



# СТРАТЕГІЇ НАЛАГОДЖЕННЯ ПРОГРАМНОГО КОДУ МОВОЮ PYTHON

Питання 6.3

# Налагодження (debugging) коду

---

- Налагодження передбачає кілька припущень:
  - Відомо, що повинна робити коректна програма.
  - Відомо, що в програмі є дефект.
  - Зрозуміло, що дефект потрібно налагодити.
  - Відомо, як це зробити.
- Розглянемо 3 стратегії налагодження винятків:
  - 1. Читання коду в місці помилки.
  - 2. Ідентифікація помилки в повідомленні.
  - 3. Перехоплення винятків (Catching Exceptions).
- Загалом винятки в Python відносять до однієї з двох категорій:
  - Винятки, що виникають **до** виконання коду (синтаксичні помилки, SyntaxErrors)
  - Винятки, що виникають **під час** виконання коду.
    - Інтерпретація коду починається лише тоді, коли відсутні синтаксичні помилки.

# Синтаксичні помилки (SyntaxError)

---

- Найпростіші для виправлення.
  - Тип `SyntaxError` та його підтип `IndentationError`.
  - Python не може інтерпретувати / токенізувати команду коректно через помилки в її написанні.
  - Часто виникають через опечатки в коді

```
imprt pygame
```

File "load\_tiles.py", line 2  
imprt pygame  
^

`SyntaxError: invalid syntax`

```
TILE_POSITIONS = [  
    ('#', 0, 0), # wall  
    (' ', 0, 1), # floor  
    ('.', 2, 0), # dot  
    ('*', 3, 0), # player
```

`SyntaxError: unexpected EOF while parsing`

# Схожі дефекти в коді можуть вести до різних повідомлень про помилку!

---

- Приклад: редактор коду рахує дужки та вказує на їх відсутність.
  - Можна відстежити `SyntaxError`, проте опис часто неточний.
  - Відсутність відкриваючої дужки спровокує іншу помилку, ніж відсутність закриваючої: отримаємо `IndentationError`.

```
TILE_POSITIONS = ('#', 0, 0), # wall
                  (' ', 0, 1), # floor
                  ('.', 2, 0), # dot
                  ('*', 3, 0), # player
                  ]
```

IndentationError: unexpected indent

# 1. Стратегія читання помилки в місці виникнення

---

- Найбільш поширена стратегія:
  - Спершу переглянути рядок, на який вказує повідомлення про помилку.
  - Далі розгляньте рядок над ним.
  - Скопіюйте частину коду в окремий файл: чи залишилась `SyntaxError`?
  - Перевірте пропущені двокрапки після інструкцій `if`, `for`, `def`, `class` тощо.
  - Перевірте відсутні дужки, редактор має допомогти.
  - Перевірте незакриті лапки.
  - Закоментуйте рядок коду, що викликає помилку: чи залишилась помилка?
  - Перевірте версію Python.
  - Перевірте, що код відповідає PEP8

## 2. Стратегія розгляду повідомлення про помилку

---

- Для створення зображення утворимо словник будівельних блоків (tiles) лабіринту.
  - Прямокутники є об'єктами `pygame.Rect`.
  - Створюємо прямокутники в допоміжній функції `get_tile_rect()`, а словник – у функції `load_tiles()`.

```
from pygame import image, Rect, Surface

def get_tile_rect(x, y):
    """Converts tile indices to a pygame.Rect"""
    return Rect(x * SIZE, y * SIZE, SIZE, SIZE)

def load_tiles():
    """Returns a dictionary of tile rectangles"""
    tiles = {}
    for symbol, x, y in TILE_POSITIONS:
        tiles[x] = get_tile_rect(x, y)
    return tiles
```

## 2. Стратегія розгляду повідомлення про помилку

---

- Тепер можна викликати функцію та спробувати отримати блок стінки (wall tile, позначений '#') з нашого словника:

```
tiles = load_tiles()  
r = get_tile_rect(0, 0)  
wall = tiles['#']
```



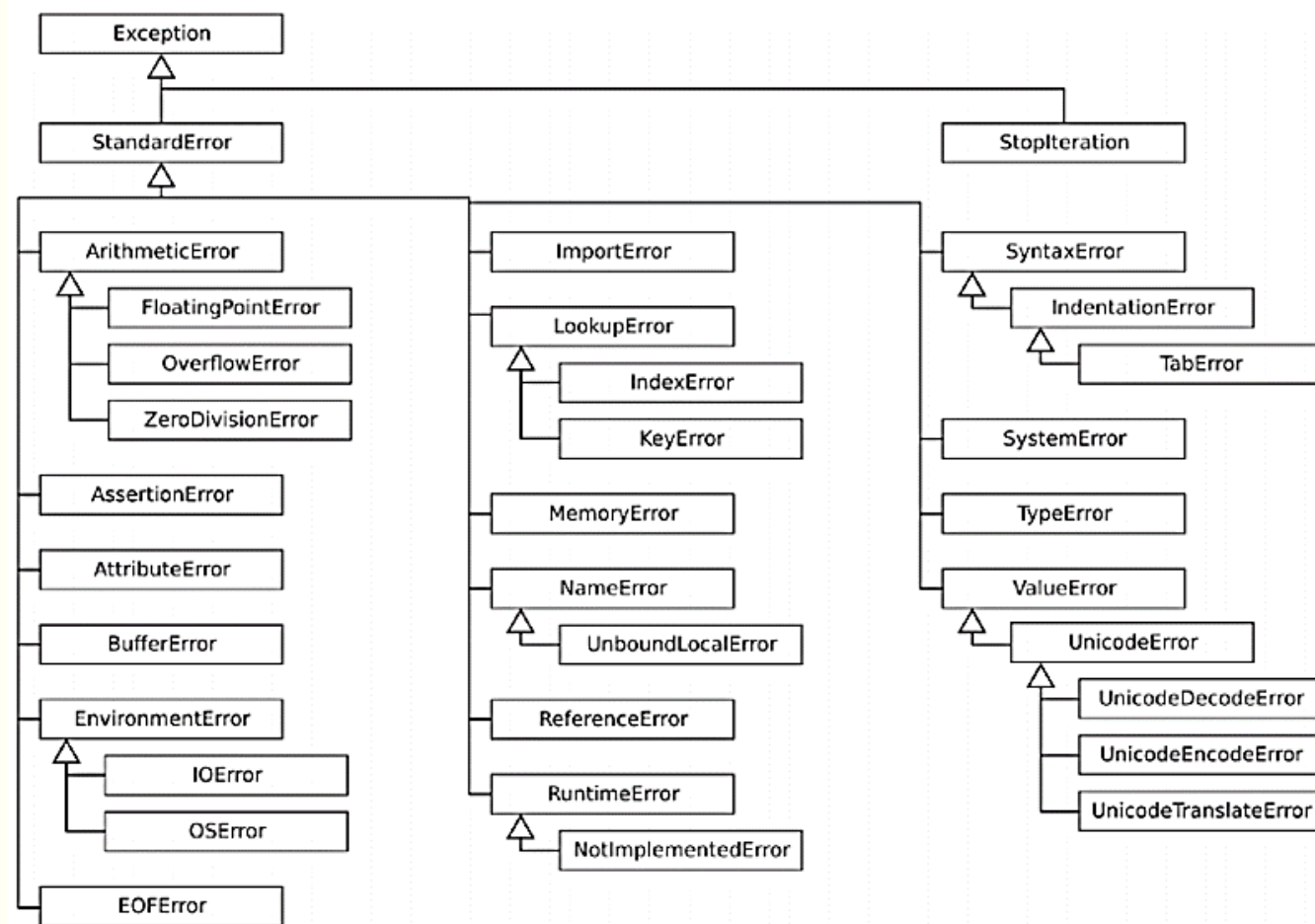
```
Traceback (most recent call last):  
  File "load_tiles.py", line 32, in <module>  
    wall = tiles['#']  
KeyError: '#'
```

- Незважаючи на помилку, проблем із запитом на отримання '#' з tiles не видно: помилка десь в іншому місці.
- Повідомлення про помилку в Python містить 3 важливі частини:
  - Тип помилки,
  - Опис помилки,
  - Трасування (the traceback).

# Тип помилки

- Повідомлення про помилку означає, що Python викликав деякий клас винятка (Exception).
  - Усі винятки є підкласами класу Exception.
  - У Python 3.5 присутні 47 різних типів виключень.
  - Побачити повний список можна, виконавши код

```
[x for x in dir(__builtins__)  
if 'Error' in x]
```





# Опис помилки та трасування (traceback)

---

- Текст після типу помилки описує проблему, яка трапилась.
  - Точність таких описів різна.
  - Наприклад, при виклику функцій із занадто великою або малою кількістю аргументів:  
**`TypeError: get_tile_rect() takes 2 positional arguments but 3 were given`**
- У випадку з `KeyError` єдина отримана інформація – символ `'#'`.
  - Цієї інформації недостатньо.
- Трасування містить точну інформацію, де в коді трапився виняток:
  1. Копію виконаного коду. Інколи дефект відстежується відразу.
  2. Номер рядка, в якому трапилась помилка. Дефект повинен бути в цьому рядку або в раніше виконаному коді.
  3. Виклики функцій, що призвели до помилки. При отримуванні довгого тексту починайте його читання з кінця. Не завжди, проте часто записи в кінці підказують, де шукати проблему.

```
Traceback (most recent call last):  
  File "load_tiles.py", line 32, in <module>  
    wall = tiles['#']  
KeyError: '#'
```

# Дедуктивний логічний вивід

---

- Якщо ключ '#' відсутній у словнику, чи взагалі ключ записується?
- У якому рядку це відбувається?
- Чи був досягнутий цей рядок?
  - У функції `load_tiles()` присвоюються неправильні ключі:
  - Потрібно писати `tiles[symbol] = get_tile_rect(x, y)`
- Певне логічне виведення було необхідним, оскільки Exception трапився в іншому рядку, ніж вказав опис помилки.
  - Може існувати кілька можливих дефектів, що ведуть до одного симптому.
  - Інколи доводиться перевіряти кілька місць.

```
def load_tiles():  
    """Returns a dictionary of tile rectangles"""  
    tiles = {}  
    for symbol, x, y in TILE_POSITIONS:  
        tiles[x] = get_tile_rect(x, y)  
    return tiles
```

### 3. Перехоплення виключень (Catching Exceptions)

---

- Далі можна спробувати завантажувати зображення з tiles:

```
from pygame import image, Rect  
tile_file = open('tiless.xpm', 'rb')
```



```
FileNotFoundError: [Errno 2] No such file or directory: 'tiless.xpm'
```

- Дефект – помилка в назві файлу.
  - Потрібно перевірити шлях до файлу та його назву в коді.
  - Часто проблеми в деталях: абсолютних та відносних шляхах, пропущених знаках «\_», «-», «\\» у рядках у мові Python.
  - Дефект, який веде до IOError, майже завжди полягає в неправильній назві файлу.

### 3. Перехоплення виключень

---

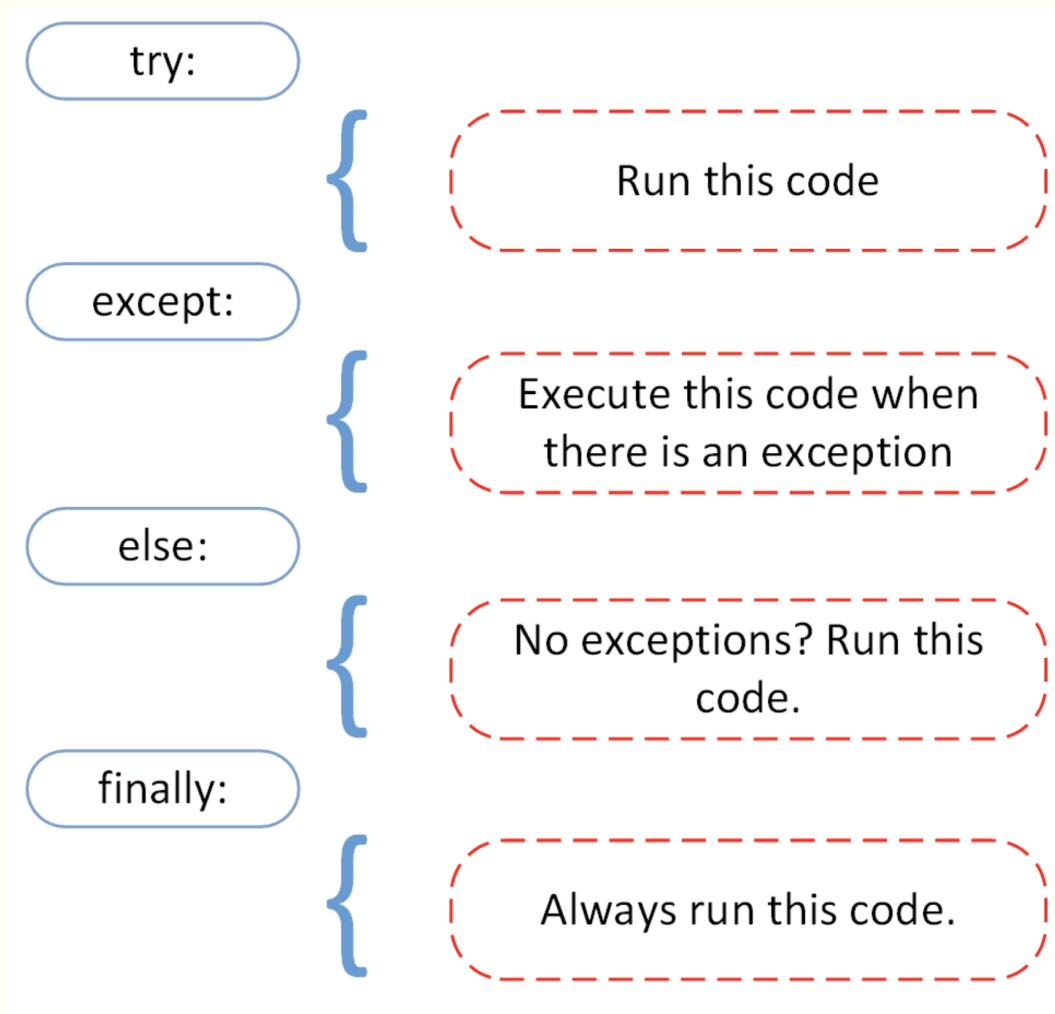
- Неможливо попередити кожний Python-виняток.
  - Альтернатива – реагувати на Винятки в коді програми.
  - Спробувати виконати операцію з перспективою її збою дозволяє оператор `try..except`:

```
filename = input("Enter the file name: ")
try:
    tiles = load_tile_file(filename)
except IOError:
    print("File not found: {}".format(filename))
```

- В операторі `except` реагуємо на конкретний виняток.
  - Перехоплення (Catching) виключень корисне для зберігання важливих даних до переривання роботи програми.

the path Exceptions take through a program is *invisible* in the code.

---



- Буває складно врахувати всі можливі шляхи виконання програми та місця виникнення винятків.
- Потрібно обережно вирішувати, які винятки перехоплювати:
  - diaper pattern – погана практика:

```
try:  
    call_some_functions()  
except:  
    pass
```
  - Перехоплюється все, винятки зникають, проте значно ускладнюється їх діагностика.
  - Хороша практика – використовувати try.. except тільки у визначених ситуаціях та перехоплювати певний тип виключень.

# Кращі практики налагодження IOError

---

1. Знайти точне місце файлу в терміналі чи переглядачі.
  2. Надрукувати шлях до файлу, що використовується в програмі. Порівняти зі справжнім шляхом.
  3. Перевірити поточну робочу директорію (`import os; print(os.getcwd ())`).
  4. Замінити відносні шляхи абсолютними.
- **На Unix:** перевірити права доступу до файлу.
    - Використовуйте модуль `os.path` для обробки шляхів та папок.
    - Обережно із бекслешами у шляху! Необхідно замінити їх на слеші (/) або подвійні бекслеші (\\).

# Помилки та дефекти: резюме

---

- Підсумовуємо спостереження:
  - Інколи повідомлення про помилку прямо вказує на дефект (`SyntaxError`).
  - Часто це не так і потрібно розглядати кілька можливостей (як з `IOError`).
  - Інші повідомлення про помилку більш запутані та вимагають досвіду, щоб з'ясувати проблему.
- У Python багато дефектів можуть призводити до однієї помилки.
  - Поширена ситуація для `TypeError`, `ValueError`, `AttributeError` та `IndexError`.
- Інша ситуація, коли ми надаємо неправильні дані бібліотечним функціям.
  - Винятки викидається з бібліотеки, а не з Вашого коду.
  - Тут повідомлення про помилку визначальне: звертайте увагу на всі три риси – місце в коді, тип помилки та результати трасування.

# Основні причини виникнення дефектів

---

- **Помилки під час реалізації:** відсутні двокрапки, помилки в назвах змінних, забуті чи неправильно використані параметри тощо.
  - Більшість цих дефектів знайдуться рано, часто з проявом виключень.
- **Погане планування** – код уже задумувався неправильно: обрано недоречний підхід до вирішення задачі, не враховано важливі деталі реалізації тощо.
  - Важче розпізнаються. Хороша стратегія усунення – тестування на ранньому етапі розробки.
- **Поганий дизайн коду** веде до дефектів опосередковано: надмірний код та недостатня документованість ускладнює подальші зміни програми.



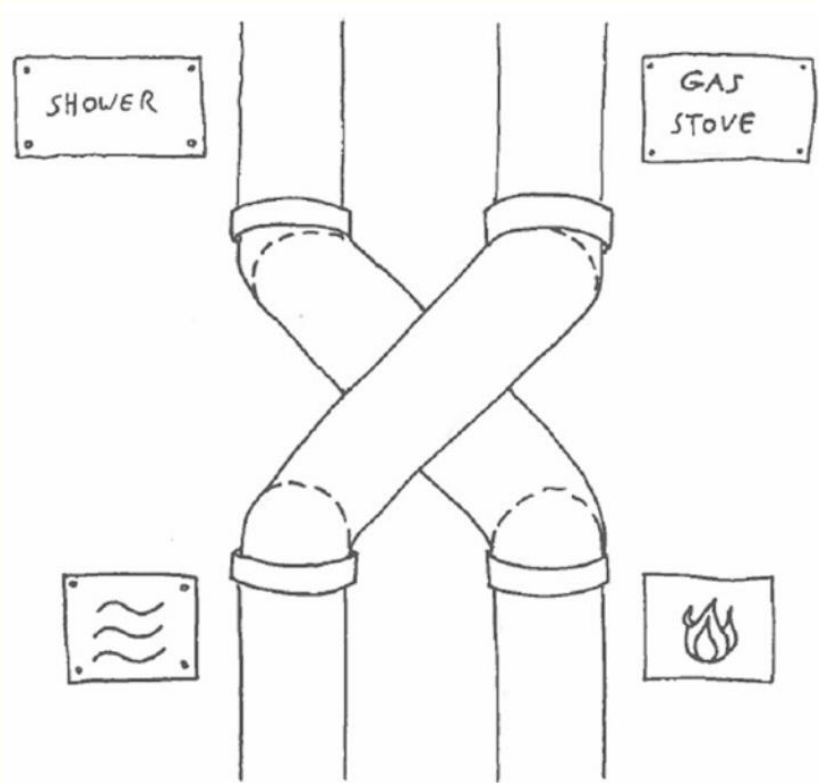
# Рекомендовані практики написання коду

---

- Дефекти, які не виводять повідомлення про помилку, називають **семантичними помилками** (*semantic errors*).
  - Для їх відстеження корисно перевіряти очікуваний вивід при заданому вводі даних.
  - Деякі помилки видно відразу з виводу, деякі потрібно віднайти в коді.
  - Деякі дефекти не спричиняють помилку виводу напряду, а поширюються в програмі, поки не спричинять помилку виводу.
- Крім дефектів, код може містити проектні недоліки (**design weaknesses**).
  - Такий код підлягає помилкам доповнення функціональності, а вже існуючі помилки важче виявляти.

# Семантичні помилки в Python

---

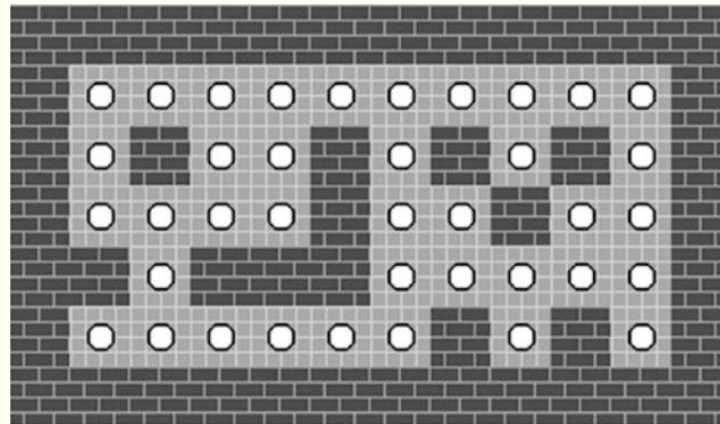


- Трапляються, коли програма працює без виключень, проте результат її роботи відрізняється від очікуваного.
  - Такі дефекти важче відстежити, оскільки немає повідомлень про помилку.
  - Причини семантичних помилок інколи дуже прості: опечатки, пропущені рядки, неправильний порядок інструкцій та ін.
  - Перша можлива проблема – недостатня чіткість очікувань.

# Очікування перевіряються простими прикладами

---

- **Ввід:** два цілих числа  $x$  та  $y$ , що визначають розмір лабіринту в блоках (grid tiles).
- **Вивід:** випадковий лабіринт, що складається з сітки  $x * y$  блоків.
  - Лабіринт оточується фреймом зі стінок (wall tiles, #).
  - Містить взаємопов'язану мережу з'єднаних коридорів, розділених стінками та заповнених крапками.
  - Виглядає, наприклад, так:



# Дефекти у присвоєнні змінних

---

- Розпочнемо з функції `create_grid_string()`, яка продукує остаточний лабіринт.
  - Вхід: Python-множина з крапок та розміри сітки `x`, `y`.
  - Вивід: лабіринт у вигляді рядка.
  - Множина забезпечує унікальність позиції кожної крапки:

```
dots = set(((1,1), (1,2), (1,3), (2,2), (3,1), (3,2), (3,3)))  
create_grid_string(dots, 5, 5)
```

- Очікуємо отримати:

```
#####  
#.#.#  
#...#  
#.#.#  
#####
```

# Можлива реалізація create\_grid\_string()

---

```
def create_grid_string(dots, xsize, ysize):  
    grid = ""  
    for y in range(ysize):  
        for x in range(xsize):  
            grid += "." if (x, y) in dots else "#"  
        grid += "\n"  
    return grid
```

- Починаємо з присвоєння сітці порожнього рядка.
- Можливі проблеми:
  - Пропустивши цю інструкцію, отримаємо UnboundLocalError.
  - Ініціалізувавши на рядок нижче, в циклі, отримаємо тільки #####:
    - for y in range(ysize):  
 grid = ""

# Ненавмисне присвоєння чи порівняння

---

- Схожа семантична помилка, пов'язана з неправильним оператором присвоєння.
  - Наприклад, замінимо оператор «+=» на «=»: `grid = "\n"`
  - Скоріше всього, результат опечатки.
- На прикладі тернарного виразу `grid = "." if (x, y) in dots else "#"`
  - Змінна `grid` реініціалізується на кожній ітерації циклу.
  - Виведеться тільки символ `#`.
- Менш поширена помилка: `grid == "\n"`
  - Часто є результатом опечатки.
  - У результаті дефекту вивід не міститиме переходів на новий рядок:  
`#####.#.##...##.#.#####`
- Відстежити дефект можна за 2 кроки:
  - 1) зрозуміти, що відсутні переходи на новий рядок.
  - 2) перевірити відповідний код.

# Неправильні змінні у виразі

---

- Нехай буде сітка розміром (xsize, ysize) замість (x, y) з тернарного виразу:

- `grid += "." if (xsize, ysize) in dots else "#"`
- `if`-умова ніколи не стане `True`, а вираз ніколи не поверне `"."`
- Отримаємо суцільну стіну:

```
#####  
#####  
#####  
#####  
#####
```

- Можна переплутати рядки в тернарному виразі:

- Запишемо `grid += "#" if (x, y) in dots else "."`
- Вивід буде інвертованим:

```
.....  
.#.#.  
.###.  
.#.#.  
.....
```

- Дефекти, засновані на інверсії, переміщенні та плутанині дуже поширені.

- Такий вид дефектів може накладатись на інші дефекти.
- Наприклад, інвертування з оператором `not` (вивід знову коректний):
- `grid += "#" if not (xsize, ysize) in dots else "."`

# Дефекти в індексації

- 1) Для створення лабіринту генеруємо всі позиції, в яких крапки можуть з'являтися (крім меж лабіринту).
  - Для сітки  $5 \times 5$  очікуваний вивід:
  - Можна застосувати спискове включення (розглядатиметься пізніше):

```
#####  
#...#  
#...#  
#...#  
#####
```

```
def get_all_dot_positions(xsize, ysize):  
    return [(x,y) for x in range(1, xsize-1) for y in range(1, ysize-1)]
```

- Утворений список надходить у функцію `create_grid_string()`, яка генерує коректний вивід:
  - `positions = get_all_dot_positions(5, 5)`
  - `print(create_grid_string(positions, 5, 5))`

range(0, xsize)	range(1, xsize)	range(0, xsize-1)	range(2, xsize+2)
.....	#####	....#	#####
.....	#....	....#	#####
.....	#....	....#	##.##
.....	#....	....#	#####
.....	#....	#####	#####



# Використання неправильних індексів

- Для створення з'єднаних коридорів потрібно ідентифікувати сусідні положення в сітці.

- У 2D-сітці їх 8:

- Можемо згенерувати вивід сітки:

- `neighbors = get_neighbors(2, 2)`
  - `print(create_grid_string(neighbors, 5, 5))`

```
def get_neighbors(x, y):  
    return [  
        (x, y-1), (x, y+1), (x-1, y), (x+1, y),  
        (x-1, y-1), (x+1, y-1), (x-1, y+1), (x+1, y+1)  
    ]
```

```
#####  
#...#  
#.#.#  
#...#  
#####
```

- Можливі проблеми:

- 1) збільшення або зменшення індексів на 1 у функції `get_neighbors()`.
- 2) в одному з кортежів `x` та `y` можуть помінятися місцями, замість `(x, y+1)` отримаємо `(y, x+1)`.
  - Відразу помилка не проявиться, проте обчислення сусідів у комірці 3, 2 виведе геть інший рядок:

- Код погано спроектовано спочатку.

- Для квадратних сіток порядок `x` та `y` не такий важливий, як для прямокутних.
  - Читати функцію `get_neighbors()` складно.

```
#####  
##...  
##.#.  
##.#.  
##.##
```

# Приклад абсолютно некоректного повернення

---

```
def get_neighbors(x, y):  
    return [  
        (x, -1), (y, x+1), (x-(1), y), (x+1), y,  
        (x,(-1, y)), (x+1, y, 1), (x-1, y+1, x+1, y+1)  
    ]
```

- Код все ще буде виконуватись



```
#####  
#####  
##.##  
#####  
##.##
```

# Дефекти в інструкціях управління ходом виконання програми

---

- Алгоритм генерації лабіринту може працювати так:
  1. Створити список усіх позицій у сітці.
  2. Обрати випадкову позицію зі списку.
  3. Якщо позиція має до 4 сусідніх крапок, позначити її крапкою.
  4. Інакше позначити позицію стіною.
  5. Повторити кроки 1-4, поки список позицій не стане порожнім.
- Для розміру 5x5 лабіринт конструюється збалансований, коридори зв'язані між собою.
  - Крім того, допускається круговий шлях, який доречний при втечі від привидів.

# Реалізація алгоритму

---

```
def generate_dot_positions(xsize, ysize):
    positions = get_all_dot_positions(xsize, ysize)
    dots = set()
    while positions != []:
        x, y = random.choice(positions)
        neighbors = get_neighbors(x, y)
        free = [nb in dots for nb in neighbors]
        if free.count(True) < 5:
            dots.add((x, y))
            positions.remove((x, y))
    return dots
```

- Алгоритм не ідеальний.
  - При тестуванні інколи можна помітити недоступні зони лабіринту – ще один недолік проектування.
  - Для наших цілей процедури достатньо, проте для більшого розміру алгоритм має бути вдосконалено.

# Дефекти в булевих виразах

---

- Оператори `if` та `while` містять булеві вирази.
  - Дефекти в умовах змінюють команди, які мають виконуватись далі.
  - Наприклад, випадково використано символ `(>)` замість `(<)`:
    - `if free.count(True) > 5:`
    - Булевий вираз ніколи не стане `True`, позиції не будуть відмічатись крапками.
- Аналогічний за результатом дефект – пропуск виклику `free.count()`.
  - У Python 2.7 такий вираз спрацює: `if free.count < 5:`
  - У Python 3.5 буде викинуто виняток.

# Дефекти з відступами

---

- Поширена проблема в циклах while.
  - Усунення табуляції у виклику `positions.remove()` :

```
...  
    if free.count(True) < 5:  
        dots.add((x, y))  
positions.remove((x, y))  
return dots
```

- Наслідок: список позицій ніколи не стає порожнім, програма зациклюється.
  - Теоретичне питання – чи можна довести існування дефекту в програмі?
  - Практичне питання – скільки чекати, поки не станеться проблема?

# Дефекти у використанні функцій

---

- Доповнимо програму короткою функцією `create_maze()`, яка використовує код, створений для генерації лабіринту:

```
def create_maze(xsize, ysize):  
    """Returns a xsize*ysize maze as a string"""  
    dots = generate_dot_positions(xsize, ysize)  
    maze = create_grid_string(dots, xsize, ysize)  
    return maze  
  
maze = create_maze(12, 7)  
print(maze)
```

- 1. Забування викликати функцію, пропустивши дужки:
  - `maze = create_maze`

# Дефекти у використанні функцій

---

- 2. Пропуск оператора return.
  - Наслідок: функція не передає результат у головну програму, повертає None.
  - Часто виняток не генерується, результат інтерпретується як булевий, програма продовжує виконання.
- 3. Відсутність збереження значення, яке повертається.
  - Можна викликати функцію, проте забути зберегти у змінну результат її роботи: `create_maze(12, 7)`
  - Перша спроба визначити помилку – вивести
  - `create_maze(12, 7)`
  - `print(maze)`
  - Згенерується виняток `NameError`.





# ДЯКУЮ ЗА УВАГУ!

Наступне питання: дослідницьке кодування та інструменти налагодження Python-коду