MATEMATYKA W PYTHONIE

ALGEBRA, STATYSTYKA, ANALIZA MATEMATYCZNA I INNE DZIEDZINY

AMIT SAHA

Tytuł oryginału: Doing Math with Python: Use Programming to Explore Algebra, Statistics, Calculus, and More!

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-7493-5

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/matpyt
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/matpyt.zip

Printed in Poland.

Spis treści

1

Operacje na liczbach

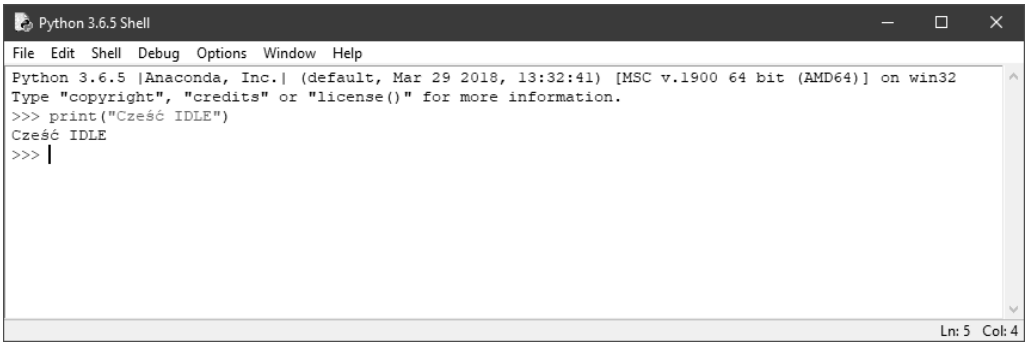
SPRÓBUJMY WYKONAĆ PIERWSZE KROKI NA DRODZE DO STOSOWANIA PYTHONA DO POZNAWANIA ŚWIATA MATEMATYKI I NAUKI. NA RAZIE ZAJMIEMY SIĘ PROSTYMI ZAGADNIENIAMI, TAK BYŚ MÓGŁ NABYĆ WPRAWY w posługiwaniu się samym Pythonem. Zaczniemy od podstawowych operacji matematycznych, a następnie napiszemy proste programy wykonujące działania na liczbach, które pozwolą zrozumieć działanie tych operacji. A zatem zaczynajmy!

Podstawowe operacje matematyczne

Naszym „najlepszym przyjacielem" w tej książce będzie interaktywna powłoka Pythona — *Python Shell*. (Informacje o tym, jak zainstalować Pythona oraz uruchomić powłokę Pythona, można znaleźć w dodatku A). Kiedy to już zrobimy, spróbujemy się przywitać (patrz rysunek 1.1). W tym celu wystarczy wpisać `print ↪("Cześć IDLE")` i nacisnąć klawisz *Enter*. Program IDLE posłucha naszej instrukcji i wyświetli tekst w oknie. Gratuluję — właśnie napisałeś pierwszy program!

Zawsze kiedy pojawi się sekwencja znaków `>>>`, będzie to oznaczało, że program IDLE jest gotowy na wykonanie kolejnych instrukcji.

Python może działać jako wspaniały kalkulator, wykonując różnego rodzaju obliczenia. Wystarczy wpisać wyrażenie matematyczne, a Python je wykona. Kiedy naciśniemy klawisz *Enter*, natychmiast pojawi się wynik.



Rysunek 1.1. Interaktywna powłoka IDLE

Zresztą sam spróbuj! Liczby można dodawać, używając operatora dodawania (+), oraz odejmować, używając operatora odejmowania (-). Poniżej przedstawiłem kilka przykładów:

```
>>> 1 + 2
3
>>> 1 + 3.5
4.5
>>> -1 + 2.5
1.5
>>> 100 - 45
55
>>> -1.1 + 5
3.9
```

Do wykonywania mnożenia służy operator *:

```
>>> 3 * 2
6
>>> 3.5 * 1.5
5.25
>>>
```

Z kolei do dzielenia używamy operatora /:

```
>>> 3 / 2
1.5
>>> 4 / 2
2.0
```

Jak widać, kiedy prosimy o wykonanie operacji dzielenia, Python zwraca liczbę zawierającą część ułamkową. Jeśli zależy nam na wyniku, który będzie mieć postać liczby całkowitej, czyli pozbawionej części ułamkowej, należy użyć operatora dzielenia całkowitego // (ang. *floor division*):

```
>>> 3 // 2
1
```

Operator ten dzieli pierwszy argument przez drugi, a następnie zaokrągla wynik do najbliższej mniejszej liczby całkowitej. Operacja ta staje się ciekawa, kiedy jeden z argumentów jest liczbą ujemną. Oto przykład:

```
>>> -3 // 2
-2
```

Wynikiem operacji dzielenia całkowitego jest liczba całkowita mniejsza od wyniku dzielenia (`-3/2 = -1.5`, a zatem końcowym wynikiem jest `-2`).

Jeśli natomiast interesuje nas reszta z dzielenia, to możemy ją wyznaczyć, używając operatora modulo `%`:

```
>>> 9 % 2
1
```

Potęgi można wyliczać przy użyciu operatora wykładniczego (**). Jego zastosowanie przedstawiają poniższe przykłady:

```
>>> 2 ** 2
4
>>> 2 ** 10
1024
>>> 1 ** 10
1
```

Operatora wykładniczego można także używać do obliczania potęg o wykładniku mniejszym niż 1. Na przykład pierwiastek kwadratowy liczby n można zapisać jako $n^{1/2}$, a pierwiastek trzeciego stopnia jako $n^{1/3}$:

```
>>> 8 ** (1/3)
2.0
```

Jak widać na tym przykładzie, do łączenia operacji matematycznych w celu tworzenia bardziej złożonych wyrażeń można używać nawiasów. Python będzie wykonywał obliczenia zgodnie ze standardową kolejnością działań: najpierw nawiasy, następnie potęgowanie, mnożenie, dzielenie, dodawanie i w ostatniej kolejności dzielenie. Przyjrzyjmy się dwóm zamieszczonym poniżej wyrażeniom: pierwszemu bez nawiasów i drugiemu, w którym nawisy zostały użyte:

```
>>> 5 + 5 * 5
30
```

```
>>> (5 + 5) * 5
50
```

W przypadku pierwszego z nich Python najpierw wylicza mnożenie, `5 * 5`, co daje 25, a następnie dodawanie, `25 + 5`, co daje w wyniku 30. W przypadku drugiego wyrażenia najpierw jest wyliczana wartość w nawiasie (zgodnie z oczekiwaniami): `5 + 5`, co daje 10, i `10 * 5`, co daje 50.

Tak wyglądają absolutne podstawy operowania na liczbach w języku Python. A teraz zobaczmy, jak można przypisywać liczbom nazwy.

Etykiety: przypisywanie nazw liczbom

Kiedy zaczniemy pisać bardziej złożone programy, konieczne będzie przypisywanie liczbom jakichś nazw — po części wynika to z wygody, jednak w głównej mierze jest to po prostu konieczne. Oto kilka prostych przykładów:

```
>>> a = 3 ❶
>>> a + 1
4
>>> a = 5 ❷
>>> a = 1
6
```

W instrukcji oznaczonej cyfrą ❶ przypisujemy liczbie 3 nazwę `a`. Następnie, kiedy poprosimy Pythona o wyliczenie wartości wyrażenia `a + 1`, sprawdzi on, że z nazwą `a` skojarzona jest wartość 3, zatem doda ją do liczby 1 i wyświetli wynik: 4. W przypadku instrukcji oznaczonej cyfrą ❷ zmieniamy wartość `a` na 5, a zmiana zostanie uwzględniona w kolejnej operacji dodawania. Stosowanie nazwy `a` jest wygodne, gdyż wystarczy, że zmienimy wartość, do której odwołuje się ta nazwa, a Python użyje nowej wartości wszędzie tam, gdzie później nazwa `a` zostanie użyta.

Tego rodzaju nazwy określa się mianem *etykiet* (ang. *labels*). W innych językach programowania można się także spotkać z określaniem tego samego rozwiązania słowem *zmienne*. Jednak zważywszy na to, że *zmienna* jest także terminem matematycznym (używanym do określenia czegoś takiego jak x w równaniu $x + 2 = 3$), w tej książce będę używał tego terminu wyłącznie w kontekście równań i wyrażeń matematycznych.

Różne rodzaje liczb

Zapewne zauważyłeś, że podczas prezentowania operacji matematycznych użyłem dwóch rodzajów liczb: liczb bez części dziesiętnej, o których już wiemy, że są to *liczby całkowite*, oraz liczb z częścią dziesiętną, które programiści nazywają

liczbami zmiennoprzecinkowymi. My, ludzie, nie mamy większych problemów z operowaniem na liczbach niezależnie od tego, jak są zapisane: jako liczby całkowite, w formie liczb z częścią dziesiętną, w formie ułamków zwykłych czy też nawet w formie liczb rzymskich. Jednak w niektórych programach, które będziemy pisać w niniejszej książce, wykonywanie operacji będzie mieć sens wyłącznie w razie posługiwania się liczbami określonego typu; dlatego czasami będzie się pojawiała konieczność napisania specjalnego fragmentu kodu, który będzie sprawdzał, czy wpisywane liczby są odpowiedniego typu.

Pyton traktuje liczby całkowite oraz zmiennoprzecinkowe jako wartości dwóch różnych *typów*. Przy użyciu funkcji `type()` można sprawdzić, jakiego typu liczba została wpisana. Oto kilka przykładów:

```
>>> type(3)
<class 'int'>
>>> type(3.5)
<class 'float'>
>>> type(3.0)
<class 'float'>
```

Jak pokazuje ten przykład, Python sklasyfikował liczbę `3` jako liczbę całkowitą (typu `'int'`), natomiast liczbę `3.0` jako liczbę zmiennoprzecinkową (typu `'float'`). Doskonale wiemy, że z matematycznego punktu widzenia liczby 3 i 3,0 są swoimi odpowiednikami, jednak w bardzo wielu przypadkach Python będzie je traktował inaczej, gdyż są one różnych typów.

Niektóre programy, które napiszemy w dalszej części tego rozdziału, będą działały prawidłowo wyłącznie na liczbach całkowitych. Jak już pokazałem na wcześniejszych przykładach, Python nie rozpozna wartości `1.0` lub `4.0` jako liczb całkowitych, dlatego też, jeśli będziemy chcieli, by takie wartości były prawidłowymi danymi wejściowymi dla tych programów, będziemy musieli skonwertować je do postaci liczb całkowitych. Na szczęście Python udostępnia funkcję, która na to pozwala:

```
>>> int(3.8)
3
>>> int(3.0)
3
```

Funkcja `int()` pobiera liczbę zmiennoprzecinkową, pozbywa się jej części dziesiętnej i zwraca uzyskaną liczbę całkowitą. Funkcja `float()` jest podobna, choć wykonuje konwersję w przeciwnym kierunku:

```
>>> float(3)
3.0
```

Funkcja `float()` pobiera liczbę całkowitą, dodaje do niej część dziesiętną i zwraca wartość zmiennoprzecinkową.

Operacje na ułamkach

Python potrafi także wykonywać operacje na ułamkach, jednak do tego celu będziemy potrzebowali modułu `fractions`. *Moduł* można sobie wyobrazić jako napisany przez kogoś innego program, którego możemy używać we własnych programach. Moduł może zawierać klasy, funkcje, a nawet definicje etykiet. Moduł może należeć do standardowej biblioteki Pythona bądź być udostępniany niezależnie z innych lokalizacji. W tym drugim przypadku przed zastosowaniem modułu konieczne będzie jego zainstalowanie.

Moduł `fractions` należy do standardowej biblioteki Pythona, co oznacza, że już jest zainstalowany. Definiuje on klasę `Fraction`, która jest właśnie tym, czego będziemy używać do stosowania w naszych programach ułamków. Nim będziemy mogli skorzystać z tej klasy, musimy ją *zaimportować* — importowanie to sposób, by powiedzieć Pythonowi, że chcemy używać klasy pochodzącej z konkretnego modułu. Przeanalizujmy prosty przykład: stworzymy w nim nową etykietę, `f`, która będzie się odwoływać do ułamka 3/4:

```
>>> from fractions import Fraction ❶
>>> f = Fraction(3, 4) ❷
>>> f ❸
Fraction(3, 4)
```

W pierwszej kolejności importujemy moduł `fractions` (❶). Następnie tworzymy obiekt klasy `Fraction`, przekazując do niego licznik i mianownik ułamka, który planujemy utworzyć (❷). W ten sposób tworzymy obiekt reprezentujący ułamek 3/4. Kiedy spróbujemy wyświetlić obiekt (❸), Python przedstawi ułamek w formie `Fraction(`*`licznik`*`, `*`mianownik`*`)`.

Tak tworzone ułamki można stosować we wszystkich podstawowych operacjach matematycznych, w tym także w operacjach porównywania. Można także, w ramach jednego wyrażenia matematycznego, łączyć ułamki, liczby całkowite i zmiennoprzecinkowe:

```
>>> Fraction(3, 4) + 1 + 1.5
3.25
```

Jeśli w wyrażeniu została użyta jakaś wartość zmiennoprzecinkowa, to wynik także będzie mieć postać wyrażenia zmiennoprzecinkowego.

Z drugiej strony, jeśli w wyrażeniu zostaną użyte wyłącznie liczby całkowite i ułamki reprezentowane przez obiekty `Fraction`, to wynik także będzie mieć postać ułamka i to nawet jeśli jego mianownik będzie mieć wartość 1.

```
>>> Fraction(3, 4) + 1 + Fraction(1/4)
Fraction(2, 1)
```

Teraz już znasz podstawy operowania na ułamkach w Pythonie. Przejdźmy zatem do zupełnie innego rodzaju liczb.

Liczby zespolone

Liczby, którymi posługiwaliśmy się do tej pory, są nazywane *liczbami rzeczywistymi*. Python pozwala także na posługiwanie się *liczbami zespolonymi*, których część urojona jest oznaczana poprzez dodanie litery *j* lub *J* (co jest niezgodne z konwencją matematyczną, w której część urojona liczb zespolonych jest oznaczana literą *i*). Na przykład liczba zespolona $2 + 3i$ w języku Python zostanie zapisana jako $2 + 3j$:

```
>>> a = 2 + 3j
>>> type(a)
<class 'complex'>
```

Jak widać, kiedy przekażemy liczbę zespoloną jako argument wywołania funkcji `type()`, Python poinformuje nas, że jest to obiekt typu `complex`.

Liczby zespolone można także definiować przy użyciu funkcji `complex()`:

```
>>> a = complex(2, 3)
>>> a
(2+3j)
```

W tym przypadku argumenty wywołania funkcji `complex()` określają odpowiednio wartości części rzeczywistej i urojonej tworzonej liczby zespolonej. Funkcja ta zwraca liczbę zespoloną.

Liczby zespolone można dodawać i odejmować dokładnie tak samo jak liczby rzeczywiste:

```
>>> b = 3 + 3j
>>> a + b
(5+6j)
>>> a - b
(-1+0j)
```

Tak samo wykonywane są operacje mnożenia i dzielenia:

```
>>> a * b
(-3+15j)
>>> a / b
(0.8333333333333334+0.16666666666666666j)
```

Na liczbach zespolonych nie można natomiast wykonywać operacji wyznaczania reszty z dzielenia (%) oraz dzielenia całkowitego (//).

Część rzeczywistą oraz urojoną liczb zespolonych można pobierać, używając odpowiednio atrybutów `real` oraz `imag`, jak pokazałem na poniższym przykładzie:

```
>>> z = 2 + 3j
>>> z.real
2.0
>>> z.imag
3.0
```

Sprzężenie liczby zespolonej to liczba zespolona mająca taką samą część rzeczywistą i część urojoną o tej samej wielkości, lecz przeciwnym znaku. W Pythonie sprzężenie liczby zespolonej można wyznaczać przy użyciu metody `conjugate()`:

```
>>> z.conjugate()
(2-3j)
```

Obie części liczb zespolonych, rzeczywista i urojona, są liczbami rzeczywistymi. Korzystając z części rzeczywistej i urojonej, można wyliczyć wielkość danej liczby zespolonej. Służy do tego wzór $\sqrt{x^2 + y^2}$, w którym x i y reprezentują odpowiednio część rzeczywistą i urojoną liczby zespolonej. W Pythonie wyrażenie obliczające wielkość liczby zespolonej można zapisać w następujący sposób:

```
>>> (z.real ** 2 + z.imag ** 2) ** 0.5
3.605551275463989
```

Nieco prostszym sposobem wyznaczania wielkości liczby zespolonej jest użycie funkcji `abs()`. W przypadku przekazania argumentu będącego liczbą rzeczywistą funkcja ta zwraca jego wartość bezwzględną. Na przykład wywołania `abs(5)` oraz `abs(-5)` zwrócą wartość `5`. Jednak w przypadku przekazania liczby zespolonej funkcja `abs()` zwraca jej wielkość:

```
>>> abs(z)
3.605551275463989
```

Moduł `cmath` (jego nazwa stanowi połączenie angielskich słów *complex math* — matematyka na liczbach zespolonych) wchodzący w skład standardowej biblioteki Pythona udostępnia wiele wyspecjalizowanych funkcji operujących na liczbach zespolonych.

Pobieranie danych wejściowych od użytkownika

Kiedy zaczniemy pisać programy, bardzo przyda się nam wygodny i prosty sposób pobierania danych wejściowych wpisywanych przez użytkownika. Takie możliwości zapewnia funkcja `input()`. Dzięki niej można pisać programy, które proszą użytkownika o wpisanie liczby, następnie wykonują na niej jakieś operacje i wyświetlają ich wynik. Zobaczmy, jak wygląda zastosowanie tej funkcji w praktyce:

```
>>> a = input() ❶
1 ❷
```

W wierszu oznaczonym cyfrą ❶ wywołujemy funkcję `input()`, która czeka, aż użytkownik coś wpisze (❷) i naciśnie klawisz *Enter*. Wpisane dane zostają skojarzone z etykietą `a`:

```
>>> a
'1'   ❸
```

Warto zwrócić uwagę na apostrofy widoczne wokół cyfry 1 w wierszu ❸. Funkcja `input()` zwraca wpisane dane w formie *łańcucha*. W języku Python łańcuchem nazywamy dowolny ciąg znaków zapisanych pomiędzy dwoma apostrofami. Python pozwala na zapisywanie łańcuchów nie tylko pomiędzy znakami apostrofu, lecz także cudzysłowu:

```
>>> s1 = 'oto łańcuch'
>>> s2 = "oto łańcuch"
```

W tym przykładzie obie etykiety, `s1` i `s2`, odwołują się do tego samego łańcucha.

Nawet jeśli wszystkimi znakami w łańcuchu będą cyfry, to Python nie potraktuje go jako liczby, jeśli nie pozbędziemy się znaków apostrofu lub cudzysłowu. Dlatego nim będziemy mogli wykonać jakiekolwiek operacje matematyczne na danych wpisanych przez użytkownika, będziemy musieli skonwertować je do postaci liczb odpowiedniego typu. Do konwersji łańcuchów na liczbę całkowitą lub zmiennoprzecinkową służą odpowiednio funkcje `int()` oraz `float()`; przykłady ich zastosowania przedstawiłem poniżej:

```
>>> a = '1'
>>> int(a) + 1
2
>>> float(a) + 1
2.0
```

To są dokładnie te same funkcje `int()` i `float()`, które poznaliśmy już wcześniej, jednak tym razem zamiast konwertować liczby jednego typu na inny, używamy tych funkcji do skonwertowania łańcucha (`'1'`) na liczbę (odpowiednio: `2` i `2.0`). Trzeba jednak zauważyć, że funkcja `int()` nie może skonwertować na liczbę łańcucha zawierającego tekstową reprezentację liczby z częścią dziesiętną. Jeśli przekażemy do niej łańcuch zawierający liczbę zmiennoprzecinkową (taki jak `'2.5'` lub `'2.0'`), próba wykonania funkcji spowoduje wyświetlenie komunikatu o błędzie:

```
>>> int('2.0')
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    int('2.0')
ValueError: invalid literal for int() with base 10: '2.0'
```

To przykład wystąpienia tak zwanego *wyjątku*, który stanowi sposób, w jaki Python informuje, że nie jest w stanie kontynuować działania programu ze względu na wystąpienie błędu. W tym konkretnym przypadku wystąpił wyjątek typu `ValueError`. (Krótkie przypomnienie informacji o wyjątkach i nie tylko można znaleźć w dodatku B).

Jeśli spróbujemy skonwertować łańcuch, na przykład pobrany przy użyciu funkcji `input()`, zawierający tekstową reprezentację ułamka, takiego jak 3/4, to Python także nie będzie potrafił skonwertować go ani na liczbę całkowitą, ani zmiennoprzecinkową. Także w takim przypadku zostanie zgłoszony wyjątek `ValueError`.

```
>>> a = float(input())
3/4
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    a = float(input())
ValueError: could not convert string to float: '3/4'
```

Warto zastanowić się nad wykonywaniem konwersji wewnątrz bloku `try...`
`↪except`, co pozwoli na *obsługę* takich wyjątków i ostrzeganie użytkownika, że program napotkał nieprawidłowe dane wejściowe. Bloki `try...except` zostały opisane w następnym punkcie rozdziału.

Obsługa wyjątków i nieprawidłowych danych wejściowych

Jeśli jeszcze nie znasz konstrukcji `try...except`, to powinieneś wiedzieć, że podstawowa idea jej działania polega na tym, że jeśli wykonujemy grupę instrukcji wewnątrz bloku `try...except` i jeśli zostanie w nich zgłoszony jakiś błąd, to działanie programu nie zostanie przerwane i żadne informacje o błędzie nie zostaną wyświetlone. Zamiast tego realizacja kodu zostanie przeniesiona do bloku `except`, gdzie będziemy mieli okazję wykonać odpowiednie operacje, na przykład wyświetlić stosowny komunikat informacyjny lub spróbować innego rozwiązania.

Oto w jaki sposób można wykonać konwersję wewnątrz bloku `try...except` i wyświetlić przydatny komentarz w razie wpisania nieprawidłowych danych:

```
>>> try:
        a = float(input('Proszę wpisać liczbę: '))
except ValueError:
        print('Wpisana liczba jest nieprawidłowa!')
```

Należy zwrócić uwagę, że konieczne jest podanie typu wyjątku, który chcemy obsługiwać. W powyższym przykładzie jest to wyjątek `ValueError`, dlatego też został użyty zapis `except ValueError`.

Jeśli użytkownik wpisze teraz nieprawidłową wartość, taką jak 3/4, to zostanie wyświetlony pomocny komunikat o błędzie, taki jak ten pokazany w wierszu ❶:

```
Proszę wpisać liczbę:  3/4
Wpisana liczba jest nieprawidłowa! ❶
```

Jak widać na ostatnim przykładzie, funkcja `input()` także pozwala na wyświetlenie komunikatu; można go użyć, aby określić, jakiego rodzaju informacji oczekujemy. Oto przykład:

```
>>> a = input('Wpisz liczbę całkowitą: ')
```

W efekcie użytkownik zobaczy wskazówkę zalecającą wpisanie liczby całkowitej:

```
Wpisz liczbę całkowitą:  1
```

W wielu programach zamieszczonych w tej książce będziemy prosić użytkownika o wpisywanie liczb, dlatego należy pamiętać, by przed rozpoczęciem wykonywania na nich operacji matematycznych zadbać o ich odpowiednie skonwertowanie. Operacje wprowadzenia i konwersji danych można połączyć i zapisać w formie jednej instrukcji, takiej jak poniższa:

```
>>> a = int(input())
1
>>> a + 1
2
```

Takie rozwiązanie zadziała idealnie, jeśli użytkownik wpisze liczbę całkowitą. Ale jak już mieliśmy okazję się przekonać, jeśli użytkownik wpisze liczbę zmiennoprzecinkową (nawet stanowiącą odpowiednik liczby całkowitej, taką jak 1.0), to wykonanie takiego wywołania spowoduje zgłoszenie błędu:

```
>>> a = int(input())
1.0
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    a = int(input())
ValueError: invalid literal for int() with base 10: '1.0'
```

Aby uniknąć tego błędu, moglibyśmy przechwytywać wyjątek `ValueError` w taki sam sposób, jak robiliśmy to wcześniej w celu obsługi ułamków. W ten sposób program przechwytywałby próby wpisywania liczb zmiennoprzecinkowych, które w programie operującym na liczbach całkowitych nie nadają się do użytku. Niemniej jednak takie rozwiązanie spowodowałoby odrzucenie liczb takich jak `1.0` lub `2.0`, które Python *traktuje* jako wartości zmiennoprzecinkowe, choć są odpowiednikami liczb całkowitych i działałyby równie dobrze jak liczby wpisane jako wartości całkowite.

Aby ominąć ten problem, użyjemy metody `is_integer()`, która pozwoli nam odrzucić wszystkie liczby mające jakieś liczby znaczące po przecinku dziesiętnym. (Ta metoda jest dostępna wyłącznie dla liczb typu `float`; nie można jej wywoływać na rzecz liczb, które od razu zostały wpisane jako liczby całkowite).

Oto przykład użycia tej metody:

```
>>> 1.1.is_integer()
False
```

W tym przykładzie używamy metody `is_integer()`, by sprawdzić, czy `1.1` jest liczbą całkowitą. Wywołanie zwróciło wartość `False`, gdyż `1.1` jest wartością zmiennoprzecinkową. Z drugiej strony wywołanie tej metody na rzecz liczby `1.0`, która także jest liczbą zmiennoprzecinkową, zwróci wartość `True`:

```
>>> 1.0.is_integer()
True
```

Możemy zatem zastosować metodę `is_integer()`, by odrzucać wartości, które nie są liczbami całkowitymi, a jednocześnie zagwarantować, że wartości takie jak `1.0`, wpisane jako zmiennoprzecinkowe, lecz stanowiące odpowiedniki liczb całkowitych, będą używane. Już niebawem pokażę, jak można użyć tej metody w większym programie.

Wpisywanie ułamków i liczb zespolonych

Klasa `Fraction`, którą poznaliśmy we wcześniejszej części rozdziału, zapewnia możliwość konwersji na obiekty `Fraction` łańcuchów takich jak `'3/4'`. Poniższy przykład pokazuje, w jaki sposób można pozwolić użytkownikowi na wpisanie ułamka:

```
>>> a = Fraction(input('Wpisz ułamek: '))
Wpisz ułamek: 3/4
>>> a
Fraction(3, 4)
```

Zobaczmy, co się stanie po wpisaniu ułamka takiego jak `3/0`:

```
>>> a = Fraction(input('Wpisz ułamek: '))
Wpisz ułamek: 3/0
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    a = Fraction(input('Wpisz ułamek: '))
  File "c:\DEVEL\Anaconda3\lib\fractions.py", line 178, in __new__
    raise ZeroDivisionError('Fraction(%s, 0)' % numerator)
ZeroDivisionError: Fraction(3, 0)
```

Wyjątek `ZeroDivisionError` informuje nas (o czym już zresztą wiemy), że ułamki z zerem w mianowniku są niedozwolone. Jeśli planujemy zapewnić użytkownikom możliwość wpisywania ułamków jako danych używanych przez program, to zawsze należy przychwytywać te wyjątki. Oto w jaki sposób można to zrobić:

```
>>> try:
        a = Fraction(input('Wpisz ułamek: '))
except ZeroDivisionError:
        print('Nieprawidłowy ułamek!')

Wpisz ułamek:  3/0
Nieprawidłowy ułamek!
```

Teraz za każdym razem, gdy użytkownik programu wpisze ułamek z 0 w mianowniku, program wyświetli komunikat `Nieprawidłowy ułamek!`.

Analogicznie funkcja `complex()` pozwala konwertować łańcuchy takie jak `'2+3j'` na liczby zespolone:

```
>>> z = complex(input('Wpisz liczbę zespoloną: '))
Wpisz liczbę zespoloną:  2+3j
>>> z
(2+3j)
```

Jeśli wpiszemy łańcuch taki jak `'2 + 3j'`, ze znakami odstępu, to zostanie zgłoszony wyjątek `ValueError`:

```
>>> z = complex(input('Wpisz liczbę zespoloną: '))
Wpisz liczbę zespoloną: 2 + 3j
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    z = complex(input('Wpisz liczbę zespoloną: '))
ValueError: complex() arg is a malformed string
```

Podczas konwertowania łańcuchów na liczby zespolone warto przechwytywać wyjątki `ValueException`, tak jak robiliśmy podczas konwertowania liczb innych typów.

Pisanie programów wykonujących obliczenia matematyczne

Skoro poznaliśmy już podstawowe pojęcia i możliwości, możemy połączyć je z instrukcjami warunkowymi oraz pętlami, by pisać w Pythonie programy, które będą nieco bardziej zaawansowane i przydatne.

Obliczanie dzielników liczb całkowitych

Kiedy niezerowa liczba całkowita `a` dzieli inną liczbę całkowitą `b`, dając resztę z dzielenia `0`, to mówimy, że `a` jest *dzielnikiem* `b`. Na przykład liczba `2` jest dzielnikiem wszystkich liczb parzystych. Można napisać funkcję, taką jak ta zamieszczona poniżej, która będzie sprawdzać, czy jedna niezerowa liczba całkowita `a` jest dzielnikiem innej liczby całkowitej `b`:

```
>>> def is_factor(a, b):
        if b % a == 0:
            return True
        else:
            return False
```

Jak widać, w kodzie funkcji użyliśmy operatora modulo `%`, przedstawionego wcześniej w tym rozdziale, by wyliczyć resztę z dzielenia. Jeśli kiedykolwiek zastanawialiśmy się, czy 4 jest dzielnikiem liczby 1024, to funkcja `is_factor()` może nam na to pytanie odpowiedzieć:

```
>>> is_factor(4, 1024)
True
```

Zastanówmy się, w jaki sposób można wyznaczyć wszystkie dzielniki dowolnej dodatniej liczby całkowitej n. Otóż dla każdej liczby z zakresu od 1 do n trzeba podzielić n przez daną liczbę i sprawdzić resztę z dzielenia. Jeśli reszta z dzielenia będzie mieć wartość 0, będzie to oznaczać, że dana liczba jest dzielnikiem liczby n. Zastosujemy funkcję `range()`, żeby napisać program, który sprawdzi wszystkie liczby z zakresu od 1 do n.

Jednak zanim napiszemy kompletny program, przyjrzyjmy się, jak działa funkcja `range()`. Poniższy przykład przedstawia typowy sposób jej zastosowania:

```
>>> for i in range(1, 4):
        print(i)


1
2
3
```

W tym przykładzie użyliśmy pętli `for` i przekazaliśmy do funkcji `range()` dwa argumenty. Funkcja ta zaczyna od pierwszego z przekazanych argumentów (*wartości początkowej*) i zwraca wszystkie liczby całkowite, kończąc na liczbie bezpośrednio *poprzedzającej* tę podaną jako drugi argument (*wartość końcowa*). W tym przykładzie kazaliśmy Pythonowi wyświetlić wszystkie wartości z zakresu od 1 do 4. Koniecznie trzeba zwrócić uwagę, że oznacza to, że wartość 4 nie zostanie wyświetlona — ostatnią wyświetloną wartością będzie liczba bezpośrednio poprzedzająca wartość końcową (czyli 3). Trzeba także zauważyć, że argumentami funkcji `range()` mogą być jedynie liczby całkowite.

Funkcji `range()` można także używać bez podawania wartości początkowej; w takim przypadku przyjmuje ona wartość domyślną `0`. Oto przykład:

```
>>> for i in range(5):
        print(i)


0
1
2
3
4
```

Różnica pomiędzy dwiema kolejnymi wartościami zwracanymi przez funkcję `range()` jest określana jako *krok*. Domyślnie wartością tego kroku jest 1. Aby określić wartość tego kroku, należy przekazać trzeci argument w wywołaniu funkcji `range()` (w tym przypadku podanie wartości początkowej jest *konieczne*). Na przykład poniższy fragment kodu wyświetla wszystkie liczby nieparzyste mniejsze od 10:

```
>>> for i in range(1, 10, 2):
        print(i)


1
3
5
7
9
```

No dobrze, skoro już wiemy, jak działa funkcja `range()`, to jesteśmy gotowi, by napisać program do wyznaczania dzielników liczby całkowitej. Ponieważ będzie

to dość długi program, dlatego zamiast pisać go w interaktywnym wierszu poleceń programu IDLE, użyjemy do tego edytora. Nowe okno edytora można otworzyć w programie IDLE, wybierając z menu opcję *File/New File*. Warto zwrócić uwagę, że na początku kodu został umieszczony komentarz, który rozpoczyna się i kończy trzema znakami apostrofu (`'`). Umieszczony w nim tekst nie zostanie wykonany przez Pythona jako fragment programu — jest to zwyczajny komentarz przeznaczony dla nas, ludzi.

```
'''
Program znajduje dzielniki podanej liczby całkowitej.
'''

def factors(b):
    for i in range(1, b+1): ❶
        if b % i == 0:
            print(i)

if __name__ == '__main__':

    b = input('Proszę wpisać liczbę: ')
    b = float(b)

    if b > 0 and b.is_integer(): ❷
        factors(int(b))
    else:
        print('Proszę wpisać liczbę całkowitą większą od zera.')
```

Funkcja `factors()` definiuje pętlę `for`, której zawartość jest wykonywana jeden raz dla każdej liczby z zakresu od 1 do wpisanej liczby całkowitej; zakres ten jest tworzony przy użyciu funkcji `range()` (❶). W tym przypadku chcemy, by liczba wpisana przez użytkownika (`b`) także była uwzględniona w sprawdzanym zakresie, dlatego wartość końcowa przekazana w wywołaniu `range()` ma postać `b+1`. Dla każdej z liczb reprezentowanych przez `i` funkcja sprawdza, czy dana liczba dzieli liczbę wejściową bez reszty, a jeśli okaże się, że reszta z dzielenia faktycznie wynosi 0, to analizowana liczba zostaje wyświetlona jako dzielnik.

Po uruchomieniu programu (w tym celu należy wybrać z menu opcję *Run/Run Module*) użytkownik zostanie poproszony o wpisanie liczby. Jeśli zostanie wpisana liczba całkowita, program wyświetli jej dzielniki. Oto przykład:

```
Proszę wpisać liczbę: 25
1
5
25
```

W razie wpisania liczby mniejszej od zera lub wartości, która nie jest liczbą całkowitą, program wyświetla komunikat o błędzie z prośbą o wpisanie liczby całkowitej większej od zera:

```
Proszę wpisać liczbę: 15.5
Proszę wpisać liczbę większą od zera.
```

Ten przykład pokazuje, jak dzięki sprawdzaniu danych wpisywanych w programie można sprawić, by programy były bardziej przyjazne dla użytkownika. Ponieważ nasz program potrafi znajdować jedynie dzielniki dodatnich liczb całkowitych, sprawdzamy, czy wpisana przez użytkownika wartość jest większa od zera i czy jest liczbą całkowitą, używając do tego funkcji `is_integer()` (wiersz ❷). Sprawdzenie obu tych warunków pozwala nam zagwarantować, że dane wejściowe będą poprawne. Jeśli wpisana liczba nie będzie dodatnią liczbą całkowitą, to zamiast niezrozumiałych informacji o wyjątku program wyświetli przyjazne wskazówki.

Generowanie tabliczki mnożenia

Rozważmy trzy liczby: a, b i n, gdzie n jest liczbą całkowitą, spełniające zależność:

$$a \cdot n = b$$

Można powiedzieć, że b jest n-tą *wielokrotnością* liczby a. Na przykład 4 jest drugą wielokrotnością liczby 2, a 1024 jest 512. wielokrotnością liczby 2.

Tabliczka mnożenia dla określonej liczby wyświetla listę jej wszystkich wielokrotności. Na przykład tabliczka mnożenia liczby 2 wygląda jak na poniższym przykładzie (pokazałem na nim jedynie trzy pierwsze wielokrotności):

$$2 \cdot 1 = 2$$
$$2 \cdot 2 = 4$$
$$2 \cdot 3 = 6$$

Nasz kolejny program będzie generował tabliczkę mnożenia liczby podanej przez użytkownika, zawierającą jej dziesięć pierwszych wielokrotności. W programie użyjemy metody `format()` oraz funkcji `print()`, dzięki którym zapewnimy, że generowane przez niego wyniki będą ładniejsze i bardziej czytelne. Na wypadek gdybyś nigdy wcześniej nie spotkał się z funkcją `format()`, poniżej pokrótce opiszę jej działanie.

Metoda `format()` pozwala przekazywać etykiety i zapewnia, że skojarzone z nimi wartości zostaną wyświetlone w ładnym, czytelnym łańcuchu, z wykorzystaniem określonych dodatkowych sposobów formatowania. Na przykład, gdybym dysponował nazwami wszystkich owoców, które kupiłem w sklepie, i gdyby każda z nich została skojarzona z odrębną etykietą, a ja chciałbym wyświetlić je wszystkie w jednym, logicznym zdaniu, to za pomocą metody `format()` mógłbym to zrobić w następujący sposób:

```
>>> item1 = 'jabłka'
>>> item2 = 'banany'
>>> item3 = 'winogrona'
>>> print('W warzywniaku kupiłem {0}, {1} i {2}.'.format(item1, item2, item3))
W warzywniaku kupiłem jabłka, banany i winogrona.
```

W tym przykładzie w pierwszej kolejności utworzyliśmy trzy etykiety (`item1`, `item2`, `item3`), z których każda została skojarzona z innym łańcuchem (`'jabłka'`, `'banany'`, `'winogrona'`). Następnie w wywołaniu funkcji `print()` przekazaliśmy łańcuch zawierający specjalne zamienniki, mające postać cyfr zapisanych w nawiasach klamrowych: `{0}`, `{1}` i `{2}`. Ten łańcuch został użyty do wywołania metody `format()`, do której zostały przekazane trzy utworzone wcześniej etykiety. To wywołanie nakazuje Pythonowi zastąpić umieszczone w łańcuchu zamienniki wartościami etykiet w kolejności, w jakiej zostały podane. A zatem w wyświetlonym tekście symbol `{0}` zostanie zastąpiony łańcuchem skojarzonym z pierwszą etykietą, symbol `{1}` łańcuchem skojarzonym z drugą etykietą i tak dalej.

Nie ma konieczności kojarzenia wartości, które chcemy wyświetlać, z etykietami — wartości można podawać bezpośrednio w wywołaniu metody `format()`, jak pokazałem w kolejnym przykładzie:

```
>>> print('Liczba 1: {0}, liczba 2: {1} '.format(1, 3.578))
Liczba 1: 1, liczba 2: 3.578
```

Trzeba zwrócić uwagę, że liczba zamienników w łańcuchu oraz liczba etykiet lub wartości w wywołaniu metody `format()` muszą być równe.

Skoro już wiemy, jak działa metoda `format()`, możemy przyjrzeć się programowi generującemu tabliczkę mnożenia:

```
'''
Generator tabliczki mnożenia
'''
def multi_table(a):

    for i in range(1, 11): ❶
        print('{0} x {1} = {2}'.format(a, i, a*i))

if __name__ == '__main__':
    a = input('Wpisz liczbę: ')
    multi_table(float(a))
```

Główne możliwości programu zostały zaimplementowane w formie funkcji `multi_table()`. Funkcja ta wymaga przekazania liczby (`a`), dla której należy wyświetlić tabliczkę mnożenia. Ponieważ tabliczka ta ma zawierać pierwszych dziesięć wielokrotności, używamy pętli `for` (❶), która dla każdej liczby z zakresu od 1 do 11 wyliczy iloczyn danej liczby z liczbą przekazaną jako argument wywołania (`a`), a następnie wyświetli odpowiednio przygotowany łańcuch.

Po uruchomieniu programu użytkownik zostanie poproszony o podanie liczby, a następnie program wyświetli dla niej tabliczkę mnożenia:

```
Wpisz liczbę:  5
5.0 x 1 = 5.0
5.0 x 2 = 10.0
5.0 x 3 = 15.0
```

```
5.0 x 4 = 20.0
5.0 x 5 = 25.0
5.0 x 6 = 30.0
5.0 x 7 = 35.0
5.0 x 8 = 40.0
5.0 x 9 = 45.0
5.0 x 10 = 50.0
```

Prawda, że tabliczka wygląda bardzo porządnie i schludnie? To dlatego, że do jej wygenerowania użyliśmy metody `format()`, która pozwoliła generować kolejne wiersze tabliczki na podstawie jednolitego, czytelnego szablonu.

Jednak metoda `format()` pozwala na uzyskanie jeszcze większej kontroli nad postacią generowanych łańcuchów. Na przykład, jeśli chcemy, by generowane liczby miały tylko dwa miejsca po przecinku dziesiętnym, to metoda `format()` pozwala nam to określić. Oto przykład:

```
>>> '{0}'.format(1.25456)
'1.25456'
>>> '{0:.2f}'.format(1.25456)
'1.25'
```

Pierwsza z przedstawionych instrukcji wyświetla liczbę w jej pierwotnej postaci. Z kolei w drugiej instrukcji zmodyfikowaliśmy łańcuch zamiennika do postaci `{0:.2f}`, co oznacza, że wartości mają być wyświetlane jako liczby zmiennoprzecinkowe (`f`) i zawierać jedynie dwie liczby po przecinku dziesiętnym. Jak pokazują wyświetlone wyniki, faktycznie zostały wyświetlone tylko dwa miejsca dziesiętne. Warto zwrócić uwagę, że w przypadku, gdy liczba zawiera więcej miejsc dziesiętnych, niż będzie wyświetlanych, zostanie ona odpowiednio zaokrąglona; oto przykład:

```
>>> '{0:.2f}'.format(1.25556)
'1.26'
```

W tym przykładzie liczba `1.25556` została zaokrąglona do dwóch miejsc dziesiętnych i wyświetlona jako `1.26`. Jeśli używamy łańcucha formatującego `.2f`, a wyświetlana liczba jest całkowita, to na jej końcu zostaną dodane zera:

```
>>> '{0:.2f}'.format(1)
'1.00'
```

Zera zostały dodane, gdyż zażądaliśmy, aby wyświetlana liczba miała dokładnie dwa miejsca dziesiętne.

Konwersja jednostek miar

Międzynarodowy Układ Jednostek Miar definiuje siedem *wielkości podstawowych*. Są one następnie używane do określania *wielkości pochodnych*. Długość (a także szerokość, wysokość i głębokość), czas, masa oraz temperatura należą do wielkości podstawowych. Każda z nich ma swoją jednostkę, odpowiednio są to: metr, sekunda, kilogram oraz kelwin.

Jednak każda z tych jednostek podstawowych posiada także jednostki pochodne. Jesteśmy przyzwyczajeni, że temperatura jest podawana jako 30 stopni Celsjusza lub 86 stopni Fahrenheita, a nie 303,15 kelwina. Czy to oznacza, że temperatura 303,15 kelwina jest trzy razy gorętsza od 86 stopni Fahrenheita? Absolutnie nie! Tych dwóch temperatur nie można porównywać wyłącznie na podstawie wartości liczbowych, gdyż są wyrażone w zupełnie innych jednostkach miary, choć mierzą tę samą wielkość fizyczną — temperaturę. Dwa pomiary jakiejś wielkości fizycznej można porównywać wyłącznie wtedy, gdy będą wyrażone przy użyciu tej samej jednostki miary.

Konwersja jednostek miar może być trudnym zadaniem i dlatego w szkole bardzo często jesteśmy proszeni o rozwiązywanie problemów wymagających wykonania konwersji jednostek. To bardzo dobry sposób na przetestowanie podstawowych umiejętności matematycznych. Python jest bardzo dobry z matematyki, a w odróżnieniu od niektórych uczniów ciągłe wykonywanie w pętli różnych obliczeń matematycznych nigdy go nie męczy! Dlatego też kolejny program, który napiszemy, będzie pozwalał na przeliczanie jednostek miar.

Zacznijmy od długości. W USA i Wielkiej Brytanii do określania długości są zazwyczaj używane cale i mile, natomiast w większości innych krajów na świecie — centymetry i kilometry.

Cal ma długość 2,54 centymetra, a do przeliczania cali na centymetry można użyć operacji mnożenia. Tak wyznaczony wynik można następnie podzielić przez 100, aby uzyskać długość wyrażoną w metrach. Poniższy przykład pokazuje, jak można przeliczyć 25,5 cala na metry:

```
>>> (25.5 * 2.54) / 100
0.6476999999999999
```

Z kolei mila to około 1,609 kilometra, a zatem jeśli nasz cel leży w odległości 650 mil, to będzie to oznaczało, że musimy pokonać 650·1,609 kilometra:

```
>>> 650 * 1.609
1045.85
```

Teraz zajmijmy się konwersją jednostek *temperatury*, a konkretnie zamianą stopni Fahrenheita na Celsjusza i na odwrót. Temperaturę wyrażoną w stopniach Fahrenheita można przeliczyć na stopnie Celsjusza, używając następującego wzoru:

$$C = (F - 32) \cdot \frac{5}{9}$$

Przy czym *F* to temperatura wyrażona w stopniach Fahrenheita, a *C* to odpowiadająca jej temperatura wyrażona w stopniach Celsjusza. Wiadomo, że 98,6 stopnia Fahrenheita to normalna temperatura ludzkiego ciała. Aby wyliczyć tę temperaturę wyrażoną w stopniach Celsjusza, zapiszemy powyższy wzór w formie wyrażenia języka Python:

```
>>> F = 98.6
>>> (F - 32) * (5 / 9)
37.0
```

Najpierw utworzyliśmy etykietę `F` i skojarzyliśmy ją z liczbą `98.6` — temperaturą wyrażoną w stopniach Fahrenheita. Następnie zapisaliśmy wzór służący do skonwertowania tej temperatury na stopnie Celsjusza, a po jego wykonaniu okazało się, że temperaturze `98.6` stopnia Fahrenheita odpowiada `37.0` stopni Celsjusza.

Aby skonwertować temperaturę ze stopni Celsjusza na Fahrenheita, należy użyć poniższego wzoru:

$$F = \left(C \cdot \frac{9}{5}\right) + 32$$

Ten wzór możemy zastosować w kodzie Pythona w analogiczny sposób:

```
>>> C = 37
>>> C * (9 / 5) + 32
98.60000000000001
```

W pierwszym wierszu utworzyliśmy etykietę `C` i skojarzyliśmy ją z liczbą `37` (normalną temperaturą ludzkiego ciała). W drugim wierszu skonwertowaliśmy tę temperaturę na stopnie Fahrenheita, uzyskując wartość `98.6`.

Ciągłe wpisywanie tych wzorów byłoby jednak bardzo uciążliwe. Dlatego napiszemy program, który za nas będzie wykonywał te konwersje. Program będzie wyświetlał menu pozwalające użytkownikowi na wybranie konwersji, którą chce wykonać, następnie poprosi o podanie wartości do skonwertowania, po czym wyświetli uzyskany wynik. Jego kod przedstawiłem poniżej:

```
'''
Konwerter jednostek: mile i kilometry
'''

def print_menu():
    print('1. Kilometry na mile')
    print('2. Mile na kilometry')

def km_miles():
    km = float(input('Wpisz odległość wyrażoną w kilometrach: '))
    miles = km / 1.609

    print('Odległość wyrażona w milach: {0}'.format(miles))
```

```
def miles_km():
    miles = float(input('Wpisz odległość wyrażoną w milach: '))
    km = miles * 1.609

    print('Odległość wyrażona w kilometrach: {0}'.format(km))

if __name__ == '__main__':
    print_menu() ❶
    choice = input('Wybierz rodzaj konwersji, którą chcesz wykonać: ') ❷
    if choice == '1':
        km_miles()

    if choice == '2':
        miles_km()
```

Ten program jest nieco dłuższy od poprzednich, jednak nie ma się czego obawiać. Tak naprawdę jest całkiem prosty. Zacznijmy od wiersza oznaczonego cyfrą ❶. W tym wierszu wywoływana jest funkcja `print_menu()`, która wyświetla na ekranie menu z dwiema dostępnymi opcjami konwersji. W wierszu ❷, jeśli użytkownik wpisze z klawiatury liczbę `1` (kilometry na mile), to zostanie wywołana funkcja `km_miles()`. Jeśli natomiast użytkownik wpisze liczbę `2` (mile na kilometry), to zostanie wywołana funkcja `miles_km()`. Każda z tych funkcji w pierwszej kolejności prosi użytkownika o wpisanie wielkości wyrażonej w jednostkach, które mają zostać skonwertowane (czyli w kilometrach w przypadku funkcji `km_miles()` oraz w milach w przypadku funkcji `miles_km()`). Następnie program wykonuje konwersję, używając odpowiedniego wzoru, i wyświetla uzyskany wynik.

Poniżej przedstawiłem przykładowe wyniki wygenerowane przez program:

```
1. Kilometry na mile
2. Mile na kilometry
Wybierz rodzaj konwersji, którą chcesz wykonać: 2 ❶
Wpisz odległość wyrażoną w milach: 100
Odległość wyrażona w kilometrach: 160.9
```

Użytkownik został poproszony o dokonanie wyboru w wierszu oznaczonym cyfrą ❶ i w odpowiedzi wpisał cyfrę `2` (chcąc skonwertować odległość wyrażoną w milach na kilometry). Następnie program poprosił użytkownika o podanie konwertowanej odległości w milach, po czym wyświetlił wynik konwersji.

Na razie program konwertuje jedynie kilometry i mile, jednak w wyzwaniach programistycznych zamieszczonych na końcu tego rozdziału rozbudujesz go tak, aby konwertował także wielkości wyrażone w innych jednostkach.

Obliczanie pierwiastków równań kwadratowych

Co robimy, kiedy dysponujemy równaniem takim jak $x + 500 - 79 = 0$ i musimy wyznaczyć wartość zmiennej x? W tym przypadku wystarczy przeorganizować czynniki równania, tak by zmienna była po jednej stronie znaku równości,

a stałe (500, –79 i 10) po drugiej. W efekcie uzyskujemy równanie o postaci: $x = 10 - 500 + 79$.

Wyznaczenie wartości wyrażenia po prawej stronie pozwala nam poznać wartość zmiennej x, która będzie stanowić nasze rozwiązanie i która jest nazywana *pierwiastkiem* równania. W języku Python powyższe równanie można rozwiązać w następujący sposób:

```
>>> x = 10 - 500 + 79
>>> x
-411
```

Ten przykład przedstawia *równanie liniowe*. Po odpowiednim zapisaniu czynników po obu stronach znaku równości wyliczenie wyniku takiego równania jest dość proste. Z drugiej strony w przypadku równań takich jak $x^2 + 2x + 1 = 0$ znalezienie pierwiastków zmiennej x wiąże się z obliczeniem złożonego wzoru, nazywanego *wzorem równania kwadratowego*. Równania takie jak powyższe są nazywane *równaniami kwadratowymi* i mają ogólną postać $ax^2 + bx + c = 0$, gdzie a, b i c są jakimiś stałymi. Wzory pozwalające na wyznaczenie pierwiastków równania kwadratowego mają następującą postać:

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \text{ oraz } x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

Równanie kwadratowe ma dwa pierwiastki — dwie wartości, dla których obie strony równania kwadratowego będą sobie równe (choć czasami może się okazać, że te dwie wartości są równe). Odzwierciedlają to dwa warianty wzoru równania kwadratowego, na x_1 i x_2.

Gdy porówna się równanie $x^2 + 2x + 1 = 0$ z ogólną postacią równania kwadratowego, widać, że $a = 1$, $b = 2$, a $c = 1$. Te wartości można wstawić bezpośrednio do wzoru równania kwadratowego i wyliczyć x_1 i x_2. Chcąc wyznaczyć pierwiastki równania kwadratowego w języku Python, w pierwszym kroku musimy zapisać wartości a, b i c odpowiednio w etykietach `a`, `b` i `c`:

```
>>> a = 1
>>> b = 2
>>> c = 1
```

Następnie warto zauważyć, że w obu wzorach występuje to samo wyrażenie $b^2 - 4ac$; możemy zatem zdefiniować nową etykietę D, spełniającą warunek $D = \sqrt{b^2 - 4ac}$:

```
>>> D = (b**2 - 4*a*c)**0.5
```

Jak widać, pierwiastek kwadratowy wyrażenia $b^2 - 4ac$ obliczyliśmy, podnosząc je do potęgi 0,5. Teraz już możemy napisać wzory wyznaczające wartości x_1 i x_2:

```
>>> x_1 = (-b + D)/(2*a)
>>> x_1
-1.0
>>> x_2 = (-b - D)/(2*a)
>>> x_2
-1.0
```

W tym przypadku wartości obu pierwiastków równania kwadratowego są takie same i jeśli każdą z nich podstawimy do równania $x^2 + 2x + 1 = 0$, to okaże się, że jego wartość wyniesie 0.

Nasz kolejny program łączy wszystkie powyższe czynności w jednej funkcji o nazwie `roots()`, która pobiera wartości *a*, *b* i *c* w formie parametrów, wylicza pierwiastki równania, a następnie je wyświetla:

```
'''
Obliczanie pierwiastków równania kwadratowego
'''

def roots(a, b, c):
    D = (b*b - 4*a*c)**0.5
    x_1 = (-b + D)/(2*a)
    x_2 = (-b - D)/(2*a)

    print('x1: {0}'.format(x_1))
    print('x2: {0}'.format(x_2))

if __name__ == '__main__':
    a = input('Wpisz wartość a: ')
    b = input('Wpisz wartość b: ')
    c = input('Wpisz wartość c: ')
    roots(float(a), float(b), float(c))
```

W tym programie w pierwszej kolejności tworzymy etykiety `a`, `b` i `c`, z którymi kojarzymy wartości trzech stałych równania kwadratowego. Następnie wywołujemy funkcję `roots()`, przekazując do niej te wartości jako argumenty (przy czym wcześniej konwertujemy je na liczby zmiennoprzecinkowe). Ta funkcja podstawia przekazane wartości do wzoru równania kwadratowego, wylicza pierwiastki tego równania i je wyświetla.

Po uruchomieniu program poprosi użytkownika o podanie wartości *a*, *b* i *c* określających stałe równania kwadratowego, którego pierwiastki użytkownik chce wyliczyć.

```
Wpisz wartość a:  1
Wpisz wartość b:  2
Wpisz wartość c:  1
x1: -1.0
x2: -1.0
```

Warto spróbować rozwiązać jeszcze kilka innych równań kwadratowych, podając inne wartości stałych; okaże się, że za każdym razem obliczone przez program pierwiastki równania będą prawidłowe.

Najprawdopodobniej wiesz, że pierwiastkami równania kwadratowego mogą być także liczby zespolone. Na przykład właśnie nimi są pierwiastki równania $x^2 + x + 1 = 0$. Przedstawiony program może znajdować także takie pierwiastki. Sprawdźmy, czy faktycznie tak jest — uruchommy program jeszcze raz i wpiszmy następujące wartości stałych: $a = 1$, $b = 1$ i $c = 1$:

```
Wpisz wartość a: 1
Wpisz wartość b: 1
Wpisz wartość c: 1
x1: (-0.49999999999999994+0.8660254037844386j)
x2: (-0.5-0.8660254037844386j)
```

Jak widać, wyświetlone pierwiastki są liczbami zespolonymi (o czym świadczy wystąpienie litery `j`), a program nie miał najmniejszych problemów z ich wyliczeniem i wyświetleniem.

Czego nauczyłeś się w tym rozdziale

Świetna robota — udało Ci się zakończyć pierwszy rozdział książki! Nauczyłeś się w nim pisać programy, które rozpoznają liczby całkowite, zmiennoprzecinkowe, zespolone oraz ułamki (zapisywane w formie ułamków zwykłych lub liczb z częścią po przecinku dziesiętnym). Zobaczyłeś programy, które generują tabliczkę mnożenia, przeliczają jednostki miar i obliczają pierwiastki równania kwadratowego. Sądzę, że jesteś bardzo podekscytowany wykonaniem pierwszego kroku na drodze do pisania programów, które będą za Ciebie wykonywać obliczenia matematyczne. Jednak zanim przejdziemy do kolejnego rozdziału, przedstawię kilka wyzwań programistycznych, które dadzą Ci możliwość dalszego utrwalenia zdobytej wiedzy.

Wyzwania programistyczne

Poniżej zamieściłem kilka wyzwań, które umożliwią Ci przećwiczenie zagadnień opisanych w tym rozdziale. Każdy z przedstawionych problemów może zostać rozwiązany na wiele różnych sposobów, a przykładowe rozwiązania można znaleźć w dodatku C.

Nr 1. Automat parzysty – nieparzysty

Spróbuj napisać program symulujący „automat parzysty – nieparzysty”, który pobiera od użytkownika liczbę i wykonuje dwie operacje:

1. Wyświetla informację, czy liczba jest parzysta, czy nieparzysta.
2. Wyświetla podaną liczbę oraz 9 kolejnych liczb parzystych lub nieparzystych.

Jeśli użytkownik wpisze `2`, program powinien wyświetlić komunikat `Liczba parzysta`, a następnie ciąg liczb `2, 4, 6, 8, 10, 12, 14, 16, 18, 20`. Jeśli użytkownik wpisze wartość `1`, program powinien wyświetlić komunikat `Liczba nieparzysta` i ciąg liczb `1, 3, 5, 7, 9, 11, 13, 15, 17, 19`.

Program powinien używać metody `is_integer()`, by wyświetlać komunikat o błędzie, gdy wpisana liczba będzie mieć jakieś znaczące cyfry po przecinku dziesiętnym.

Nr 2. Ulepszony generator tabliczki mnożenia

Nasz program generujący tabliczkę mnożenia jest całkiem fajny, jednak wyświetla tylko dziesięć pierwszych wielokrotności podanej liczby. Rozbuduj program w taki sposób, by użytkownik mógł podać nie tylko liczbę, dla której zostanie wygenerowana tabliczka mnożenia, lecz także *ile wielokrotności* należy w tej tabliczce uwzględnić. Na przykład użytkownik powinien mieć możliwość sporządzenia tabliczki mnożenia dla liczby 9, uwzględniającej jej 15 wielokrotności.

Nr 3. Ulepszony konwerter jednostek

Możliwości programu do konwersji jednostek, który napisaliśmy we wcześniejszej części rozdziału, ograniczają się do przeliczania kilometrów na mile i na odwrót. Spróbuj rozszerzyć program o konwersję jednostek masy (na przykład kilogramy na funty i na odwrót) i temperatury (na przykład stopnie Celsjusza na Fahrenheita i na odwrót).

Nr 4. Kalkulator ułamków

Napisz kalkulator, który będzie potrafił wykonywać podstawowe działania matematyczne na dwóch ułamkach. Powinien on prosić użytkownika o wpisanie dwóch ułamków oraz wybranie operacji, którą należy wykonać. W ramach ułatwienia pracy poniżej przedstawiłem program, który pozwala na wykonywanie tylko jednej operacji — dodawania:

```
'''
Operacje na ułamkach
'''

from fractions import Fraction

def add(a, b):
    print('Wynik dodawania: {0}'.format(a+b))

if __name__ == '__main__':
    a = Fraction(input('Wpisz pierwszy ułamek: ')) ❶
    b = Fraction(input('Wpisz drugi ułamek: ')) ❷
```

```
op = input('Wpisz, którą operację chcesz wykonać: dodaj, odejmij, podziel, pomnóż: ')
if op == 'dodaj':
    add(a,b)
```

Większość rozwiązań zastosowanych w tym programie mogłeś zobaczyć już wcześniej. W wierszach ❶ i ❷ prosimy użytkownika o wpisanie ułamków. Następnie prosimy go o wpisanie operacji, którą na tych ułamkach będzie chciał wykonać. Jeśli użytkownik wpisze `'dodaj'`, to zostanie wywołana funkcja `add()`, którą zdefiniowaliśmy na początku kodu i która oblicza sumę dwóch ułamków przekazanych jako argumenty. Funkcja `add()` wykonuje dodawanie i wyświetla uzyskany wynik. Oto przykład wykonania tego programu:

```
Wpisz pierwszy ułamek:  3/4
Wpisz drugi ułamek:  1/4
Wpisz, którą operację chcesz wykonać: dodaj, odejmij, podziel, pomnóż:  dodaj
Wynik dodawania: 1
```

Spróbuj rozbudować ten program o inne operacje matematyczne: odejmowanie, dzielenie i mnożenie. Oto przykład, jak powinny wyglądać wyniki generowane przez program w przypadku wybrania operacji odejmowania:

```
Wpisz pierwszy ułamek:  3/4
Wpisz drugi ułamek:  1/4
Wpisz, którą operację chcesz wykonać: dodaj, odejmij, podziel, pomnóż:  odejmij
Wynik dodawania: 2/4
```

W przypadku dzielenia powinieneś poinformować użytkownika, czy to pierwszy ułamek jest dzielony przez drugi, czy na odwrót.

Nr 5. Zapewnij użytkownikowi możliwość wyjścia

Wszystkie programy, które do tej pory napisaliśmy, pozwalają użytkownikowi wyłącznie na jednokrotne podanie danych wyjściowych i wykonanie zamierzonych czynności. W ramach przykładu przyjrzyjmy się programowi do generowania tabliczki mnożenia: użytkownik wpisuje liczbę, a program generuje tabliczkę mnożenia dla podanej liczby i kończy działanie. Gdyby użytkownik chciał wygenerować tabliczkę mnożenia dla innej liczby, musiałby ponownie uruchomić program.

Z punktu widzenia użytkownika znacznie wygodniej by było, gdyby miał możliwość wyboru, czy chce zakończyć program, czy kontynuować korzystanie z niego. Kluczem do napisania programu działającego w taki sposób jest stworzenie *pętli nieskończonej*, czy też pętli, której działanie zostaje przerwane, dopiero kiedy jawnie tego zażądamy. Poniżej przedstawiłem szkielet takiego programu:

```
'''
Pętla, która działa, aż zażądamy jej zakończenia
'''
```

```
def fun():
    print('Działam na wieki w nieskończonej pętli...')

if __name__ == '__main__':
    while True: ❶
        fun()
        answer = input('Czy chcesz zakończyć działanie programu? (t) oznacza "tak": ') ❷
        if answer == 't':
            break
```

W tym programie definiujemy pętlę o postaci `while True` (❶). Pętla `while` działa aż do momentu, kiedy jej warunek przyjmie wartość `False`. Ponieważ w naszym programie warunkiem pętli jest stała `True`, zatem będzie ona wykonywana w nieskończoność, chyba że przerwiemy jej działanie w jakiś sposób. Wewnątrz pętli wywołujemy funkcję `fun()`, której działanie ogranicza się do wyświetlenia komunikatu `Działam na wieki w nieskończonej pętli...`. W wierszu oznaczonym liczbą ❷ pytamy użytkownika, czy chce zakończyć działanie programu. Jeśli w odpowiedzi na to pytanie użytkownik wpisze `t`, to wychodzimy z pętli, używając w tym celu instrukcji `break` (instrukcja ta powoduje natychmiastowe wyjście z najbardziej zagnieżdżonej pętli, z pominięciem wszystkich pozostałych instrukcji, które się w niej znajdują). Jeśli użytkownik wpisze coś innego (lub jeśli nic nie wpisze — ograniczy się do naciśnięcia klawisza *Enter*), pętla `while` dalej będzie działać i wyświetlać komunikat… tak długo, aż użytkownik zdecyduje się zakończyć działanie programu. Poniżej przedstawiłem przykładowe wyniki generowane przez ten program:

```
Działam na wieki w nieskończonej pętli...
Czy chcesz zakończyć działanie programu? (t) oznacza "tak":  n
Działam na wieki w nieskończonej pętli...
Czy chcesz zakończyć działanie programu? (t) oznacza "tak":  n
Działam na wieki w nieskończonej pętli...
Czy chcesz zakończyć działanie programu? (t) oznacza "tak":  n
Działam na wieki w nieskończonej pętli...
Czy chcesz zakończyć działanie programu? (t) oznacza "tak":  t
```

Bazując na tym przykładzie, przepiszemy program generujący tabliczkę mnożenia w taki sposób, by działał tak długo, aż użytkownik zażąda jego zakończenia. Tę nową wersję programu przedstawiłem poniżej:

```
'''
Generator tabliczki mnożenia działający w pętli,
którą użytkownik może przerwać
'''

def multi_table(a):
    for i in range(1, 11):
        print('{0} x {1} = {2}'.format(a, i, a*i))

if __name__ == '__main__':
```

```
    while True:
        a = input('Wpisz liczbę: ')
        multi_table(float(a))
        answer = input('Czy chcesz skończyć? (t) oznacza "tak": ')
        if answer == 't':
            break
```

Porównując powyższy program z tym napisanym wcześniej, łatwo można zauważyć, że jedyną zmianą, jaką w nim wprowadziliśmy, jest dodanie pętli `while` zawierającej prośbę o podanie liczby, wywołanie funkcji `multi_table()`, pytanie, czy należy przerwać działanie programu, i instrukcję warunkową z instrukcją `break` umożliwiającą wyjście z pętli.

Kiedy uruchomimy program, poprosi on użytkownika o podanie liczby i wyświetli jej tabliczkę mnożenia; zupełnie tak, jak robił wcześniej. Jednak następnie program zapyta użytkownika, czy ten chce zakończyć jego działanie. Jeśli użytkownik nie wyrazi takiego życzenia, program będzie gotowy do wyświetlenia tabliczki mnożenia dla kolejnej liczby. Poniżej przedstawiłem przykładowe wyniki generowane przez nową wersję programu:

```
Wpisz liczbę: 2
2.0 x 1 = 2.0
2.0 x 2 = 4.0
2.0 x 3 = 6.0
2.0 x 4 = 8.0
2.0 x 5 = 10.0
2.0 x 6 = 12.0
2.0 x 7 = 14.0
2.0 x 8 = 16.0
2.0 x 9 = 18.0
2.0 x 10 = 20.0
Czy chcesz skończyć? (t) oznacza "tak": n
Wpisz liczbę:
```

Spróbuj przepisać któreś z innych programów przedstawionych w tym rozdziale, tak by kontynuowały działanie aż do momentu, gdy użytkownik zażąda ich zakończenia.


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

PYTHON: ROZWIĄZUJ PROBLEMY MATEMATYCZNE W ZUPEŁNIE INNY SPOSÓB!

Mało kto lubi matematykę, zwłaszcza algebrę czy analizę matematyczną. Wydaje się trudna i niezrozumiała. Bardzo łatwo popełnić błędy podczas rozwiązywania równań różniczkowych czy całek. Jeśli jednak powierzysz najtrudniejszą i najżmudniejszą część obliczeń komputerowi, szybko się przekonasz, że to fascynująca dziedzina wiedzy. Docenisz też jej przydatność na różnych płaszczyznach! Programy, które ułatwią Ci rozwiązywanie problemów matematycznych, łatwo napiszesz samodzielnie, w Pythonie. To język, który do tych celów nadaje się idealnie — sprawdź, jak satysfakcjonujące i zabawne jest rozwiązywanie zadań matematycznych z Pythonem!

Dzięki tej książce nauczysz się używać Pythona do rozwiązywania problemów matematycznych z takich dziedzin jak statystyka, geometria, rachunek prawdopodobieństwa czy analiza matematyczna. Zaczniesz od prostych zadań, jak wyznaczanie dzielników liczb całkowitych i rozwiązywanie równań kwadratowych, aby stopniowo przejść do złożonych zagadnień. Napiszesz program do rozwiązywania nierówności, rysowania wykresu toru lotu pocisku, tasowania talii kart, obliczania pola powierzchni koła, badania ciągu Fibonacciego, złotego podziału — i wiele innych. Odkryjesz nowe sposoby poznawania matematyki i zdobędziesz cenne umiejętności programistyczne, z których będziesz mógł korzystać nie tylko podczas nauki!

W książce między innymi:

- opisywanie i wizualizacja danych z wykorzystaniem statystyki oraz różnych wykresów
- teoria zbiorów i rachunku prawdopodobieństwa
- problemy algebraiczne i obliczenia symboliczne
- rysowanie kształtów geometrycznych i badanie fraktali
- pisanie programów do rachunku różniczkowego i całkowego

Amit Saha jest inżynierem oprogramowania, pracował między innymi dla Red Hat i Sun Microsystems. Brał udział w różnych projektach *open source*, w tym SymPy i CPython. Jest twórcą Fedory Scientific — dystrybucji Linuksa przeznaczonej dla naukowców i nauczycieli. Napisał kilka książek technicznych.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-7493-5
9 788328 374935

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 69,00 zł

no starch press