Словники (Dictionaries)

Словник схожий на список, але порядок елементів не має значення, і вони не вибираються зсувом (offset), таким як 0 або 1. Натомість ви вказуєте унікальний ключ, який потрібно пов'язати з кожним значенням. Цей ключ часто є рядком, але насправді це може бути будь-який із незмінних типів Python: boolean, integer, float, tuple, string та інші. Словники змінюються, тому ви можете додавати, видаляти та змінювати їх елементи ключ-значення.

Створення за допомогою {}

Щоб створити словник, потрібно поставити фігурні дужки ({}) навколо пар розділених комами ключ: значення. Найпростіший словник - це порожній, який взагалі не містить ключів або значень:

```
>>> empty_dict = {}
>>> empty_dict
{}
```

А ось приклад словника телефонних номерів:

```
phonebook = {'Chris':'555-1111', 'Katie':'555-2222', 'Joanne':'555-3333'}
```

Створення за допомогою dict()

```
>>> customer = dict(first="Wile", middle="E", last="Coyote")
>>> customer
{'first': 'Wile', 'middle': 'E', 'last': 'Coyote'}
```

Перетворення за допомогою dict()

Ви також можете використовувати функцію dict() для перетворення двозначних послідовностей у словник. Ви можете інколи стикатися з такими послідовностями "ключ-значення", як-от "Strontium, 90, Carbon, 14." Перший елемент у кожній послідовності використовується як ключ, а другий як значення.

Ось невеликий приклад використання lol (список списків із двох пунктів):

```
>>> lol = [ ['a', 'b'], ['c', 'd'], ['e', 'f'] ]
>>> dict(lol)
{'a': 'b', 'c': 'd', 'e': 'f'}
```

Додавання або зміна елемента за допомогою [key]

Додати елемент до словника легко. Просто зверніться до елемента за його ключем і призначте значення. Якщо ключ вже був у словнику, існуюче значення замінюється новим. Якщо ключ новий, його значення додається до словника. На відміну від списків, вам не потрібно турбуватися про те, що Python під час призначення призначатиме виняток, вказавши індекс, який виходить за межі діапазону.

```
>>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222', 'Joanne':'555-3333'}
>>> phonebook['Joe'] = '555-0123'
>>> phonebook['Chris'] = '555-4444'
>>> phonebook
{'Chris': '555-4444', 'Joanne': '555-3333', 'Joe': '555-0123', 'Katie': '555-2222'}
```

```
>>>
```

Отримання елемента за допомогою [key] або за допомогою get()

Це найпоширеніше використання словника. Ви вказуєте словник і ключ, щоб отримати відповідне значення: Використаємо `phonebook` з попереднього прикладу:

>>>phonebook[‘Chris’]

‘555-4444’

Можна також використати метод gey() словника.

>>> phonebook.get([‘Chris’])

‘555-4444’

Отримання всіх ключів за допомогою keys()

Ви можете використовувати keys(), щоб отримати всі ключі у словнику. Для наступних кількох прикладів ми будемо використовувати такий зразок словника:

```
>>> signals = {'green': 'go', 'yellow': 'go faster', 'red':
'smile for the camera'}
>>> signals.keys()
dict_keys(['green', 'yellow', 'red'])
```

У Python 2 key() просто повертає список. Python 3 повертає dict_keys(), який є ітерабельним переглядом ключів.

Але часто вам справді потрібен список. У Python 3 вам потрібно викликати list(), щоб перетворити об’єкт dict_keys у список.

```
>>> list( signals.keys() )
['green', 'yellow', 'red']
```

Отримання всіх значень за допомогою values()

Щоб отримати всі значення у словнику, використовуйте values ():

```
>>> list( signals.values() )
['go', 'go faster', 'smile for the camera']
```

Отримання всіх пар ключ-значення за допомогою items()

Якщо ви хочете отримати всі пари ключ-значення зі словника, використовуйте функцію items ():

```
>>> list( signals.items() )
[('green', 'go'), ('yellow', 'go faster'), ('red', 'smile for
the camera')]
```

Кожен ключ і значення повертаються у вигляді кортежа, наприклад ('green', 'go').

Отримання довжини за допомогою len ()

Порахуйте свої пари ключ-значення:

```
>>> len(signals)
3
```

Поєднання словників за допомогою {** a, ** b}

Починаючи з Python 3.5, є новий спосіб об’єднання словників за допомогою **

```
>>> first = {'a': 'agony', 'b': 'bliss'}
>>> second = {'b': 'bagels', 'c': 'candy'}
```

```
>>> {**first, **second}
{'a': 'agony', 'b': 'bagels', 'c': 'candy'}
```

Поєднання словників за допомогою update()

Ви можете скористатися функцією update(), щоб скопіювати ключі та значення одного словника в інший.

```
>>> first = {'a': 'agony', 'b': 'bliss'}
>>> second = {'b': 'bagels', 'c': 'candy'}
>>> first.update(second)
>>> first
{'a': 'agony', 'b': 'bagels', 'c': 'candy'}
```

Видалити елемент з заданим ключем за допомогою del

```
>>> del first['b']
>>> first
{'a': 'agony', 'c': 'candy'}
```

Отримати елемент за ключем і видалити його за допомогою pop ()

Це поєднання get () та del. Якщо ви надаєте pop () ключ і він існує у словнику, він повертає відповідне значення і видаляє пару ключ-значення. Якщо його немає, він створює виняток:

```
>>> first = {'a': 'agony', 'b': 'bliss', 'c': 'candy'}
>>> len(first)
3
>>> first.pop('b')
'bliss'
>>> len(first)
2
```

Видалити всі елементи з clear ()

Щоб видалити всі ключі та значення зі словника, використовуйте clear () або просто призначте порожній словник ({}) імені:

```
>>> first.clear()
>>> first
{}
>>> first = {}
>>> first
{}
```

Перевірка наявності ключа з in

Якщо ви хочете дізнатися, чи існує ключ у словнику, скористайтеся in.

```
>>> first = {'a': 'agony', 'b': 'bliss', 'c': 'candy'}
>>> 'a' in first
True
```

Призначення за допомогою =

Як і у випадку зі списками, якщо ви внесете зміни до словника, це відобразиться у всіх іменах, які посилаються на нього:

```
>>> signals = {'green': 'go',
... 'yellow': 'go faster',
... 'red': 'smile for the camera'}
>>> save_signals = signals
>>> signals['blue'] = 'confuse everyone'
>>> save_signals
```

```
{'green': 'go',
'yellow': 'go faster',
'red': 'smile for the camera',
'blue': 'confuse everyone'}
```

Копіювання за допомогою copy ()

Щоб насправді скопіювати ключі та значення зі словника в інший словник, ви можете скористатися copy ():

```
>>> signals = {'green': 'go',
... 'yellow': 'go faster',
... 'red': 'smile for the camera'}
>>> original_signals = signals.copy()
>>> signals['blue'] = 'confuse everyone'
>>> signals
{'green': 'go',
'yellow': 'go faster',
'red': 'smile for the camera',
'blue': 'confuse everyone'}
>>> original_signals
{'green': 'go',
'yellow': 'go faster',
'red': 'smile for the camera'}
>>>
```

Ітерація за допомогою for – in

Ітерація над словником (або його функцією keys ()) повертає ключі. У цьому прикладі ключі - це типи карт у настільній грі Clue (Cluedo outside of North America):

```
>>> accusation = {'room': 'ballroom', 'weapon': 'lead pipe',
... 'person': 'Col. Mustard'}
>>>  for card in accusation:  # or, for card in
accusation.keys():
...     print(card)
...
room
weapon
person
```

Щоб перебирати значення, а не ключі, використовуйте метод values() словника:

```
>>> for value in accusation.values():
...     print(value)
...
ballroom
lead pipe
Col. Mustard
```

Щоб повернути ключ і значення як кортеж, можна скористатися функцією items ():

```
>>> for item in accusation.items():
...     print(item)
...
('room', 'ballroom')
('weapon', 'lead pipe')
```

```
('person', 'Col. Mustard')
```

Генератор словника

Найпростіша форма генератора словника наступна:

```
{key_expression : value_expression for expression in iterable}
```

Наприклад

```
>>> word = 'letters'
>>> letter_counts = {letter: word.count(letter) for letter in word}
>>> letter_counts
{'l': 1, 'e': 2, 't': 2, 'r': 1, 's': 1}
```

Інша форма генератора словника:

```
{key_expression : value_expression for expression in iterable if condition}
```

Наприклад:

```
>>> vowels = 'aeiou'
>>> word = 'onomatopoeia'
>>> vowel_counts = {letter: word.count(letter) for letter in set(word)
if letter in vowels}
>>> vowel_counts
{'e': 1, 'i': 1, 'o': 4, 'a': 2}
```

Функції

Першим кроком для повторного використання коду (code reuse) є функція: іменований фрагмент коду, відокремлений від усіх інших. Функція може приймати будь-яке число та тип вхідних параметрів та повертати будь-яке число та тип результатів виводу.

Ви можете зробити дві речі з функцією:

• Визначити її з нулем або більше параметрів.

• Викликати її і отримати нуль або більше результатів.

Визначення функції за допомогою def

Щоб визначити функцію Python, ви вводите def, ім'я функції, дужки, що містять будь-які *параметри* введення до функції, а потім, нарешті, двокрапку (:). Імена функцій мають ті ж правила, що і імена змінних (вони повинні починатися з літери або _ і містити лише букви, цифри або _).

Ось найпростіша функція Python:

```
>>> def do_nothing():
...     pass
```

Виклик функції

Ви викликаєте цю функцію, просто ввівши її ім’я та дужки.

```
>>> do_nothing()
>>>
```

Аргументи та параметри

Значення, які ви передаєте функції під час її виклику, відомі як *аргументи*. Коли ви викликаєте функцію з аргументами, значення цих аргументів копіюються у відповідні параметри всередині функції.

```
def total(a, b)  # function accepting prameters
    result = a + b
    print("Sum of ", a, " and ", b, " = ", result)

a = int(input("Enter the first number : ")
b = int(input("Enter the second number : ")
total(a, b)  # function call with two arguments

OUTPUT
Enter the first number : 10
Enter the second number : 20
Sum of 10 and 20 = 30
```

Позиційні аргументи

Python обробляє аргументи функцій дуже гнучко, якщо порівнювати з багатьма мовами. Найбільш відомі типи аргументів - позиційні аргументи, значення яких копіюються у відповідні параметри за порядком.

Ця функція будує словник з його позиційних вхідних аргументів і повертає його:

```
>>> def menu(wine, entree, dessert):
... return {'wine': wine, 'entree': entree, 'dessert': dessert}
...
>>> menu('chardonnay', 'chicken', 'cake')
{'wine': 'chardonnay', 'entree': 'chicken', 'dessert': 'cake'}
```

Хоча це дуже поширене явище, мінус позиційних аргументів полягає в тому, що вам потрібно пам’ятати значення кожної позиції. Якби ми забули і викликали menu() з вином як останнім аргументом замість першого, страва була б зовсім іншою:

```
>>> menu('beef', 'bagel', 'bordeaux')
{'wine': 'beef', 'entree': 'bagel', 'dessert': 'bordeaux'}
```

Аргументи ключових слів

Щоб уникнути плутанини позиційних аргументів, можна вказати аргументи за іменами відповідних параметрів навіть у порядку, відмінному від їх визначення у функції:

```
>>> menu(entree='beef', dessert='bagel', wine='bordeaux')
{'wine': 'bordeaux', 'entree': 'beef', 'dessert': 'bagel'}
```

Ви можете змішувати позиційні та ключові аргументи. Давайте спочатку вкажемо вино, але використовуємо ключові аргументи для страви та десерту:

```
>>> menu('frontenac', dessert='flan', entree='fish')
{'wine': 'frontenac', 'entree': 'fish', 'dessert': 'flan'}
```

Якщо ви викликаєте функцію з позиційними та ключовими аргументами, позиційні аргументи мають бути на першому місці.

Вказування значення параметрів за замовчуванням

Ви можете вказати значення за замовчуванням для параметрів. Це значення використовується, якщо абонент не надає відповідний аргумент.

```
>>> def menu(wine, entree, dessert='pudding'):
...       return {'wine': wine, 'entree': entree, 'dessert':
dessert}
```

Цього разу спробуйте викликати menu() без аргументу десерту:

```
>>> menu('chardonnay', 'chicken')
{'wine': 'chardonnay', 'entree': 'chicken', 'dessert': 'pud-
ding'}
```

Якщо ви надаєте аргумент, він використовується замість типового:

```
>>> menu('dunkelfelder', 'duck', 'doughnut')
{'wine': 'dunkelfelder', 'entree': 'duck', 'dessert': 'dough-
nut'}
```

Розгорнути/зібрати позиційні аргументи за допомогою *

Якщо ви програмували на C або C ++, можна припустити, що зірочка (*) у програмі Python має щось спільне з покажчиком. Ні, у Python немає вказівників. При використанні всередині функції з параметром зірочка *групує змінну кількість позиційних аргументів в єдиний кортеж* значень параметрів. У наступному прикладі args - це кортеж параметрів, який є результатом нульових або більше аргументів, які були передані функції print_args ():

```
>>> def print_args(*args):
...     print('Positional tuple:', args)
...
```

Якщо ви викликаєте функцію без аргументів, ви не отримаєте нічого в *args:

```
>>> print_args()
Positional tuple: ()
```

Все, що ви задасте, буде надруковано як кортеж args:

```
>>> print_args(3, 2, 1, 'wait!', 'uh...')
Positional tuple: (3, 2, 1, 'wait!', 'uh...')
```

Це корисно для написання таких функцій, як print (), які приймають змінну кількість аргументів. Якщо ваша функція також потребує позиційних аргументів, поставте їх на перше місце; *args йде в кінці і захоплює все інше.

Розгорнути/зібрати аргументи ключових слів за допомогою **

Ви можете використовувати дві зірочки (**) для групування аргументів ключових слів у словнику, де назви аргументів - це ключі, а їх значення - відповідні значення словника. Наступний приклад визначає функцію print_kwargs () для друку аргументів ключових слів:

```
>>> def print_kwargs(**kwargs):
... print('Keyword arguments:', kwargs)
...
```

Тепер спробуйте викликати її з деякими ключовими аргументами:

```
>>> print_kwargs()
Keyword arguments: {}
>>>    print_kwargs(wine='merlot',    entree='mutton',    des-
sert='macaroon')
Keyword arguments: {'dessert': 'macaroon', 'wine': 'merlot',
```

```
'entree': 'mutton'}
```

Усередині функції kwargs - це параметр словник.

Keyword-Only аргументи

Можна передати аргумент ключового слова з такою ж назвою, що і позиційний параметр, ймовірно, це не призведе до того, що вам потрібно. Python 3 дозволяє вказувати лише ключові аргументи. Як випливає з назви, вони повинні надаватися як name=value, а не позиційно як значення. Єдиний * у визначенні функції означає, що наступні параметри start та end повинні бути надані як іменовані аргументи, якщо ми не хочемо їх значень за замовчуванням:

```
>>> def print_data(data, *, start=0, end=100):
...     for value in (data[start:end]):
...         print(value)
...
>>> data = ['a', 'b', 'c', 'd', 'e', 'f']
>>> print_data(data)
a
b
c
d
e
f
>>> print_data(data, start=4)
e
f
>>> print_data(data, end=2)
a
b
```

Змінні та незмінні аргументи

Пам'ятайте, що якщо ви призначили один і той самий список двом змінним, ви могли б змінити його за допомогою будь-якої з них. І що ви це зробити не можете, якщо обидві змінні посилаються на щось на кшталт цілого чи рядка. Це є тому, що список був змінним, а ціле число та рядок незмінними.

Ви повинні стежити за такою ж поведінкою при передачі аргументів функціям. Якщо аргумент змінюється, його значення можна змінити всередині функції за допомогою відповідного параметра:

```
>>> outside = ['one', 'fine', 'day']
>>> def mangle(arg):
...     arg[1] = 'terrible!'
...
>>> outside
['one', 'fine', 'day']
>>> mangle(outside)
>>> outside
['one', 'terrible!', 'day']
```

Docstrings

Чіткість має значення, справді каже дзен Python. До визначення функції можна додати документацію, включивши рядок на початку тіла функції. Це рядок docstring:

```
def print_if_true(thing, check):
    '''
    Prints the first argument if a second argument is true.
    The operation is:
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument.
    '''
    if check:
        print(thing)
```

Всі функції в Python є First-Class Citizens

Згадаймо мантру Python: «все є об'єктом». Це включає в себе числа, рядки, кортежі, списки, словники та функції. Функції є first-class citizens у Python. Ви можете призначити їх змінним, використовувати їх як аргументи для інших функцій та повертати їх із функцій. Це дає вам можливість робити деякі речі на Python, які важко виконати багатьма іншими мовами.

Щоб перевірити це, давайте визначимо просту функцію під назвою answer (), яка не має жодних аргументів; вона просто друкує номер 42:

```
>>> def answer():
...     print(42)
```

Давайте визначимо іншу функцію під назвою run_something. У неї є один аргумент, який називається func, функція для запуску:

```
>>> def run_something(func):
...     func()
```

Якщо ми передаємо answer в run_something (), ми використовуємо функцію як дані, так само як і з будь-яким іншим типом даних:

```
>>> run_something(answer)
42
```

Зверніть увагу, що ми передали answer, а не answer(). У Python ці дужки означають виклик цієї функції. Без дужок, Python просто розглядає функцію як будь-який інший об'єкт. Це тому, що, як і все інше в Python, функція - це об'єкт:

```
>>> type(run_something)
<class 'function'>
```

Внутрішні функції

Ви можете визначити функцію в іншій функції:

```
>>> def outer(a, b):
...     def inner(c, d):
...         return c + d
...     return inner(a, b)
...
>>>
>>> outer(4, 7)
11
```

Внутрішня функція може бути корисною під час виконання складного завдання кілька разів у межах іншої функції, щоб уникнути циклів або дублювання коду. Для прикладу рядка ця внутрішня функція додає текст до свого аргументу:

```
>>> def knights(saying):
```

```
...     def inner(quote):
...         return "We are the knights who say: '%s'" % quote
...     return inner(saying)
...
>>> knights('Ni!')
"We are the knights who say: 'Ni!'"
```

Закриття (Closures)

Внутрішня функція може виконувати роль закриття. Це функція, яка динамічно генерується іншою функцією і може як змінювати, так і запам'ятовувати значення змінних, створених поза функцією.

Наступний приклад ґрунтується на попередньому прикладі knight(). Давайте назвемо нову функцію knight2 (), і перетворимо функцію inner() на закриття під назвою inner2(). Ось відмінності:

• inner2 () використовує зовнішній параметр saying безпосередньо, а не отримує його як аргумент.

• knights2 () повертає ім'я функції inner2 замість того, щоб викликати її:

```
>>> def knights2(saying):
...     def inner2():
...        return "We are the knights who say: '%s'" % saying
...     return inner2
...
```

Функція inner2() знає значення переданого saying і запам'ятовує його. Рядок return inner2 повертає цю спеціалізовану копію функції inner2 (але не викликає її). Це своєрідне закриття: динамічно створена функція, яка запам'ятовує, звідки вона прийшла.

Давайте двічі викличемо knights2() з різними аргументами:

```
>>> a = knights2('Duck')
>>> b = knights2('Hasenpfeffer')
```

Гаразд, а які типи у a і b?

```
>>> type(a)
<class 'function'>
>>> type(b)
<class 'function'>
```

Вони є функціями, але вони також є закриттями:

```
>>> a
<function knights2.<locals>.inner2 at 0x10193e158>
>>> b
<function knights2.<locals>.inner2 at 0x10193e1e0>
```

Якщо ми їх викликаємо, вони пам’ятають saying, який використовувався, коли їх створювали knights2:

```
>>> a()
"We are the knights who say: 'Duck'"
>>> b()
"We are the knights who say: 'Hasenpfeffer'"
```

Анонімні функції: lambda

Лямбда-функція Python - це анонімна функція, виражена як єдиний оператор. Ви можете використовувати його замість звичайної крихітної функції.

Щоб проілюструвати це, давайте спочатку зробимо приклад, який використовує звичайні функції. Для початку давайте визначимо функцію edit_story(). ЇЇ аргументи такі:

• words — список слів

• func — функція, що застосовується до кожного слова в словах.

```
>>> def edit_story(words, func):
...     for word in words:
...         print(func(word))
```

Тепер нам потрібен список слів і функція для застосування до кожного слова:

```
>>> stairs = ['thud', 'meow', 'thud', 'hiss']
```

Нехай функція кожне слово буде писати з великої літери та додаватиме знак оклику:

```
>>> def enliven(word):
...     return word.capitalize() + '!'
```

Тепер викличемо функцію edit_story():

```
>>> edit_story(stairs, enliven)
Thud!
Meow!
Thud!
Hiss!
```

Нарешті ми доходимо до лямбди. Функція enliven () була настільки короткою, що ми могли б замінити її на лямбду:

```
>>> edit_story(stairs, lambda word: word.capitalize() + '!')
Thud!
Meow!
Thud!
Hiss!
```

Lambda містить нуль або більше аргументів, розділених комами, за якими йде двокрапка (:), а потім-визначення функції. Ви не використовуєте дужки з lambda, як це було б під час виклику функції, створеної за допомогою def.

Часто використання реальних функцій, таких як enliven(), набагато зрозуміліше, ніж використання лямбд. Лямбди переважно корисні для випадків, коли в іншому випадку вам потрібно було б визначити багато крихітних функцій і запам’ятати, як ви їх усі назвали. Зокрема, ви можете використовувати лямбди в графічних інтерфейсах користувача для визначення функцій зворотного виклику.

Модулі та конструкція import

Ми будемо створювати та використовувати код Python у кількох файлах. Модуль - це просто файл будь-якого коду Python. Вам не потрібно робити нічого особливого, будь-який код Python може бути використаний як модуль іншими.

Ми посилаємося на код інших модулів за допомогою оператора import Python. Це робить код та змінні в імпортованому модулі доступними для вашої програми.

Імпортування модуля

Найпростіший спосіб використання оператора import –

`import module`,

де module - це ім’я іншого файлу Python без розширення .py.

Скажімо, ви та кілька інших бажаєте чогось швидкого на обід, але не хочете довгих обговорень, і ви завжди вибираєте те, що хоче найгучніша людина. Нехай комп'ютер вирішує! Давайте напишемо модуль з однією функцією, яка повертає випадковий вибір фаст-фуду, та основну програму, яка викликає його та друкує вибір.

Модуль (fast.py) показаний у Лістингу 1.

Лістинг 1. Файл fast.py

```
from random import choice
places = ['McDonalds", "KFC", "Burger King", "Taco Bell",
"Wendys", "Arbys", "Pizza Hut"]
def pick():
    """Return random fast food place"""
    return choice(places)
```

Лістинг 2 показує основну програму, яка його імпортує (назвемо її lunch.py).

Лістинг 2. Файл lunch.py

```
import fast
place = fast.pick()
print("Let's go to", place)
```

Розмістимо ці два файли в одному каталозі та запустимо lunch.py як основну програму, вона отримає доступ до модуля `fast` та запустить функцію pick(), яка поверне випадковий результат зі списку рядків:

```
$ python lunch.py
Let's go to Burger King
$ python lunch.py
Let's go to Pizza Hut
$ python lunch.py
Let's go to Arbys
```

Ми використовували імпорт у двох різних місцях:

• Основна програма lunch.py імпортувала наш новий модуль `fast`.

• Файл модуля fast.py імпортував функцію `choice` зі стандартного бібліотечного модуля Python під назвою `random`.

Ми також використовували імпорт двома різними способами в нашій головній програмі та нашому модулі:

• У першому випадку ми імпортували весь модуль `fast`, але його потрібно було використовувати як префікс для pick(). Після цього оператора імпорту все у файлі fast.py доступне для головної програми. Ставлячи назву модуля перед функцією, ми уникаємо будь-яких неприємних конфліктів імен. У іншому модулі може бути функція pick(), і ми б не викликали її помилково.

• У другому випадку ми знаходимось у модулі і знаємо, що тут немає нічого іншого з назвою choice, тому ми імпортували функцію choice() безпосередньо з модуля random.

Ми могли б написати fast.py, як показано в Лістингу 3, імпортуючи random у функції pick() замість верхньої частини файлу.

Лістинг 3. Файл fast2.py

```
    places = ['McDonalds", "KFC", "Burger King", "Taco Bell",
"Wendys", "Arbys", "Pizza Hut"]
    def pick():
        import random
        return random.choice(places)
```

Імпортування модуля з іншою назвою

У нашій головній програмі lunch.py ми називали імпорт `fast`. Але що, якщо:

• Десь є інший модуль з назвою `fast`?

• Хочете використовувати більш мнемонічну назву?

У цих випадках можна імпортувати за допомогою псевдоніма (alias), як показано в Лістингу 4. Скористаємося псевдонімом f.

Лістинг 4. Файл fast3.py

```
import fast as f
place = f.pick()
print("Let's go to", place)
```

Імпортуйте з модуля лише те, що вам потрібно

Ви можете імпортувати цілий модуль або лише його частини. Ви щойно побачили останнє: нам потрібна була лише функція choice() з модуля `random`.

Як і сам модуль, ви можете використовувати псевдонім для кожного імпортованого елементу.

Повторимо lunch.py ще кілька разів. Спочатку імпортуйте pick() з модуля `fast` з його оригінальною назвою (Лістинг 5).

Лістинг 5. Файл fast4.py

```
from fast import pick
place = pick()
print("Let's go to", place)
```

Тепер імпортуйте його як who_cares (Лістинг 6).

Лістинг 6. Файл fast5.py

```
from fast import pick as who_cares
place = who_cares()
print("Let's go to", place)
```

Модуль sys

Модуль sys забезпечує доступ до деяких змінних і функцій, які взаємодіють з інтерпретатором Python.

sys.argv - список аргументів командного рядка, що передаються сценарію Python. sys.argv [0] є іменем скрипта.

Наступна програма демонструє доступ до аргументів командного рядка. Вона починається з повідомлення про кількість аргументів командного рядка, наданих програмі, та ім’я вихідного файлу, який виконується. Потім вона продовжує роботу і відображає аргументи, які з’являються після назви вихідного файлу, якщо такі значення були надані. В іншому випадку відображається пові-

домлення, яке вказує на те, що не було аргументів командного рядка, окрім виконуваного файлу .py.

```
import sys

print("The program has", len(sys.argv), \
      "command line argument(s).")
print("The name of the .py file is", sys.argv[0])
if len(sys.argv) > 1:
    print("The remaining arguments are:")
    for i in range(1, len(sys.argv)):
        print(" ", sys.argv[i])
else:
    print("No additional arguments were provided.")
```

Пакети (Packages)

Ми перейшли від єдиних рядків коду, до багаторядкових функцій, до окремих програм, до кількох модулів в одному каталозі. Якщо у вас немає багато модулів, той же каталог працює добре.

Щоб дозволити додаткам Python ще більше масштабуватися, ви можете організувати модулі у файли та ієрархії модулів, які називаються пакетами. Пакет - це лише підкаталог, що містить файли .py. І ви можете заглибитися на кілька рівнів з каталогами всередині них.

Ми щойно написали модуль, який вибирає місце для швидкого харчування. Давайте додамо подібний модуль, щоб роздавати життєві поради. Ми зробимо одну нову основну програму під назвою question.py у нашому поточному каталозі. Тепер зробіть підкаталог з назвою choices і помістіть у нього два модулі —fast.py та advice.py. Кожен модуль має функцію, яка повертає рядок.

Основна програма (questions.py) має імпорт та додатковий рядок (Лістинг 7).

Лістинг 7. Файл questions.py

```
from choices import fast, advice

print("Let's go to", fast.pick())
print("Should we take out?", advice.give())
```

Це `from choices` змушує Python шукати каталог з назвою `sources`, починаючи з вашого поточного каталогу. Усередині `sources` він шукає файли fast.py та advice.py.

Перший модуль (choices/fast.py) - це той самий код, що і раніше, щойно переміщений у каталог choices (Лістинг 8).

Лістинг 8. choices/fast.py

```
from random import choice

places = ["McDonalds", "KFC", "Burger King", "Taco Bell",
"Wendys", "Arbys", "Pizza Hut"]
def pick():
    """Return random fast food place"""
```

```
    return choice(places)
```

Другий модуль (choices/advice.py) є новим, але він багато в чому працює як його родич fast-food (Лістинг 9).

Лістинг 9. choices/advice.py

```
from random import choice

answers = ["Yes!", "No!", "Reply hazy", "Sorry, what?"]
def give():
    """Return random advice"""
    return choice(answers)
```

ПРИМІТКА

Якщо ваша версія Python раніша 3.3, вам знадобиться ще одна річ у підкаталозі choices, щоб зробити його пакетом Python: файл з назвою __init__.py. Це може бути порожній файл, але Python до 3.3 має потребу в ньому, щоб розглядати каталог, що містить його, як пакет.

Запустіть основну програму questions.py (з вашого поточного каталогу, а не в `choices`), щоб побачити, що відбувається:

```
$ python questions.py
Let's go to KFC
Should we take out? Yes!
$ python questions.py
Let's go to Wendys
Should we take out? Reply hazy
$ python questions.py
Let's go to McDonalds
Should we take out? Reply hazy
```

Винятки (Exceptions)

Винятки – це помилки часу виконання (ділення на нуль, не той тип введених даних, тощо). У деяких мовах помилки позначаються спеціальними поверненими значеннями функцій. Коли справи йдуть в програмі не так, Python використовує винятки: код, який виконується, коли виникає пов'язана з винятком помилка.

Деякі з винятків ви вже бачили, наприклад, зверталися до списку або кортежу з положенням поза діапазоном або до словника з неіснуючим ключем. Коли ви запускаєте код, який може вийти з ладу за певних обставин, вам також потрібні відповідні обробники винятків, щоб перехопити будь-які потенційні помилки.

Радимо додати обробку винятків будь-де, де може виникнути виняток, щоб користувач знав, що відбувається. Можливо, вам не вдасться виправити проблему, але принаймні ви можете відзначити обставини та витончено закрити програму. Якщо виняток трапляється в якійсь функції і не фіксується там, він як бульбашка спливає наверх, поки він не буде спійманий відповідним обробником у якійсь викликовій функції. Якщо ви не надаєте власний обробник винятків, Python надрукує повідомлення про помилку та деяку інформацію про те,

де сталася помилка, а потім припинить роботу програми, як показано в наступному фрагменті:

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> short_list[position]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Обробка помилок за допомогою try і except

Замість того, щоб залишати речі на волю випадку, скористайтеся try щоб обернути код і except щоб забезпечити обробку помилок:

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> try:
...     short_list[position]
... except:
...   print('Need a position between 0 and', len(short_list)-
1, ' but got', position)
...
Need a position between 0 and 2 but got 5
```

Запускається код всередині блоку try. Якщо є помилка, виникає виняток, і код всередині блоку except запускається. Якщо помилок немає, блок except пропускається.

Використання протого except без аргументів, як ми зробили тут, є загальним для будь-якого типу винятків. Якщо може виникнути кілька типів винятків, краще надати окремий обробник винятків для кожного.

Іноді вам потрібні відомості про винятки поза типом. Ви отримуєте повний об'єкт виключення в змінній `name`, якщо використовуєте форму:

```
except exceptiontype as name
```

У наведеному нижче прикладі спочатку шукається IndexError, оскільки це тип винятку, який виникає, коли ви задаєте для послідовності неправильний індекс. Програма зберігає виняток IndexError у змінній err та будь-який інший виняток у змінній other. У прикладі друкується все, що зберігається в other, щоб показати, що ви отримуєте в цьому об’єкті:

```
>>> short_list = [1, 2, 3]
>>> while True:
...     value = input('Position [q to quit]? ')
...     if value == 'q':
...         break
...     try:
...         position = int(value)
...         print(short_list[position])
...     except IndexError as err:
...         print('Bad index:', position)
...     except Exception as other:
...         print('Something else broke:', other)
...
Position [q to quit]? 1
2
Position [q to quit]? 0
```

```
1
Position [q to quit]? 2
3
Position [q to quit]? 3
Bad index: 3
Position [q to quit]? 2
3
Position [q to quit]? two
Something else broke: invalid literal for int() with base 10: 'two'
Position [q to quit]? Q
```

Введення позиції 3 викликало IndexError, як і очікувалося. Введення two не влаштовує функцію int (), і тут спрацьовує другий except.

Робота з файлами

Відкриття файлу

Ви використовуєте функцію open у Python, щоб відкрити файл. Функція open створює об'єкт файлу та пов'язує його з файлом на диску. Ось загальний формат використання функції open:

```
file_variable = open(filename, mode)
```

У загальному форматі:

• file_variable - це ім’я змінної, яка посилатиметься на об’єкт файлу.

• filename - це рядок, що визначає ім’я файлу.

• mode - це рядок, що визначає режим (читання, запис тощо), у якому буде відкрито файл.

Перша буква режиму позначає операцію:

• **r** означає читати.

• **w** означає писати. Якщо файл не існує, його створюють. Якщо файл існує, його переписують.

• **x** означає запис, але тільки якщо файл ще не існує.

• **a** означає додавання (запис після закінчення), якщо файл існує.

Друга буква режиму - це тип файлу:

• **t** (або нічого) означає текст.

• **b** означає двійковий.

Після відкриття файлу ви викликаєте функції для читання або запису даних; це буде показано у наступних прикладах.

Нарешті, вам потрібно закрити файл, щоб переконатися, що будь-які записи завершені, а пам'ять звільнена. Пізніше ви побачите, як використовувати with для автоматизації закриття.

Ця програма відкриває файл під назвою oops.txt і закриває його, нічого не записуючи. Це створить порожній файл:

```
>>> fout = open('oops.txt', 'wt')
>>> fout.close()
```

Розглянемо наступний фрагмент коду:

```
# Read the file name from the user
fname = input("Enter the file name: ")
# Attempt to open the file
try:
```

```
        inf = open(fname, "r")
    except FileNotFoundError:
        # Display an error message and quit if the file was not
        # opened successfully
        print("'%s' could not be opened. Quitting...")
        quit()
    . . .
```

Запис в текстовий файл за допомогою print()

Давайте відтворимо oops.text, але тепер запишемо рядок і закриємо його:

```
>>> fout = open('oops.txt', 'wt')
>>> print('Oops, I created a file.', file=fout)
>>> fout.close()
```

Для print ми використовували аргумент file. Без цього print записує на стандартний вивід, який є вашим терміналом.

Запис в текстовий файл за допомогою write()

```
>>> poem = '''There was a young lady named Bright,
... Whose speed was far faster than light;
... She started one day
... In a relative way,
... And returned on the previous night.'''
>>> len(poem)
150
>>> fout = open('relativity', 'wt')
>>> fout.write(poem)
150
>>> fout.close()
```

Функція write повертає кількість записаних байтів. Вона не додає пробілів або нових рядків, як це робить print. Як і раніше, ви також можете використати print щоб записати багаторядковий рядок у текстовий файл:

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout)
>>> fout.close()
```

Якщо файл relativity є для нас цінним, подивимось, чи дійсно використання режиму x захищає нас від перезапису:

```
>>> fout = open('relativity', 'xt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'relativity'
```

Ви можете застосувати обробник винятків:

```
>>> try:
...     fout = open('relativity', 'xt')
...     fout.write('stomp stomp stomp')
... except FileExistsError:
...     print('relativity already exists!. That was a close
one.')
...
relativity already exists!. That was a close one.
```

Читання текстового файлу за допомогою read(), readline() або readlines()

Ви можете викликати read() без аргументів, щоб прочитати весь файл одночасно, як показано в наведеному нижче прикладі (будьте обережні, роблячи це з великими файлами; гігабайтний файл споживає гігабайт пам'яті):

```
>>> fin = open('relativity', 'rt' )
>>> poem = fin.read()
>>> fin.close()
>>> len(poem)
150
```

Ви можете вказати максимальну кількість символів, щоб обмежити, скільки read() повертає одночасно. Давайте прочитаємо 100 символів одночасно і додамо кожен шматок до рядка вірша, щоб відновити оригінал:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> chunk = 100
>>> while True:
...     fragment = fin.read(chunk)
...     if not fragment:
...         break
...     poem += fragment
...
>>> fin.close()
>>> len(poem)
150
```

Після того, як ви прочитаєте до кінця, подальші виклики read () повернуть порожній рядок (''), який вважається False в `if not fragment`. Це вириває нас з циклу while True.

Ви також можете прочитати файл по рядкам за допомогою readline(). У цьому наступному прикладі ми додаємо кожен рядок до рядка вірша, щоб відновити оригінал:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> while True:
... line = fin.readline()
... if not line:
... break
... poem += line
...
>>> fin.close()
>>> len(poem)
150
```

Для текстового файлу навіть порожній рядок має довжину одиниця (символ нового рядка) і оцінюється як True. Коли файл буде прочитано, readline() (як і read()) також повертає порожній рядок, який також оцінюється як False.

Найпростіший спосіб читання текстового файлу - це використання ітератора. Він повертає один рядок за раз. Це схоже на попередній приклад, але з меншим кодом:

```
>>> poem = ''
```

```
>>> fin = open('relativity', 'rt' )
>>> for line in fin:
...     poem += line
...
>>> fin.close()
>>> len(poem)
150
```

Усі попередні приклади врешті-решт побудували вірш по одному рядку. Виклик readlines () читає рядок за раз і повертає список однорядкових рядків:

```
>>> fin = open('relativity', 'rt' )
>>> lines = fin.readlines()
>>> fin.close()
>>> print(len(lines), 'lines read')
5 lines read
>>> for line in lines:
... print(line, end='')
...
There was a young lady named Bright,
Whose speed was far faster than light;
She started one day
In a relative way,
And returned on the previous night.>>>
```

Автоматичне закриття файлів за допомогою with

Якщо ви забули закрити відкритий файл, Python закриє його після того, як на нього більше не буде посилань. Це означає, що якщо ви відкриваєте файл у функції та не закриваєте його явно, він закриється автоматично після завершення функції. Але ви могли відкрити файл у довгостроковій функції або в головному розділі програми. Файл слід закрити, щоб змусити завершити всі залишкові записи.

У Python є контекстні менеджери для очищення таких речей, як відкриті файли. Ви використовуєте форму `with expression as variable:`

```
>>> with open('relativity', 'wt') as fout:
...     fout.write(poem)
...
```

Це воно. Після завершення блоку коду в контекстному менеджері (в даному випадку один рядок) (зазвичай або через виняток) файл автоматично закривається.

2.4 Питання для самоконтролю

1. Як здійснюється запис даних в файл у мові Python?
2. Як здійснюється читання даних з файлу у мові Python?
3. Як здійснюється копіювання файлу?
4. Які бувають режими роботи з файлами?

ЛІТЕРАТУРА

1. Яковенко А.В. Основи програмування. Python. Частина 1: підручник для студ. спеціальності 122 "Комп’ютерні науки". – Київ : КПІ ім. Ігоря Сікорського, 2018. – 195 с.
2. Програмування на мові Python. Методичні вказівки до виконання лабораторних робіт студентами денної та заочної форми навчання спеціальностей 123 "Комп’ютерна інженерія", 125 "Кібербезпека" / Укл.: Є.В. Мелешко – Кропивницький: ЦНТУ, 2017. – 58 с.
3. Програмування числових методів мовою Python : підруч. / А. В. Анісімов, А. Ю. Дорошенко, С. Д. Погорілий, Я. Ю. Дорогий ; за ред. А. В. Анісімова. – К. : Видавничо-поліграфічний центр "Київський університет", 2014. – 640 с.
4. Костюченко А.О. Основи програмування мовою Python: навчальний посібник. – Чернігів: ФОП Баликіна С.М., 2020. 180 с.
5. Креневич А.П. Python у прикладах і задачах. Частина 1. Структурне програмування. Навчальний посібник із дисципліни "Інформатика та програмування" – К.: ВПЦ "Київський Університет", 2017. – 206 с.
6. Лутц М. Изучаем Python, том 1, 5-е изд. — СПб.: ООО “Диалектика”, 2019. — 832 с.
7. Лутц М. Изучаем Python, том 2, 5-е изд. — СПб.: ООО “Диалектика”, 2020. — 720 с.
8. Joakim Sundnes Introduction to Scientific Programming with Python. – Springer, 2020. 148 p.
9. Bill Lubanovic Introducing Python. – O’Reilly Media, 2020. 597 p.