Rozdział 2

Zmienne, wyrażenia i instrukcje

2.1. Wartości i typy

Wartość jest jedną z podstawowych rzeczy, z którymi pracuje program. Wartości, takie jak litery czy cyfry, które widzieliśmy do tej pory, to np. 6 i „Witaj świecie!".

Wartości te należą do różnych *typów*: 6 jest liczbą całkowitą, a „Witaj świecie!" jest *ciągiem znaków*, *napisem* lub *łańcuchem znaków* (ang. *string*). Ty (i interpreter) możesz rozpoznać ciągi znaków, ponieważ są one zawarte między apostrofami lub w cudzysłowie.

Do uruchomienia interpretera używamy komendy `python3`:

```
$ python3
>>>
```

Funkcja `print()`, poza wyświetlaniem tekstu, działa również w przypadku liczb całkowitych:

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

Jeśli nie jesteś pewien, jaki typ ma dana wartość, interpreter może Ci to podpowiedzieć:

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

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

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

A co z takimi wartościami jak `'17'` i `'3.2'`? Wyglądają one jak liczby, ale tak jak ciągi znaków, są zawarte między apostrofami:

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

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

Są to ciągi znaków.

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

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

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

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

2.2. Zmienne

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

Instrukcja przypisania tworzy nowe zmienne i nadaje im wartości:

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

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

Aby wyświetlić wartość zmiennej, możesz skorzystać z funkcji `print()`:

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

Typ zmiennej jest typem wartości, do której się odnosi:

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

[2] Jest to zapis częściej stosowany w USA.

2.3. Nazwy zmiennych i słowa kluczowe

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

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

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

Jeśli nadasz zmiennej niepoprawną nazwę, otrzymasz błąd składniowy:

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

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

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

Python ma zarezerwowanych 35 słów kluczowych:

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

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

2.4. Instrukcje

Instrukcja jest jednostką kodu, którą może wykonać interpreter Pythona. Widzieliśmy ich dwa rodzaje: `print()` jako wyrażenie oraz przypisanie.

Gdy w trybie interaktywnym wpiszesz jakąś instrukcję, to interpreter ją wykona i wyświetli wynik (o ile taki istnieje).

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

Np. poniższy skrypt:

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

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

zwraca następujący wynik:

```
1
2
```

Instrukcja przypisania nie zwraca żadnych wyników.

2.5. Operatory i operandy

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

```
hour = 3
minute = 40
```

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

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

Jeśli pisałeś programy w Pythonie 2, to musisz wiedzieć, że w Pythonie 3 nastąpiła zmiana w operatorze dzielenia. W Pythonie 3 wynik takiego dzielenia jest wynikiem zmiennoprzecinkowym:

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

Operator dzielenia w Pythonie 2 dzieliłby dwie liczby całkowite i obcinałby wynik do liczby całkowitej:

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

Aby uzyskać taki sam wynik w Pythonie 3, należy użyć dzielenia całkowitoliczbowego //, zaokrąglającego w dół:

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

W Pythonie 3 dzielenie liczb całkowitych działa tak, jakbyś wprowadził wyrażenie na kalkulatorze.

2.6. Wyrażenia

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

```
17
x
x + 17
```

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

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

Natomiast w skrypcie wyrażenie samo w sobie nic nie robi! Dla początkujących to częsty powód konsternacji.

Ćwiczenie 1. Wpisz następujące wyrażenia w interpreterze Pythona, tak aby zobaczyć co one robią:

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

2.7. Kolejność wykonywania działań

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

- Nawiasy okrągłe (ang. *P*arentheses) mają najwyższy priorytet i mogą być użyte do wymuszenia ewaluacji wyrażenia w pożądanej kolejności. Ponieważ wyrażenia w nawiasach są ewaluowane jako pierwsze, `2 * (3-1)` to 4, a `(1+1)**(5-2)` to 8. Możesz również użyć nawiasów aby wyrażenie było łatwiejsze do odczytania, tak jak np. w `(minute * 100) / 60`, nawet jeśli nie zmienia to wyniku.

- Potęgowanie (ang. *E*xponentiation) ma kolejny najwyższy priorytet, więc `2**1+1` to 3, a nie 4, a `3*1**3` to 3, a nie 27.

- Mnożenie (ang. *M*ultiplication) i dzielenie (ang. *D*ivision) mają ten sam priorytet, który jest wyższy niż dodawanie (ang. *A*ddition) i odejmowanie (ang. *S*ubtraction), które również mają ten sam priorytet. Tak więc `2*3-1` to 5, a nie 4, a `6+4/2` to 8, a nie 5.

- Operatory z takim samym pierwszeństwem są ewaluowane od lewej strony do prawej. Tak więc wyrażenie `5-3-1` to 1, a nie 3, ponieważ `5-3` jest wyliczane najpierw, a potem `1` jest odejmowane od 2.

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

2.8. Operator modulo

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

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

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

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

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

2.9. Operacje na ciągach znaków

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

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

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

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

2.10. Proszenie użytkownika o wprowadzenie danych

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

[4] W Pythonie 2 funkcja ta nazywa się `raw_input()`.

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

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

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

Sekwencja `\n` na końcu komunikatu reprezentuje *nową linię*, specjalny znak, który powoduje przejście do nowej linii. Dlatego dane wprowadzane przez użytkownika znajdują się pod komunikatem.

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

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

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

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

Zobaczymy później, jak poradzić sobie z tego typu błędami.

2.11. Komentarze

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

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

```
minute = 30
# oblicza procent godziny, która upłynęła
percentage = (minute * 100) / 60
```

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

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

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

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

Poniższy komentarz w kodzie jest zbędny i bezużyteczny:

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

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

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

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

2.12. Wybór mnemonicznych nazw zmiennych

Tak długo, jak stosujesz się do prostych zasad nazewnictwa zmiennych i unikasz zastrzeżonych słów, masz duży wybór przy nadawaniu nazw swoim zmiennym. Na początku może to utrudniać zarówno czytanie, jak i pisanie kodu programu. Na przykład następujące trzy programy są identyczne pod względem tego, co robią, ale bardzo różne, gdy je czytasz i starasz się zrozumieć.

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

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

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

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

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

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

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

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

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

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

Poniższy kod jest odpowiednikiem powyższego kodu:

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

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

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

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

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

2.13. Debugowanie

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

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

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

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

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

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

Błąd wykonania (ang. *runtime error*), który może Ci się zdarzać najczęściej, to „użycie przed definicją", co oznacza próbę użycia zmiennej przed przypisaniem wartości. Może do tego dojść, gdy błędnie wpiszesz nazwę zmiennej:

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

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

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

W tym momencie najbardziej prawdopodobną przyczyną błędu semantycznego jest kolejność operacji. Przygotujmy zmienną pi:

```
>>> pi = 3.14
```

Na przykład, aby obliczyć $1/2\pi$, można pokusić się o napisanie:

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

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

2.14. Słowniczek

ciąg znaków Typ danych str, który reprezentuje sekwencje znaków; inaczej napis lub łańcuch znaków.
ewaluacja Uproszczenie wyrażenia poprzez wykonanie operacji w celu uzyskania pojedynczej wartości; inaczej wartościowanie.
instrukcja Część kodu, która reprezentuje polecenie lub akcję. Jak dotąd polecenia, które widzieliśmy, to przypisania i wyrażenie print().
kolejność wykonywania działań Zbiór zasad regulujących kolejność ewaluacji wyrażeń obejmujących wiele operatorów i operandów.
komentarz Informacja w programie, która jest przeznaczona dla innych programistów (lub osób czytających kod źródłowy) i nie ma wpływu na wykonanie programu.
konkatenacja Połączenie dwóch ciągów znaków.
liczba całkowita Typ danych int, który reprezentuje liczby całkowite.
liczba zmiennoprzecinkowa Typ danych float, który reprezentuje liczby z częściami ułamkowymi.
mnemonika Pomoc pamięciowa. Często nadajemy zmiennym nazwy mnemoniczne, tak aby łatwiej zapamiętać, co jest przechowywane w zmiennej.
operand Jedna z wartości, na której działa operator.
operator Specjalny symbol, który reprezentuje proste obliczenie, np. takie jak dodawanie, mnożenie lub konkatenację ciągów.
operator modulo Operator oznaczony znakiem procentu (%), który działa na liczbach całkowitych i zwraca resztę, gdy jedna liczba jest dzielona przez drugą.
przypisanie Instrukcja, która przypisuje wartość zmiennej.
słowo kluczowe Zastrzeżone słowo, które jest używane przez kompilator/interpreter do przetwarzania programu; nie można używać słów kluczowych, np. takich jak if, def i while jako nazw zmiennych.
typ Kategoria wartości. Typy, które widzieliśmy do tej pory, to liczby całkowite (typ int), liczby zmiennoprzecinkowe (typ float) oraz ciągi znaków (typ str).
wartość Jedna z podstawowych jednostek danych, takich jak liczba lub ciąg znaków, którą operuje program.
wyrażenie Kombinacja zmiennych, operatorów i wartości, która reprezentuje pojedynczą wartość wyniku.
zmienna Nazwa, która odnosi się do wartości.

2.15. Ćwiczenia

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

```
Podaj swoje imię: Chuck
Witaj Chuck!
```

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

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

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

Ćwiczenie 4. Załóżmy, że wykonujemy następujące instrukcje przypisania:

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

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

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

Aby sprawdzić swoje odpowiedzi, użyj interpretera Pythona.

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