologii obliczeniowej, analizujesz posty umieszczane na Facebooku i interesuje Cię najczęściej używane słowo w danej serii postów. Możesz wydrukować strumień postów z Facebooka i przejrzeć ręcznie tekst w poszukiwaniu najpopularniejszego słowa, ale zajęłoby to dużo czasu i łatwo by było o pomyłkę. Sprytniej byłoby napisać w Pythonie program, który z tym zadaniem poradzi sobie szybko i bezbłędnie, dzięki czemu zamiast marnować weekend na przeglądanie wydruków postów, będziesz mógł spędzić końcówkę tygodnia na robieniu innych fajnych rzeczy.

Dla przykładu, spójrz na poniższy tekst o klaunie i samochodzie. Wskaż, które słowo jest najczęściej używane oraz ile razy wystąpiło w tekście:

```
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, już szybsze byłoby nauczenie się języka Python i napisanie w nim programu do zliczania słów.

Jeszcze lepszą informacją jest to, że wymyśliłem już prosty program do znajdowania w pliku tekstowym najczęściej występującego słowa. 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://py4e.pl/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. Ale teraz jesteś użytkownikiem końcowym, więc po prostu korzystasz z programu i zachwycasz się jego sprytem oraz oszczędnością Twojego wysiłku. Po prostu zapisujesz powyższy kod w pliku o nazwie `words.py` (lub pobierasz kod źródłowy ze strony https://py4e.pl/code3/) i go 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*. 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 pisanych w Pythonie. Dowiemy się o zaawansowanych możliwościach Pythona oraz o tym, jak je łączyć, tak aby tworzyć przydatne programy.

Istnieje kilka schematów pojęciowych niskiego poziomu, których używamy podczas 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 Dane pozyskane 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 Wyniki programu wyświetlone na ekranie, zapisane do pliku lub na przykład przekazane do urządzenia, takiego jak głośnik, w celu odtworzenia muzyki lub wygłoszenia tekstu.

wykonanie sekwencyjne Wykonanie serii instrukcji jedna po drugiej w tej kolejności, w której występują w skrypcie.

wykonanie warunkowe Sprawdzenie, czy zachodzą określone warunki, a następnie wykonanie lub pominięcie sekwencji instrukcji.

wykonanie wielokrotne Wykonanie kilku zestawów instrukcji wielokrotnie, zwykle z pewnymi zmianami.

ponowne użycie Napisanie zestawu instrukcji raz i nadanie im nazwy, a następnie w miarę potrzeb używanie tych instrukcji w całym programie poprzez użycie tej nazwy.

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 żeby wytworzyć coś, co jest przydatne dla użytkowników tego programu.

Pokazany wcześniej program do zliczania słów bezpośrednio używa prawie wszystkich (za wyjątkiem jednego) powyższych schematów pojęciowych.

1.10. Co może pójść nie tak?

Jak widzieliśmy w naszych pierwszych rozmowach z Pythonem, musimy bardzo precyzyjnie komunikować się, gdy piszemy kod. Najmniejsze odchylenie lub błąd spowoduje, że Python przestanie w ogóle 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 wszystkich innych, to właśnie ich zna osobiście i żywi do nich urazę. Z tego powodu Python czyta ich doskonale napisane programy i odrzuca je jako "nieodpowiednie" tylko po to, by ich dręczyć.

```
>>> 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. Interpreter sprawdził, co wpisałeś, i po prostu nie może tego zrozumieć.

Python jest bardzo 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 Błąd składniowy (ang. *syntax error*) to pierwszy błąd, który popełnisz, i zarazem najłatwiejszy do naprawienia. Błąd tego typu oznacza, że naruszyłeś "gramatyczne" zasady Pythona. Python robi wszystko, co w jego mocy, by wskazać wiersz i znak, w którym zauważył, że się pogubił. Jedynym trudnym problemem błędów składniowych jest to, że czasami błąd, który wymaga naprawy, w rzeczywistości występuje w programie znacznie wcześniej niż w miejscu, w którym Python *zauważył*, że się pogubił. 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 wtedy, 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 *chciałeś*, 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 jest na farmie i chodzi za stodołą, ale nie ma tu śladu restauracji. Następnie pytasz: "Skręciłeś w lewo czy w prawo na stacji benzynowej?". A on mówi: "Postępowałem dokładnie według Twoich wskazówek, mam je zapisane. Jest napisane, żeby skręcić w lewo i jechać jeden kilometr do stacji benzynowej". Wtedy odpowiadasz: "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 lub wynik inny od zamierzonego, to rozpoczynamy poszukiwanie przyczyny błędu. *Debugowanie* to proces znajdowania w kodzie przyczyny błędu. W trakcie 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 do momentu, 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 Twój program robi, 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", będący procesem dokonywania przypadkowych zmian tak długo, aż 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 upraszczanie programu tak długo, aż dojdziesz do czegoś, co działa i 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 zanim zaczniesz demontować swój kod, skopiuj go do innego pliku. Następnie możesz z powrotem wklejać po trochu kawałki kodu.

1.12. Podróż do krainy rozwoju

W miarę gdy zagłębiasz się w kolejne części książki, nie bój się, jeśli za pierwszym razem przedstawiane koncepcje nie będą się wydawały tworzyć logicznej całości. Gdy uczyłeś się mówić, to przez kilka pierwszych lat dla nikogo 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 jeszcze 5-6 lat, a potem jeszcze kilka dodatkowych zajęło samodzielne pisanie interesujących i spójnych opowiadań.

Chcemy, abyś nauczył się języka Python znacznie szybciej, więc w następnych kilku rozdziałach uczymy wszystkiego po trochu w tym samym czasie. 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. Książka jest napisana w sposób liniowy i jeśli bierzesz udział w kursie, to będzie on również przebiegał w sposób liniowy, ale nie wahaj się podchodzić do materiału bardzo nieliniowo. Zajrzyj do kolejnych i poprzednich rozdziałów i czytaj je bez stresu. Przeglądając bardziej zaawansowany materiał bez pełnego zrozumienia szczegółów, możesz lepiej zrozumieć pytanie "dlaczego?", które często pojawia się w programowaniu. Przeglądając wcześniejszy materiał, a nawet powtarzając poprzednie ćwiczenia, zdasz sobie sprawę, że faktycznie nauczyłeś się już bardzo wielu rzeczy, nawet jeśli aktualny materiał wydaje się nieco nieprzystępny.

Zwykle gdy uczysz się swojego pierwszego języka programowania, zdarza się kilka wspaniałych chwil typu "aha!", w których możesz oderwać wzrok od dłuta, którym rzeźbisz w skale; możesz odejść na bok i zobaczyć, że rzeczywiście tworzysz piękną rzeźbę.

Jeśli coś wydaje się szczególnie trudne, to zazwyczaj nie ma sensu zarywać całej nocy i skupiać się tylko na tym. 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, i tylko 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 poziomu 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 dla ludzi do czytania i pisania.

kod maszynowy Język najniższego poziomu dla oprogramowania, który jest bezpośrednio uruchamiany 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 tekstu 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, znajdowania 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. Czym jest program?

Ćwiczenie 3. Jaka jest różnica między kompilatorem a interpreterem?

Ćwiczenie 4. Która z poniższych opcji 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 możliwości człowieka:

a. procesor,
b. pamięć główna,
c. pamięć pomocnicza,
d. urządzenie wejścia,
e. urządzenie wyjścia.

Np. "Czym mógłby być procesor w kontekście człowieka?".

Ćwiczenie 9. W jaki sposób naprawisz "Syntax Error" (błąd składniowy)?