Розділ 2

Змінні, вирази та інструкції

2.1 Значення та типи

Значення – це одна з базових речей, з якими працює програма, такі як буква або число. Ми вже розглянули значення 1, 2, та «Привіт, світе!».

Ці значення належать до різних *типів*: 2 – ціле число (англ. integer), а «Привіт, світе!» – рядок (*англ. string)*, має таку назву, адже містить «ряд» букв. Ви (та інтерпретатор) можете розпізнати рядок завдяки лапкам, у які він взятий.

Інструкція `print` також працює з цілими числами. Щоб запустити інтерпретатор, достатньо скористатися командою python.

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

Якщо ви не впевнені, до якого типу належить значення, вам підкаже інтерпретатор.

```
>>>  type('Привіт,  Світе!')
<class  'str'>
>>> type(17)
<class  'int'>
```

Цілком логічно, що рядки (англ. strings) належать до типу `str`, а цілі числа (англ. integers) – до типу `int`. Дещо менш очевидно, але десяткові числа належать до типу `float`, оскільки вони подаються у форматі з «рухомою крапкою» (англ. *floating point*).

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

А як щодо таких значень, як «17» та «3.2»? Взяті в лапки, немов рядки, проте виглядають як числа.

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

Все таки, вони є рядками.

У деяких країнах притаманно записувати великі цілі числа з комами, які розділяють групи з трьох цифр, наприклад, 1,000,000. Для Python таке ціле число не є коректним цілим числом (англ. integer), проте цілком допустиме:

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

Що ж, зовсім не те, що ми очікували, чи не так? Python сприймає 1,000,000 як три різні цілі числа, розділені комами, які він друкує з пробілами між ними.

Це перед вами перший приклад семантичної помилки: код працює, проте не робить це «правильно», і водночас не видає помилку.

2.2 Змінні

Одна з найпотужніших можливостей мови програмування – це здатність оперувати *змінними*. Змінна – це ім'я, що посилається на значення.

Інструкція присвоєння створює нові змінні та надає їм значення:

```
>>> message = 'А тепер про дещо зовсім інше'
>>> n = 17
>>> pi = 3.1415926535897931
```

У цьому прикладі показані три різні присвоєння. 1) Роядок присвоюється новій змінній з іменем `message`; 2) ціле число 17 присвоюється до `n`; 3) значення π (наближене) присвоюється `pi`.

Щоб вивести значення змінної, можна скористатися інструкцією `print`:

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

Типом змінної є тип значення, на яке вона посилається.

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

2.3 Назви змінних та ключові слова

Зазвичай програмісти обирають для своїх змінних змістовні назви, які мають конкретне значення, а також описують їхню функціональність.

Довжина назв змінних може бути будь-якою. Вони можуть містити як літери, так і цифри, але вони не можуть починатися з цифри. Дозволено використовувати великі літери, втім, краще починати назви змінних з малих (пізніше ви зрозумієте чому).

Також назва може містити знак підкреслення (_). Він досить часто використовується в назвах, що складаються з кількох слів, наприклад, my_name чи airspeed_of_unladen_swallow. З цього знаку можуть починатися назви змінних, але зазвичай цього уникають, окрім випадків, коли потрібно написати код бібліотеки для загального використання.

Якщо ви присвоїте змінній неправильну назву, у вас виникне синтаксична помилка:

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

`76trombones` – неправильна назва, адже починається з цифри, назва `more@` містить недопустимий символ, @. Що ж не так з `class`?

Виявляється, `class` – одне з *ключових слів* Python. Інтерпретатор використовує ключові слова для розпізнавання структури програми, тому їх не можна використовувати як назви змінних.

У Python налічується 33 ключових слова:

```
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
def         for         lambda      return
```

Радимо тримати цей список під рукою. У разі, якщо інтерпретатор буде скаржитися на одну з ваших змінних, і ви не знатимете причину, перевірте, чи не співпадає вона зі словом з цього списку.

2.4 Інструкції

Інструкція – це одиниця коду, яку може виконати інтерпретатор Python. Ми вже розглянули два типи інструкцій: інструкція виведення (`print`) та інструкція присвоєння.

Під час введення інструкції в інтерактивному режимі, інтерпретатор виконує її та виводить результат на екран, якщо він є.

Скрипт зазвичай містить послідовність інструкцій. Якщо інструкцій декілька, результати з'являтимуться по черзі, відповідно до порядку їх виконання.

Наприклад, скрипт

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

видає такий результат:

```
1
2
```

Інструкція присвоєння не видає жодних результатів.

2.5 Оператори та операнди

Оператори – це спеціальні символи, що позначають обчислювальні дії, такі як додавання та множення. Значення, до яких застосовується оператор, називаються *операндами*.

Оператори +, -, *, /, та ** виконують дії додавання, віднімання, множення, ділення та піднесення до степеня, наприклад:

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

У версіях Python 2.x та Python 3.x є відмінності в обчисленні оператора ділення. У Python 3.x, результатом цього прикладу ділення є число з «рухомою крапкою»:

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

Оператор ділення у версії Python 2.0 ділить два цілих числа та обрізає результат до цілого:

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

Щоб отримати таку саму відповідь у Python 3.0, скористайтеся цілочисельним діленням (// ціле число).

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

У Python 3.0 ділення цілих чисел працює набагато краще, так ніби обчислення такого ж самого виразу на калькуляторі.

2.6 Вирази

Вираз – це поєднання значень, змінних та операторів. Власне, значення вважається виразом, так само як і змінна. Відповідно, нижче наведені цілком коректні вирази (за умови, що змінній x присвоєно значення):

```
17
x
x + 17
```

Якщо ви введете вираз в інтерактивному режимі, інтерпретатор обчислить його і видасть результат:

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

Однак у скрипті вираз нічого не робить самостійно! Це завжди вводить в оману початківців.

Вправа 1: Введіть наступні інструкції в інтерпретаторі Python, щоб побачити що вони роблять:

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

2.7 Порядок виконання операцій

Якщо у виразі кілька операторів, порядок обчислення залежить від *правил пріоритету*. Стосовно математичних операторів, Python дотримується математичних норм. Запам'ятати правила допоможе абревіатура ДСМДДВ:

- ***Д***ужки мають найвищий пріоритет, цим можна скористатись для того, щоб поставити вираз у потрібному для вас порядку. Оскільки вирази в дужках обчислюють першими, 2 * (3-1) дорівнює 4, а (1+1)**(5-2) дорівнює 8. Ви також можете скористатись дужками щоб зробити вираз легшим для прочитання, як от, (minute * 100) / 60, навіть якщо це не змінить результат.

- Наступним за пріоритетом є піднесення до ***с***тепеня, тому 2**1+1 дорівнює 3, а не 4, а 3*1**3 буде 3, а не 27.
- Множення й ділення мають однаковий пріоритет, він вищий, ніж у додавання й віднімання, які також однакові за пріоритетом. Тому 2*3-1 вийде 5, а не 4, а 6+4/2 дорівнює 8, а не 5.
- Оператори з однаковим пріоритетом обчислюються послідовно зліва направо. У такому разі, вираз 5-3-1 дорівнює 1, не 3, тому що спочатку обчислюємо 5-3, а потім від 2 віднімаємо 1.

Якщо у вас виникають сумніви, завжди ставте дужки у виразах, щоб переконатися, що обчислення будуть виконані у потрібному порядку.

2.8 Модульний оператор

Модульний оператор працює з цілими числами і повертає остачу при діленні першого операнда на другий. У Python модульним оператором є знак відсотка (%). Синтаксис залишається таким самим, як і для інших операторів:

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

Отже, 7 поділити на 3 дорівнює 2 з остачею 1.

Виявляється, модульний оператор напрочуд корисний. От наприклад, можна перевірити, чи ділиться одне число на інше: якщо x % y дорівнює нулю, то x ділиться на y.

Також із числа можна видобути крайню праву цифру чи кілька цифр. Наприклад, x % 10 видобуває крайню праву цифру числа x (у системі числення з основою 10). Подібним чином, x % 100 видобуває дві останні цифри.

2.9 Операції з рядками

Оператор + працює з рядками, проте він не здійснює додавання в математичному сенсі. Замість цього він виконує *конкатенацію*, тобто об'єднує рядки. Наприклад:

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

```
>>> second = '150'
>>> print(first + second)
100150
```

Оператор * також працює з рядками, він повторює вміст рядка ціле число разів. Наприклад:

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

2.10 Запит користувача на введення даних

Іноді виникає необхідність отримати значення змінної від користувачів за допомогою клавіатури. У Python є вбудована функція `input`, яка отримує дані безпосередньо з клавіатури[1]. Під час виклику цієї функції, програма зупиняється і чекає, поки користувач щось введе. Коли він / вона натискає клавішу Enter, програма відновлює роботу і функція `input` видає введене користувачем значення у вигляді рядка.

```
>>> inp = input()
бла бла бла
>>> print(inp)
бла бла бла
```

Перед тим, як отримати дані від користувача, доцільно вивести запит, який підкаже, що саме потрібно ввести. Ви можете надати `input` рядок, який буде показано користувачеві перед паузою для введення:

```
>>> name = input("Введіть ваше ім'я\n")
Введіть ваше ім'я:
Михайло
>>> print(name)
Михайло
```

Знак \n у кінці запиту позначає новий рядок, який є окремим символом, що викликає розрив рядка. Саме тому дані, введені користувачем, з'являються під запитом.

Якщо користувач має ввести ціле число, спробуйте перетворити повернене значення до типу int за допомогою функції int():

```
>>> prompt = 'Яка… швидкість польоту ластівки без навантаження?\n'
>>> speed = input(prompt)
Яка… швидкість польоту ластівки без навантаження?
17
```

[1] У версії Python 2.0 ця функція має назву raw_input.

```
>>> int(speed)
17
>>> int(speed) + 5
22
```

Проте, якщо користувач введе не рядок цифр, а щось інше, Ви отримаєте помилку:

```
>>> speed = input(prompt)
Яка… швидкість польоту ластівки без навантаження?
Кого ви маєте на увазі, африканську чи європейську ластівку?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
```

Пізніше розглянемо, яким чином можна впоратися з помилкою такого типу.

2.11 Коментарі

Більші та складніші програми важчі у прочитанні. Формальні мови досить щільні, тому зазвичай доволі непросто зрозуміти, що і навіщо робить той чи інший фрагмент коду.

Саме тому варто робити примітки до ваших програм, у яких описується природною мовою суть їхньої роботи. Такі примітки називаються *коментарями*, і в Python вони починаються з символу #:

```
# обчислення відсотків у годині
percentage = (minute * 100) / 60
```

У цьому випадку коментар з'являється в рядку окремо. Крім того, можна додавати коментарі та в кінці рядка:

```
percentage = (minute * 100) / 60     # відсоток години
```

Усі символи від # до кінця рядка ігноруються, вони не впливають на роботу програми.

Коментарі особливо корисні для опису неочевидних можливостей коду. Логічно припустити, що читач і сам може зрозуміти *що* робить код, тому набагато доцільніше пояснити *чому* він це робить.

Наведений коментар зайвий, у ньому немає сенсу:

```
v = 5    # присвоєння значення 5 змінній v
```

А ось цей коментар містить необхідну інформацію, яку важко зрозуміти з коду:

```
v = 5    # швидкість в метрах/секунду.
```

Вдалі назви змінних зменшують необхідність у створенні коментарів, але занадто довгі назви можуть ускладнити читання складних виразів, тож існує своєрідний компроміс.

2.12 Вибір мнемонічних назв для змінних

Якщо ви дотримуєтесь простих правил найменування змінних і уникаєте зарезервованих слів, у вас є великий вибір назв для них. Спершу цей вибір може збити з пантелику під час читання готових програм та написання власних. Наприклад, наведені нижче три програми однакові стосовно своїх завдань, однак дуже відрізняються, якщо їх прочитати та спробувати зрозуміти.

```
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)
```

Інтерпретатор Python сприймає всі три програми за *однакові*, а людина бачить та сприймає їх зовсім інакше. Людина швидше зрозуміє *призначення* другої програми, тому що програміст підібрав назви змінних, які відображають його наміри стосовно того, які дані будуть зберігатися в кожній з них.

Такі продумані імена змінних ми називаємо «мнемонічними назвами змінних». Слово *мнемоніка*[2] означає «спосіб запам'ятовування». Обрані мнемонічні назви змінних насамперед допомагають запам'ятати сенс створеної змінної.

Хоч це все і звучить чудово, й це справді хороша ідея скористатися мнемонічними назвами, проте вони можуть заважати програмісту-початківцю проаналізувати та зрозуміти код. Причина в тому, що новачки ще не встигли запам'ятати зарезервовані слова (їх всього 33), тому іноді змінні з надто описовими назвами починають виглядати як частина мови, а не просто вдало підібрані назви для змінних.

Погляньмо на наступний приклад коду Python, який циклічно переглядає певні дані. Незабаром ми розберемо тему циклів, а поки спробуйте просто розібратися з тим, що це може означати:

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

[2]Детальніше про "мнемоніка" див. на https://en.wikipedia.org/wiki/Mnemonic.

Що ми бачимо? Які з токенів (for, word, in тощо) належать до зарезервованих слів, а які просто є назвами змінних? Чи розуміє Python на базовому рівні поняття слів? Програмістам-початківцям важко відрізнити, які частини коду *повинні* бути однаковими, як у цьому прикладі, а які просто були обрані програмістом.

Наведений нижче код рівнозначний зазначеному вище:

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

Новачку легше розглянути саме цей приклад і зрозуміти, які частини є зарезервованими словами, заданими Python, а які – просто назвами змінних, що обрав програміст. Цілком очевидно, що Python не має фундаментального уявлення про поняття «піца» ("pizza") і «шматочки» ("slice"), а також про те, що піца складається з сукупності кількох шматочків.

Втім, якщо наша програма дійсно спрямована на зчитування даних і пошук слів у них, то pizza і slice зовсім не відповідають значенню мнемонічних назв змінних. Якщо використовувати їх як назви змінних, вони відволікатимуть від сенсу програми.

Через досить короткий проміжок часу ви засвоїте найпоширеніші зарезервовані слова і вони навіть почнуть кидатися вам у очі:

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

Частини коду, визначені Python (`for`, `in`, `print`, and :), виділені жирним шрифтом, а змінні, обрані програмістом, (`word` and `words`) – ні. Багато текстових редакторів враховують синтаксис Python і будуть виділяти зарезервовані слова різним кольором, аби допомогти вам відокремити змінні від зарезервованих слів. Через деякий час ви зможете читати Python і швидко визначати, що є змінною, а що – зарезервованим словом.

2.13 Налагодження програм

На цьому етапі, синтаксична помилка, яку ви, найімовірніше, можете допустити, – неправильна назва змінної, наприклад, `class` чи `yield`, адже вони є ключовими словами, або ж `odd~job` та `US$`, що містять недопустимі символи.

Якщо поставити пробіл в назві змінної, Python вважатиме, що це два операнди без оператора:

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


>>> month = 09
  File "<stdin>", line 1
    month = 09
            ^
SyntaxError:  invalid  token
```

У випадку із синтаксичними помилками, сповіщення про них не надто допомагають. Найпоширенішими повідомленнями є `SyntaxError: invalid syntax` та `SyntaxError: invalid token`, жодне з них не дуже інформативне, чи не так?

Найбільш ймовірна помилка під час виконання програми – «use before def;» тобто спроба застосувати змінну до того, як їй було присвоєно значення. Це може статися, якщо ви неправильно ввели ім'я змінної:

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

Назви змінних враховують особливості літер, тому `LaTeX` не те ж саме, що й `latex`.

Ймовірною причиною семантичної помилки на цьому етапі є порядок виконання операцій. Наприклад, щоб обчислити 1/2π, ви можете помилково написати

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

Але ж спочатку виконується ділення, тому результатом буде π/2, а це далеко не одне й те ж саме! Python не може передбачити, що саме ви хотіли написати, тому в цьому випадку програма не видасть повідомлення про помилку, ви просто отримаєте хибну відповідь.

2.14 Словник

assignment (присвоєння) Інструкція, яка присвоює значення змінній.

concatenate (конкатенувати) З'єднувати два операнди.

comment (коментар) Інформація у програмі, призначена для інших програмістів (або будь-кого, хто читає вихідний код), яка не впливає на виконання програми.

evaluate (обчислювати) Спрощувати вираз шляхом виконання дій, щоб отримати єдине значення.

expression (вираз) Комбінація змінних, операторів і значень, яка представляє єдине значення результату.

floating point (з плаваючою крапкою) Тип, що позначає дробові числа.

integer (ціле число) Тип, що позначає цілі числа.

keyword (ключове слово) Зарезервоване слово, що використовується компілятором для аналізу програми; заборонено використовувати такі ключові слова, як `if`, `def`, та `while` як назви змінних.

mnemonic (мнемоніка) Спосіб запам'ятовування. Змінним часто дають мнемонічні назви, які допомагають краще запам'ятати дані, що зберігаються у них.

modulus operator (модульний оператор) Оператор, який позначають знаком відсотка (%), що працює з цілими числами і повертає остачу під час ділення.

operand (операнд) Одне зі значень, над яким проводить дії оператор.

operator (оператор) Окремий символ, який позначає прості обчислення, такі як додавання, множення чи конкатенація рядків.

rules of precedence (правила пріоритету) Набір правил, що визначають порядок обчислення виразів з кількома операторами та операндами.

statement (інструкція) Частина коду, яка позначає команду або дію. Наразі були розглянуті такі інструкції, як інструкція присвоєння та інструкція виведення виразу на екран.

string (рядок) Тип, що представляє послідовність символів.

type (тип) Категорія значень. Типи, які вже були розглянуті: цілі числа (тип `int`), числа з рухомою крапкою (тип `float`) і рядки (тип `str`).

value (значення) Основна одиниця даних, зокрема, число або рядок, якими оперує програма.

variable (змінна) Ім'я, що посилається на значення.

2.15 Вправи

Вправа 2: Напишіть програму, яка на основі введених даних запитує у користувача його ім'я, а потім вітає його.

```
Enter your name: Chuck
Hello Chuck
```

Вправа 3: Напишіть програму, яка запитує у користувача кількість годин і ставку за годину для розрахунку заробітної плати.

```
Enter Hours: 35
Enter Rate: 2.75
Pay: 96.25
```

Поки що можна не турбуватися про точність двох цифр після десяткової крапки. За бажанням, можете трохи побавитись з вбудованою функцією Python, щоб правильно округлити отриману суму до двох десяткових знаків після крапки.

Вправа 4: Уявіть, що потрібно виконати наступні інструкції присвоєння:

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

Для кожного з наступних виразів виведіть їхнє значення та тип (значення виразу).

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

Скористайтеся інтерпретатором Python, щоб перевірити свої відповіді.

Вправа 5: Напишіть програму, що запитує у користувача температуру за Цельсієм, переводить її у Фаренгейт і виводить на екран перетворене значення.