



ДОСЛІДНИЦЬКЕ КОДУВАННЯ ТА ІНСТРУМЕНТИ НАЛАГОДЖЕННЯ PYTHON-КОДУ

Питання 7.4

Дослідницьке кодування

- IPython постачає багато магічних функцій (magic functions)
 - Наприклад, можна отримати список усіх записаних у сесії команд за допомогою %hist:
 - Можна писати код в консолі, а потім копіювати його блоки в файл.
 - Або навпаки, з файлу в консоль за допомогою магічної функції %paste, яка копіює разом з відступами.
 - Можна використовувати для перевірки працездатності невеликих шматочків коду.
 - Ще можна запускати модулі Python за допомогою %run <module_name>.
 - Аналог python <program name>.
 - Перевага: всі змінні та функції з програми збережено, тому простіше переглянути їх значення після завершення програми.

```
In [1]: LEFT = (-1, 0)
In [2]: RIGHT = (1, 0)
In [3]: UP = (0, -1)
In [4]: DOWN = (0, 1)
In [5]: %hist
LEFT = (-1, 0)
RIGHT = (1, 0)
UP = (0, -1)
DOWN = (0, 1)
%hist
```

Корисні команди IPython, які підтримує інтроспекція

| Команда | Опис |
|----------------|---|
| ?name | Відображає базову інформацію про “name” |
| ?nam* | Виводить перелік усіх об’єктів, які починаються з nam |
| Tab | автозаповнення |
| pwd | аналог Unix-команди (print working directory) |
| cd name | аналог Unix-команди (change working directory) |
| ls | аналог Unix-команди (list working directory) |
| %run name.py | запускає Python-модуль на виконання |
| %paste | виконує код з буферу обміну |
| %debug name.py | запускає налагоджувальник для Python-модуля |
| %reset | очищає простір імен поточної IPython-сесії |
| %env | виводить перелік змінних (параметрів) середовища |

Огляд файлів та папок

- Можна використовувати Unix-команди, наприклад, `ls`, `cd`, `pwd` в IPython.
 - Допомагають ідентифікувати неправильні назви та шляхи до файлів.
 - Також сприяють розумінню того, які Python-модулі можна імпортувати.

```
In [11]: cd C:\Users\User\Desktop\  
...:  
C:\Users\User\Desktop  
  
In [12]: ls *.py  
Том в устройстве C имеет метку System  
Серийный номер тома: DE61-0543  
  
Содержимое папки C:\Users\User\Desktop  
  
15.11.2017  14:49          2 630 aa_tree.py  
15.11.2017  13:39          2 372 avl_tree.py  
          2 файлов          5 002 байт  
          0 папок  50 034 958 336 байт свободно
```

Огляд просторів імен (Namespaces)

- Інтроспекція використовується для заглядання всередину Python-об'єктів.
 - Питання “що всередині об'єкту?” тісно пов'язане з поняттям просторів імен (namespaces).
 - Об'єкти прив'язані до назв, наприклад, при оголошенні змінної:
 - In [8]: LEFT = (-1, 0)
 - Назва використовується як ключ словника, який Python використовує всередині себе, щоб знайти кортеж при зверненні до LEFT.
 - Такий словник називають *простором імен (namespace)*.
- `dir()` повертає список назв об'єктів у просторі імен в алфавітному порядку, починаючи з великих літер, а далі – з імен, які починаються символами підкреслення, потім – іменами, для яких перші літери маленькі.
 - IPython створює In автоматично, містить список усіх введених на даний момент команд.
 - Out також створюється IPython – це словник усіх виводів, які IPython відправляє на стандартне виведення.
 - `__builtin__` та `__builtins__` відносяться до модуля зі стандартними функціями Python, на зразок `print()`. Автоматично імпортуються при запуску Python.
 - `__doc__` - текст документації для поточного простору імен.
 - `__name__` - назва поточного простору імен.
 - `exit()` та `quit()` є функціями для виходу з IPython.

```
In [9]: dir()
['DOWN', 'In', 'LEFT', 'Out', 'RIGHT', 'UP', '_', '__',
 '__builtin__', '__builtins__', '__doc__',
 '__loader__', '__name__', '__package__', '__spec__', '_dh',
 '_i', '_i1', '_i2', '_i3', '_i4', '_i5', '_i6', '_i7',
 '_i8', '_ih', '_ii', '_iii', '_oh', '_sh', 'exit',
 'get_ipython', 'quit']
```

Огляд просторів імен об'єктів

- За допомогою інтроспекції також можна знайти власні модулі, які імпортуються в простір імен IPython.
 - Ще не відомо, що містить кожен модуль.
 - Можна використати `dir` для модуля та розглянути простір імен цих модулів:

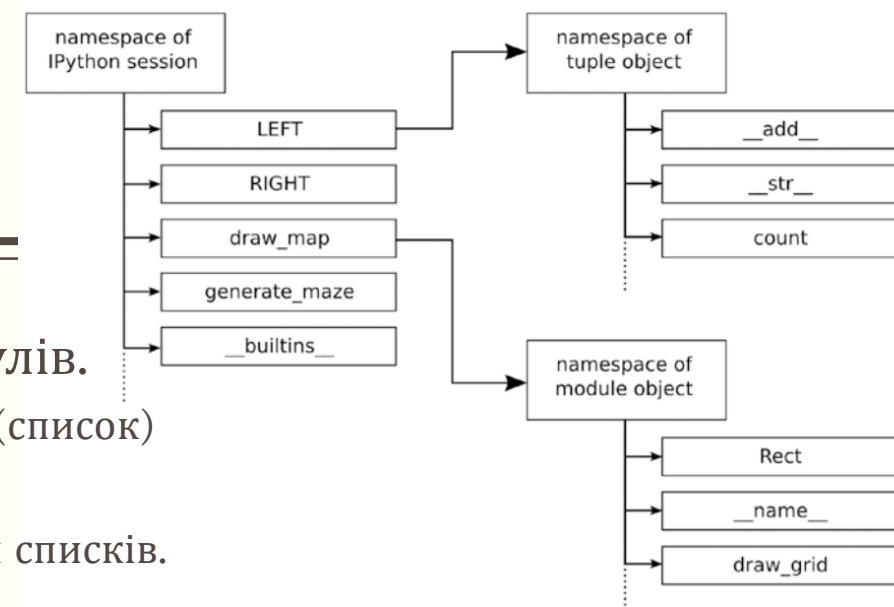
```
In [14]: dir(draw_maze)
```

This results in:

```
['Rect', 'SIZE', 'TILE_POSITIONS', 'Surface', '__builtins__',  
'__cached__', '__doc__', '__file__', '__loader__',  
'__name__', '__package__', '__spec__', 'create_maze',  
'debug_print', 'draw_grid', 'get_tile_rect', 'image',  
'load_tiles', 'parse_grid']
```

Огляд просторів імен об'єктів

- Команда `dir` застосовується як до об'єктів, так і до модулів.
 - Наприклад, проінспектуємо атрибути об'єкту `TILE_POSITIONS` (список)
 - `In [15]: dir(draw_maze.TILE_POSITIONS)`
 - Побачимо довгий список елементів, в кінці якого будуть методи списків.
- Часто корисними є атрибути:
 - `_file_` містить фізичне розташування модуля.
 - `__name__` допомагає дізнатись назви функцій, класів та модулів, з якими йтиме взаємодія (за допомогою `import .. as` або функцій як параметрів інших функцій).
- *Магічні методи (magic methods)* відображають параметри в оператори або стандартні функції:
 - Наприклад, атрибут `__add__` з простору імен визначає поведінку оператора `+` при роботі з об'єктом.
 - Атрибут `__getitem__` визначає використання індексації та квадратних дужок.
 - <https://docs.python.org/3/reference/datamodel.html>



Огляд атрибутів у Python-програмі

- Вміст простору імен також називають *атрибутами*.
 - `dir` дає назви цих атрибутів як список рядків.
 - Підчас виконання програми назви атрибутів не завжди відомі.
 - Корисними будуть інтроспекційні функції `hasattr()` та `getattr()`.
- Потрібно отримати доступ до об'єкта `draw_maze.SIZE` у програмі.
 - За допомогою `getattr()` повертається відповідний об'єкт:
In [16]: `size = getattr(draw_maze, 'SIZE')`
 - Виклик `hasattr(x, name)` перевіряє існування обраного об'єкту (True або False).
 - Приклад: зчитування з конфігураційного файлу списку модулів або функцій та динамічний доступ до них у програмі.
 - `getattr` та `hasattr` інколи використовують при налагодженні, проте частіше при динамічному додаванні модулів і функцій в програму (поширена потреба в Django).

Альтернативи `dir` в IPython

- В IPython можна швидко переглянути простори імен, тому що назви змінних, функцій та модулів автодоповнюються натисненням Tab.
 - Можна шукати простори імен за допомогою підстановочних символів (wildcards, *):
- In [17]: `?dra*`
 - Якщо простір імен у вигляді списку не обов'язковий, цей виклик дещо ефективніший за `dir`.
- Інформація від `dir` допомагає обирати, що імпортувати.
 - У прикладі потрібні `pygame.image` та стандартні модулі `random` і `sys`:
 - `from load_tiles import load_tiles`
 - `from generate_maze import create_maze`
 - `from draw_maze import draw_grid, parse_grid`
 - `from pygame import image`
 - `import random`
 - `import sys`

Механізм роботи просторів імен

- У мові Python присутні кілька команд, які напряду змінюють простори імен:
 - Присвоєння змінній значення додає її назву до простору імен або заміняють існуючу.
 - Інструкція `del` видаляє назву з простору імен.
 - Оператор `def` додає функцію в простір імен. Функція є об'єктом з власним простором імен; підчас її виконання створюється новий ПІ.
 - Оголошення класу додає його до ПІ. Клас та кожен його екземпляр при створенні отримує ще й власний простір імен.
 - Кожен `import` додає модуль чи його компонент до простору імен.
 - Оператори `for` та `with` створюють нові змінні, аналогічно до присвоєння.
 - Включення (comprehensions) створює тимчасові змінні, що зникають після завершення виконання включення.

Оновлення простору імен

- Розглянемо зміни простору імен функцією для знаходження позиції гравця в лабіринті:

```
def get_player_pos(level, player_char='*'):
    """Returns a (x, y) tuple of player char on the level"""
    for y, row in enumerate(level):
        for x, char in enumerate(row):
            if char == player_char:
                return x, y
```

- Виклик `dir()` після входу в функцію виявить `get_player_pos` у просторі імен, проте локальних змінних функції у ньому не буде.
 - При виклику `dir(get_player_pos)` їх теж не видно.
 - Змінні створюються динамічно підчас виконання функції.
- Змінні `x`, `row`, `y`, `char` з'являться при заході у відповідні цикли та видаляються після завершення роботи функції (циклів).
 - Причина: простір імен мав локальну область видимості (*local scope*).

Простори імен та Local Scope

- Дві змінні не обов'язково однакові, навіть якщо мають одну назву.

```
def f():  
    a = 2  
  
a = 1  
f()  
print(a)
```

1

```
a = 1  
  
def f():  
    print(a)  
  
f()
```

1

```
a = 1  
  
def f():  
    a = 2  
    print(a)  
  
f()
```

2

```
a = 1  
  
def f():  
    print(a)  
    a = 2  
  
f()
```

`UnboundLocalError: local variable 'a' referenced before assignment`

Простори імен – основа можливостей Python

- Спрощено, Python складається із вкладених просторів імен.
 - Простори імен часто змінюються під час роботи програми.
- Вплив на налагоджування коду:
 - Немає реальних відмінностей між функціями, методами класів та об'єктами, які містять дані.
 - Простори імен змінюються та рекомбінуються (декоратори функцій, метакласи тощо)
 - Немає строгої інкапсуляції в Python. Зручно, проте складно тримати сутності окремо; заборонити коду змінювати конкретний простір імен немає можливості.
- Необхідно тримати простори імен добре організованими!

Використання самодокументованих об'єктів

- Доступ до Docstrings за допомогою help():

- In [19]: help(draw_maze.draw_grid)

```
Help on function draw_grid in module draw_maze:
```

```
draw_grid(data, tile_img, tiles)  
Returns an image of a tile-based grid
```

- Натиснувши 'q', залишаємо help-сторінку.
 - Для глибинного розуміння доречніша документація Python, функція help() оновлює пам'ять.

help() також надає списки із вмістом пакетів

- Інколи `dir` не працює добре з пакетами (коли `__init__.py` порожній).
- Допомогає `help()`:
 - In [20]: `import pygame`
In [21]: `help(pygame)`
- Документація містить автоматично згенеровану секцію `PACKAGE CONTENTS`, де перелічені всі модулі з пакету.

```
>>> import xml
>>> help(xml)
Help on package xml:

NAME
    xml - Core XML support for Python.

DESCRIPTION
    This package contains four sub-packages:

    dom -- The W3C Document Object Model. This supports DOM Level 1 +
           Namespaces.

    parsers -- Python wrappers for XML parsers (currently only supports Expat).
```

Огляд об'єктів у IPython

- Команда `?` з назвою об'єкта дає зведення щодо типу об'єкту, його вмісту та опис:

```
In [3]: ?maze
Type:      list
String form: [['#', '#', '#', '#', '#', '#', '#'],
               ['#', '.', '.', '.', '.', '.', '#'],
               ['#', '.', '.', '.', '<...>', '.', '.', '.', '#'],
               ['#', '.', '.', '.', '.', 'x', '#'],
               ['#', '#', '#', '#', '#', '#', '#']]

Length:      7
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```


Аналіз типів об'єктів

- Можемо реалізувати функцію для пересування гравця (або скопіювати її за допомогою %paste в IPython).
 - Додаємо вектор переміщення до розташування гравця та змінюємо карту відповідно:

```
def move(level, direction):  
    """Handles moves on the level"""  
    oldx, oldy = get_player_pos(level)  
    newx = oldx + direction[0]  
    newy = oldy + direction[1]  
    if level[newy][newx] == 'x':  
        sys.exit(0)  
    if level[newy][newx] != '#':  
        level[oldy][oldx] = ' '  
        level[newy][newx] = '*'
```

Розпочнемо з початкової позиції гравця (лівий верхній кут лабіринту), позначивши гравця символом *:

```
In [22]: maze = create_maze(12, 7)
```

```
In [23]: maze[1][1] = '*'
```

```
TypeError  
Traceback (most recent call last)  
<ipython-input-5-b9a7a1b90faf> in <module>()  
----> 1 maze[1][1] = "*"   
  
TypeError: 'str' object does not support item assignment
```

Використаємо інтроспекційну функцію type() :

```
In [24]: type(maze)
```

```
str
```

Функція `type()`

- Повертає тип об'єкта (клас):
 - Працює для будь-якого вбудованого чи користувацького типу даних.
 - Виявилось, що лабіринт – один незмінюваний рядок.
 - Для внесення змін треба конвертувати рядок у двовимірний список.
- Такий код писався раніше – у функції `draw_maze.parse_grid()`:
 - `In [25]: maze = draw_maze.parse_grid(maze)`
- Тепер тип лабіринту – список:
 - `In [26]: type(maze)`
 - `list`

Збираємо код програми, яка виконує випадкові блукання в лабіринті

```
If __name__ == '__main__':  
    tile_img, tiles = load_tiles()  
    maze = create_maze(12, 7)  
  
    maze = parse_grid(maze)  
    maze[1][1] = '*'  
    for i in range(100):  
        direction = random.choice([LEFT, RIGHT, UP, DOWN])  
        move(maze, direction)  
    img = draw_grid(maze, tile_img, tiles)  
    image.save(img, 'moved.png')
```

Де потрібна інтроспекція?

- Ситуації:
 - Розгляд бібліотеки
 - Експериментування з фрагментами коду
 - Дослідження типу об'єктів під час налагодження
 - Огляд об'єктів після запуску програми в IPython
 - Визначення просторів імен, що перетинаються тощо
- Найчастіше використовується для налагодження.
- Знаходження опечаток за допомогою інтроспекції:
 - In [16]: player_pos = 7
 - In [17]: playr_pos = player_pos + 1
 - Присутній дефект видно в коді. Проте після запуску dir помилка стає очевидною:

```
..  
'player_pos',  
'playr_pos',  
..
```

Комбінація інтроспекційних функцій

- Помилки Python можна ідентифікувати за допомогою інструкції:
- `[x for x in dir(__builtin__) if 'Error' in x]`
 - Команда неточна, оскільки не всі помилки мають у назві Error.
 - Більш коректний підхід є виведення нащадків базового класу Exception з модуля `__builtins__`:

```
for name in dir(__builtins__):  
    obj = getattr(__builtin__, name)  
    if obj.__class__ == type \  
        and issubclass(obj, Exception):  
        print(obj)
```


- При написанні невеликих програм часто інтроспекція відбувається паралельно з написанням коду, а в великих проектах частіше застосовується при налагодженні конкретного дефекту в готовому коді.
 - Інколи інтроспекційні функції є частиною функціональності програми.

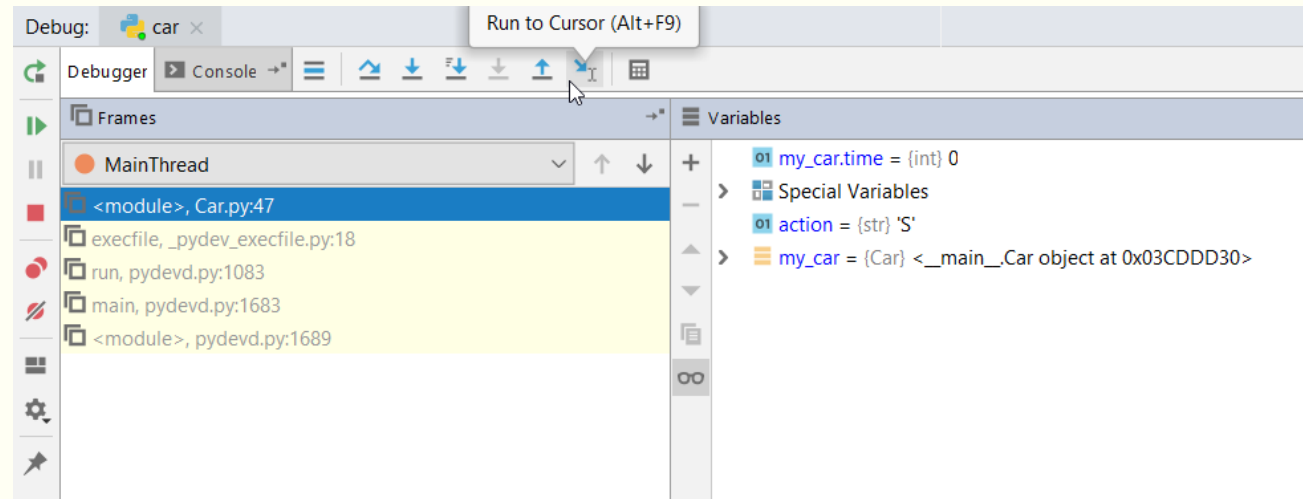
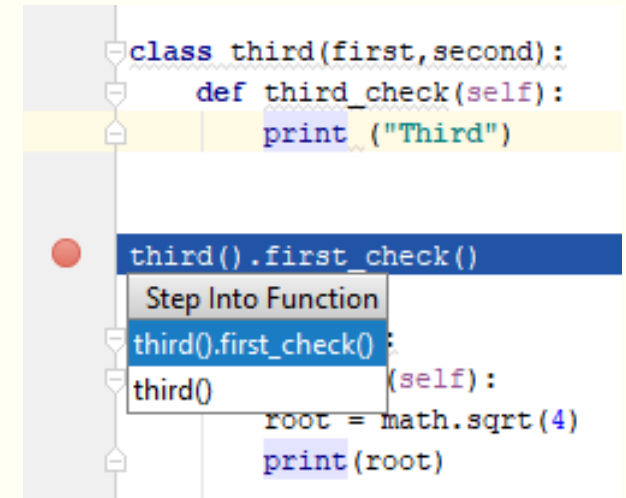
Сесії налагодження в PyCharm

- Smart step into

- Інколи при налагоджуванні потрібно зайти всередину функції / методу.
- Натисніть Shift+F7 для вибору конкретної функції:

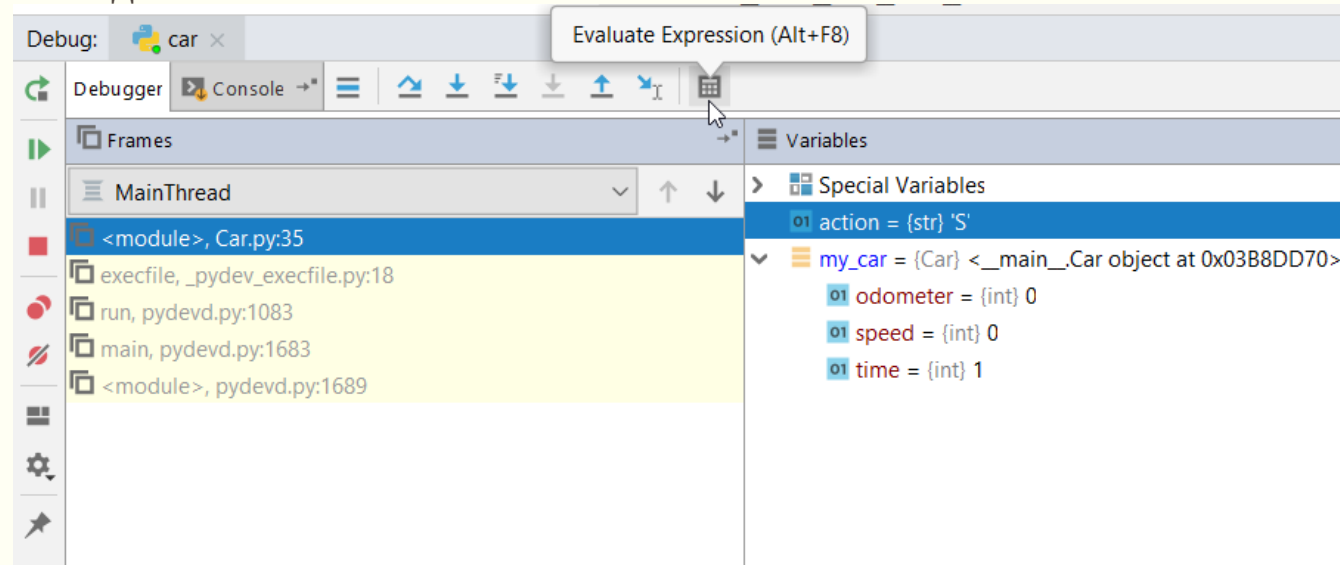
- Run to cursor

- Для зупинки виконання коду в заданій позиції курсора без додавання точки розриву натисніть Alt+F9 або іконку 



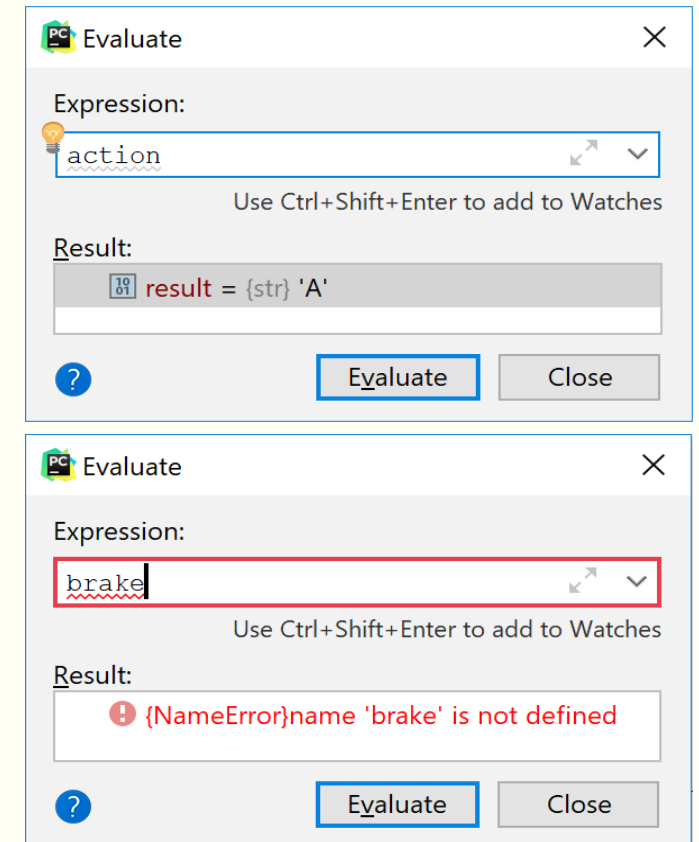
Сесії налагодження в PyCharm

- Обчислення виразу
 - У режимі налагодження можна обчислювати будь-який вираз за допомогою Alt+F8.



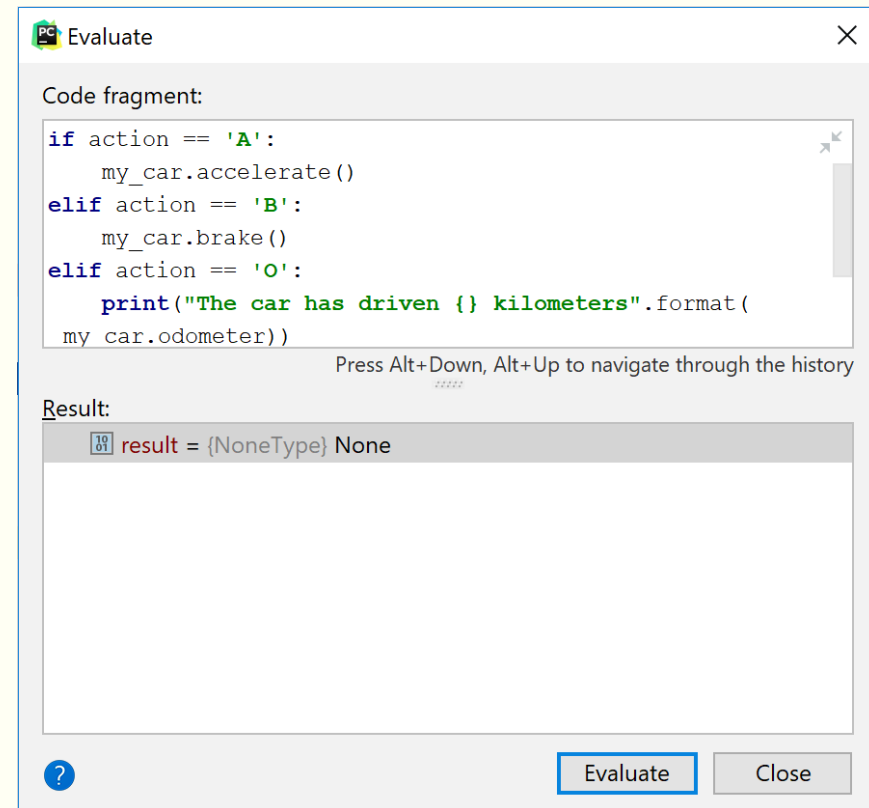
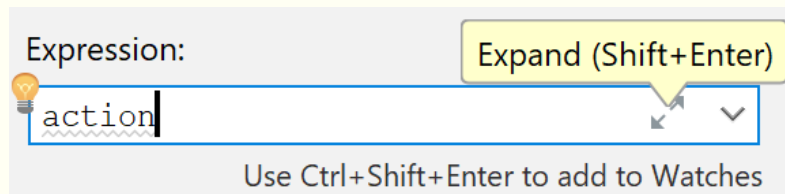
Доступні режими отримання значень:

- *Expression Mode* для обчислення однорядкових виразів.
- *Code Fragment Mode* для оцінки невеликих фрагментів коду: циклів, операторів галуження тощо.



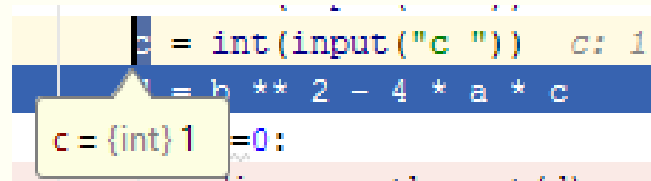
Для обчислень у фрагменті коду натисніть Shift+Enter або стрілки

- Вставте цільовий фрагмент коду в текстове поле та натисніть Evaluate.
- Для навігації по фрагменту натискайте Alt+Down та Alt+Up.

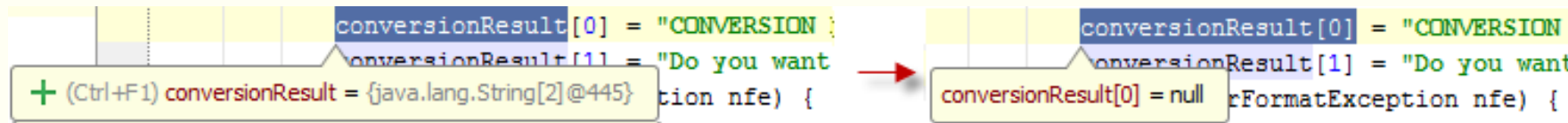


Обчислення виразів у реакторі

- Під час сесії налагоджування значення будь-якого виразу показується у спливаючих підказках при наведенні курсору.
 - Якщо вираз містить дочірні вирази, натисніть «+» для їх показу.



- Швидко обчислити вираз за допомогою Quick Evaluate:
 - Автоматично** - налаштувати Debugger | Data Views: спочатку задіяти **Show value tooltip on code selection**, а потім – виділяти фрагмент коду або натискати Ctrl + W:



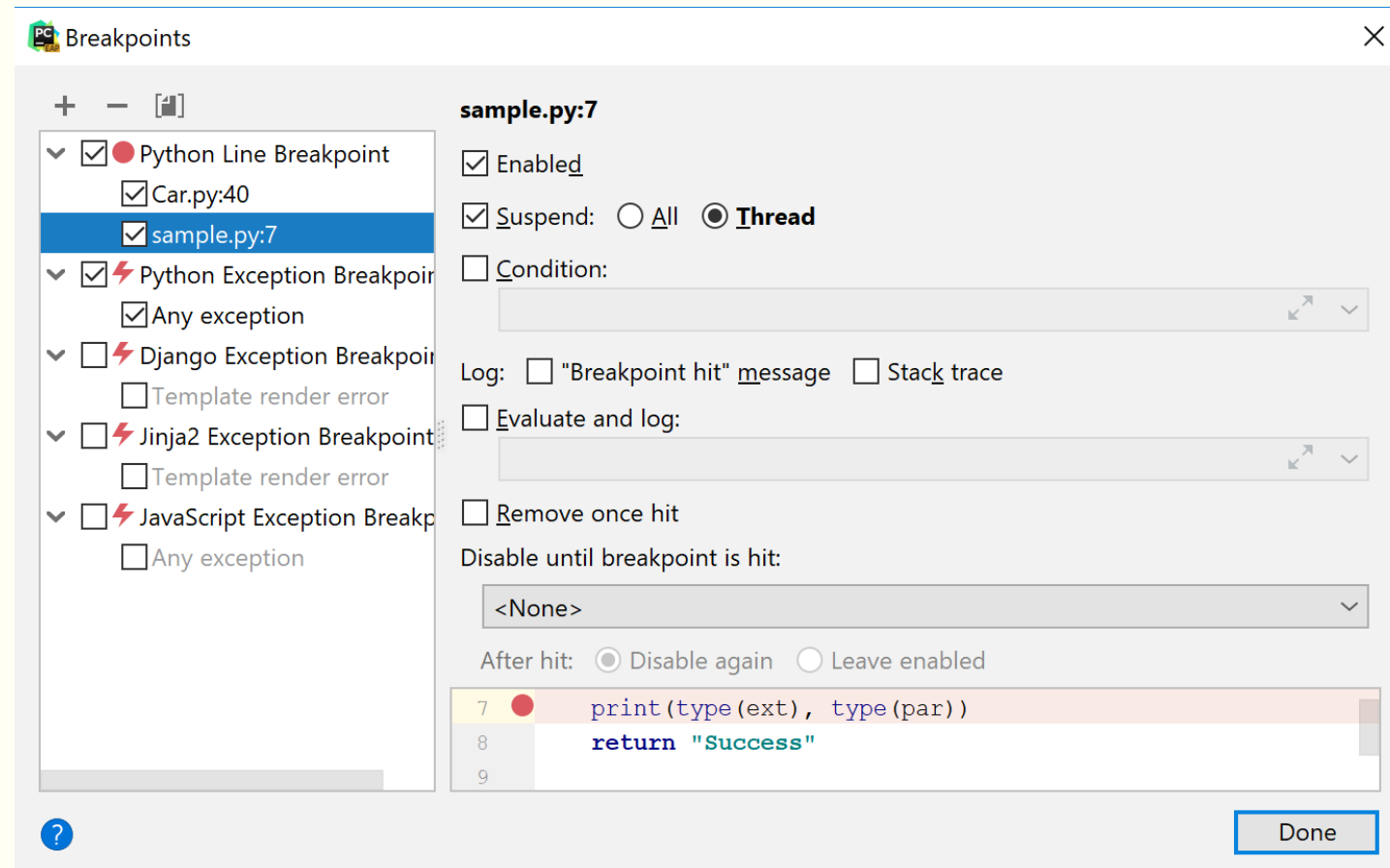
- Вручну** – розмістити каретку в потрібному місці в коді та обрати Run | Quick Evaluate Expression (або натиснути Ctrl+Alt+F8).

Хоткеї для налагоджування в PyCharm

| Дія | Хоткей |
|---|---|
| <u>Toggle breakpoint</u> (робота з точкою зупинки) | Ctrl+F8 |
| <u>Відновлення налагоджування</u> | F9 |
| <u>Step over</u> (налагодження без входження всередину функцій) | F8 |
| <u>Step into</u> (налагодження з входженням всередину функцій) | F7 |
| Стоп | Ctrl+F2 |
| <u>Переглянути деталі щодо точок зупинки</u> | Ctrl+Shift+F8 |
| Налагодити код біля каретки | Shift+F9 (в головному методі) або Shift+Alt+F9 |

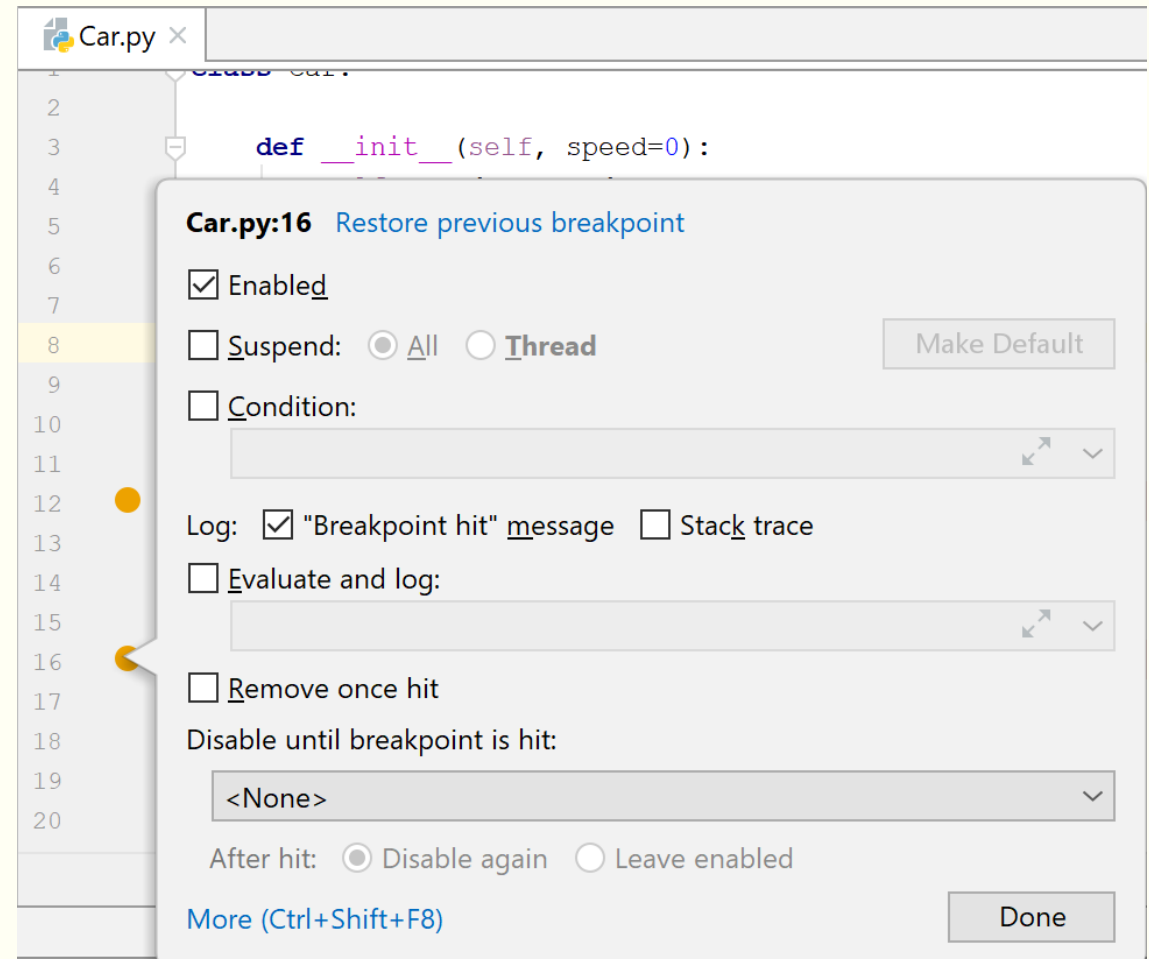
Налагодження коду в PyCharm. Breakpoints

- Для налаштування точок зупинки натисніть Ctrl+Shift+F8.



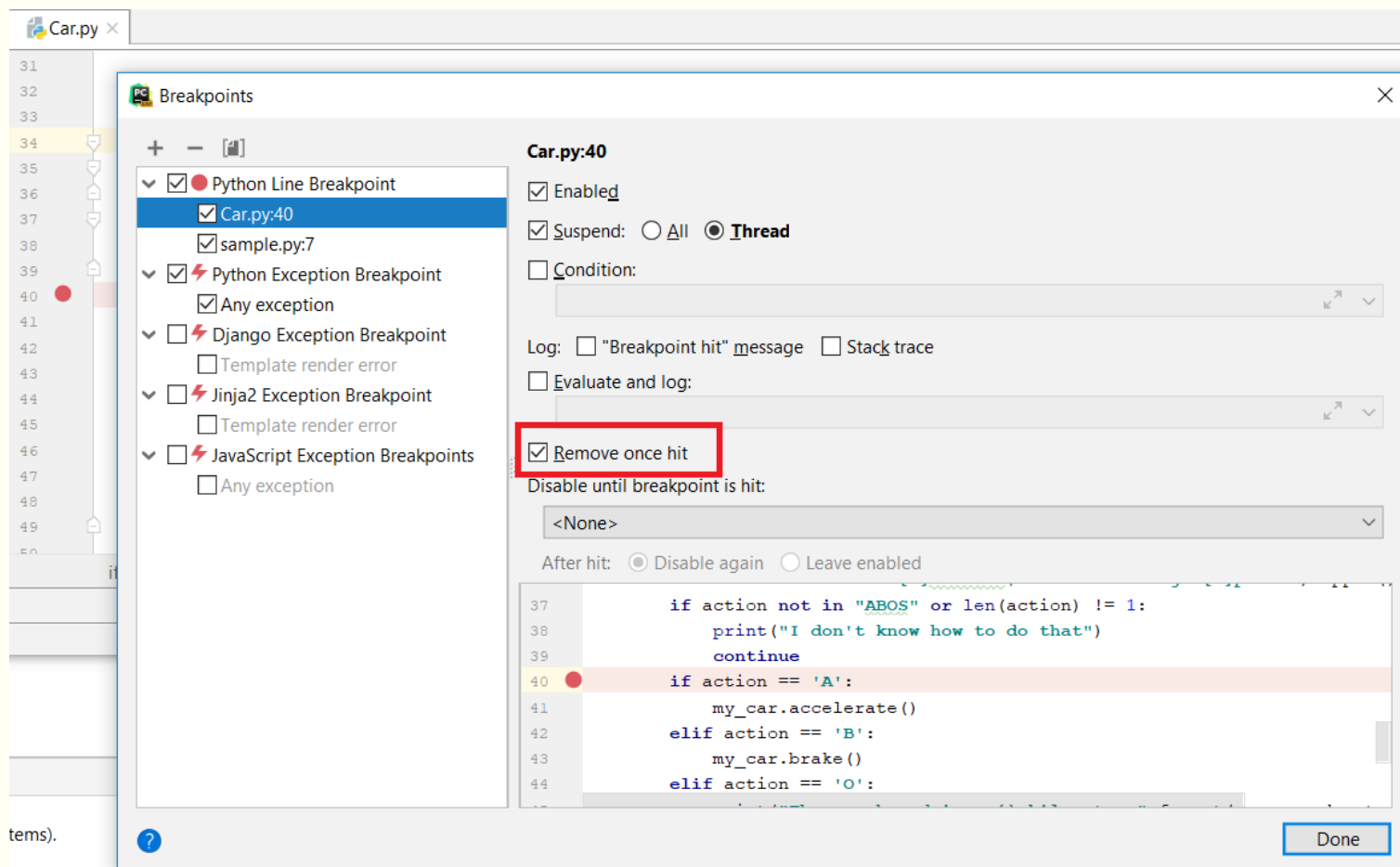
Точка зупинки з дією (action breakpoint)

- Використовуйте action breakpoints, щоб обчислити значення змінної в конкретному рядку коду без зупинки (suspend) його виконання.
 - Для створення action breakpoint при натисненні також утримуйте Shift.



Створення тимчасової точки зупинки

- Для створення одноразової точки зупинки також утримуйте Shift+Alt при натисненні.



Вимкнення точок розриву

- Щоб зробити точку зупинки недієвою, при натисненні утримуйте Alt

```
40         if action == 'A':
41             my_car.accelerate()
42         elif action == 'B':
43             my_car.brake()
44         elif action == 'O':
45             print("The car has driven {} kilometers".format(my_car.odometer))
46         elif action == 'S':
47             print("The car's average speed was {} kph".format(my_car.average_speed()))
48     my_car.step()
49     my_car.say_state()
```



ДЯКУЮ ЗА УВАГУ!

Наступна тема: об'єктно-орієнтоване програмування мовою Python

Інтроспекційні функції в Python

| Function | Description |
|-------------------------------|--|
| <code>l (list)</code> | lists a few lines around the one executed next |
| <code>dir()</code> | returns a list of names in the current namespace |
| <code>dir(x)</code> | lists the contents of the namespace in x |
| <code>help(x)</code> | shows the docstring of a Python object |
| <code>x is y</code> | checks identity of two objects (as opposed to <code>==</code>) |
| <code>type(x)</code> | returns the type of an object |
| <code>hasattr(x, s)</code> | returns <code>True</code> if the namespace of x contains the name s |
| <code>getattr(x, s)</code> | returns an attribute with name s from the namespace of x |
| <code>issubclass(x, y)</code> | returns <code>True</code> if x is a subclass of y |
| <code>isinstance(x, y)</code> | returns <code>True</code> if x is an instance of class y |
| <code>callable(x)</code> | returns <code>True</code> if x can be called |
| <code>globals(x)</code> | returns a dictionary of objects in the global scope |
| <code>locals(x)</code> | returns a dictionary of objects in the local scope (e.g., inside a function) |

Перевірка ідентичності об'єкта

- Інколи важливо перевірити, чи є 2 об'єкти дійсно ідентичними, а не лише такими, що містять однакові дані.
 - Використовують оператор `is`, а `==` порівнює лише вміст:

```
In [17]: a = [1, 2, 3]
```

```
In [18]: b = [1, 2, 3]
```

```
In [19]: a == b
```

```
Out[19]: True
```

```
In [20]: a is b
```

```
Out[20]: False
```

Перевірка екземплярів та підкласів

- За допомогою `isinstance` можна перевірити, чи є об'єкт екземпляром деякого класу:
 - `isinstance("abc", str)`
 - `True`
- За допомогою `issubclass` перевіряємо, чи є один клас породженням від іншого:
 - `issubclass(str, object)`
 - `True`

-
- Робота з Jupyter Notebook