


## ПРАКТИЧНА РАБОТА 3

### РОБОТА З ФУНКЦІЯМИ В PHYTON

Якщо не встановлено Python 3.6 та среда розробки IDE (наприклад Anaconda, PyCharm Community Edition), в браузері перейдіть на сайт <https://edube.org/sandbox>, вибравши в налаштуванні через  емулятор Python (рис.1).

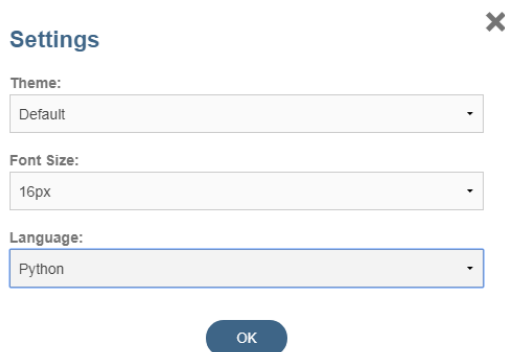


Рисунок 1 – Вікно налаштувань sandbox

Для введення значень з клавіатури використовуйте функцію `input()` з параметрами. Наприклад: `n=int(input("Enter chislo"))`.

### Теоретичні відомості

Функцією називають іменованій фрагмент програмного коду, до якого можна звернутися з іншого місця програми (але є `lambda`-функції, у яких немає імені). Як правило, функції створюються для роботи з даними, які передаються їй як аргументи, також вона може повертати значення.

#### 4.1 Створення функцій

Щоб визначити функцію в мові Python потрібно використовувати ключове слово `def`, після якого ім'я функції та в круглих дужках її параметри. Блок коду функції визначається за допомогою відступу. У функції може йти документаційний рядок, який описує те, що всередині функції відбувається. Доступ до документаційного рядка можна отримати за допомогою атрибута цієї функції (тобто імені, що належить їй) `__doc__`.

Щоб викликати функцію, потрібно використовувати круглі дужки, і якщо потрібно, передати туди параметри.

```
def func_first():  
    """My first function"""  
    print("Hello, Python!")
```

```
print(func_first.__doc__)  
func_first()
```

```
My first function  
Hello, Python!
```

#### 4.2 Параметри функцій та оператор `return`

Параметри вказуються в дужках при оголошенні функції і розділяються комами. Аналогічно передаються значення, коли викликається функція. Імена, зазначені в оголошенні функції, називаються *параметрами*, тоді як значення, які передаються в функцію при її виклику - *аргументами*.

Оператор *return* використовується для повернення значення з функції та виходу з неї. Якщо не писати *return* або не вказувати значення, що повертається, тоді за замовчуванням повернеться *None*.

```
def func_add(a,b):  
    return(a+b)  
  
print(func_add(5,6))          11  
print(func_add('5','6'))     56
```

В Python можна не вказувати явно, якого типу параметри чекає функція. З-за необхідності можна анотувати типи в разі параметрів за допомогою двокрапки, і стрілочкою (->), якщо треба вказати, якого типу значення має повернутися з функції. Однак, якщо передати навіть параметри інших типів, то код все одно виконується, тому що Python - це динамічна мова, і анотація типів покликана допомогти програмісту або його IDE відловити якісь помилки. Проте код все одно виконається.

```
def func_add(a:int, b:int) ->int:  
    return(a+b)  
  
print(func_add(5,6))          11  
print(func_add('5','6'))     56
```

Параметри можуть задаватися по їх імені - це називається *ключові параметри*. В цьому випадку для передачі аргументів функції використовується ім'я (ключ) замість позиції.

```
def func_family(name, family):  
    print("{} {}".format(name,family))  
  
func_family("Ivan", "Petrov")          Ivan Petrov!  
func_family(family="Petrov", name="Ivan")  Ivan Petrov!
```

Іноді буває потрібно визначити функцію, здатну приймати будь-яке число параметрів. Цього можна досягти за допомогою зірочок. Коли оголошується параметр із однією зірочкою (\*), то всі аргументи з цієї позиції і до кінця будуть зібрані в кортеж *tuple*. Аналогічно, коли оголошується параметр з двома зірочками (\*\*), то всі аргументи з цієї позиції і до кінця будуть зібрані в словник *dict*.

```
def func_twostars(*numbers, **keywords):  
    print("Arguments is the tuple")  
    print(type(numbers))  
    for i in numbers:
```

```

    print(i, end=" ")
    print("\nArguments is the dict")
    print(type(keywords))
    for key, value in keywords.items():
        print("{}: {}".format(key, value))

```

```
func_twostars(1,2,3,4,5,a=10, b=10)
```

```

Arguments is the tuple
<class 'tuple'>
1 2 3 4 5
Arguments is the dict
<class 'dict'>
b: 10
a: 10

```

### 4.3 Значення аргументів за замовчуванням

Частина параметрів функцій можуть бути необов'язковими і для них будуть використовуватися деякі задані значення за замовчуванням, якщо користувач не вкаже власних. Їх можна вказати, додавши до імені параметра у визначенні функції оператор присвоювання (=) з подальшим значенням.

Не рекомендується використовувати в якості значень за замовчуванням змінні типи: множини, словники та список.

*Важливо:* Значеннями за замовчуванням можуть бути забезпечені тільки параметри, що знаходяться в кінці списку параметрів. Таким чином, в списку параметрів функції параметр із значенням за замовчуванням не може передувати параметру без значення за замовчуванням.

```

def func_mult(x=5, y):
    return(x*y)

def func_mult(x, y=5):
    return(x*y)
print(func_mult("Ok"))
print(func_mult("Ok",1))

```

```

SyntaxError:      non-default
argument follows default
argument

```

```

OkOkOkOkOk
Ok

```

### 4.4 Глобальні та локальні змінні

При оголошенні змінних всередині визначення функції, вони жодним чином не пов'язані з іншими змінними з таким же ім'ям за межами функції - тобто імена змінних є локальними в функції. Це називається областю видимості змінної. Область видимості всіх змінних обмежена блоком, в якому вони оголошені, починаючи з точки оголошення імені. Не можна всередині функції змінювати змінні в глобальному контексті.

```

x = 50
def func(x):
    print("x =", x)

```

```
x = 2
print("Zamena local x na", x)
```

```
func(x)
print("x po pregnemy", x)
```

```
x = 50
Zamena local x na 2
x po pregnemy 50
```

Щоб привласнити деяке значення змінній, визначеної на вищому рівні програми (тобто не в якій-небудь області видимості, як то в функції або в класі), необхідно вказати Python, що її ім'я не локальне, а глобальне (global). Для цього використовується зарезервоване слово *global*.

```
x = 50
def func():
    global x
    print("x =", x)
    x = 2
    print("Zamena global x na", x)
```

```
func()
print("x stalo", x)
```

```
x = 50
Zamena global x na 2
x stalo 2
```

Є ще один тип області видимості, званий «нелокальний» (nonlocal) областю видимості, який є чимось середнім між локальною і глобальною областями видимості. Нелокальні області видимості застосовуються, коли визначають функції всередині функцій.

```
x = 50
x = 50
def out_func():
    x=5
    def in_func():
        nonlocal x
        x=2
        print("x in in_func() =", x)

    in_func()
    print("x in out_func()=", x)
```

```
out_func()
print("global x=", x)
```

```
x in in_func() = 2
x in out_func()= 2
global x= 50
```

В наведеному прикладі коли ми знаходимося всередині `in_func`, змінна `x`, визначена в першому рядку `out_func`, знаходиться ні в локальній області

видимості блоку `in_func`, ні в глобальному контексті. Ми оголошуємо, що хочемо використовувати саме цю змінну `x`, в такий спосіб: `nonlocal x`.

#### 4.5 Lambda-функція

Lambda-функція - це безіменна функція з довільним числом аргументів і обчислює один вираз. Тіло такої функції не може містити більше однієї інструкції (або виразу). Дану функцію можна використовувати в рамках будь-яких конвеєрних обчислень (наприклад всередині *filter()*, *map()* і *reduce()*) або самотійно, в тих місцях, де потрібно провести якісь обчислення, які зручно "загорнути" в функцію.

```
(lambda x: x ** 2) (5) 25
```

Lambda-функцію можна привласнити будь-якої змінній і надалі використовувати її в якості імені функції.

```
sqrt = lambda x: x ** 0.5  
sqrt (25) 5.0
```

Списки можна обробляти lambda-функціями всередині таких функцій, як *map()*, *filter()*, *reduce()*. Функція *map* приймає два аргументи, перший - це функція, яка буде застосована до кожного елементу списку, а другий - це список, який потрібно обробити.

```
l = [1, 2, 3, 4, 5, 6, 7]  
list (map (lambda x: x ** 3, l)) [1, 8, 27, 64, 125, 216, 343]
```

#### Завдання для самотійного виконання.

Виконайте завдання з практичних робіт 1 та 2 у вигляді функцій. В одній із функцій один з параметрів повинен бути завданням за замовчуванням.