Урок 4. Проектная деятельность. Функции: аргументы. Виды аргументов.

ФИО_____

Тест:

1. *Что вернёт функция без return?*
   A) 0 B) False C) None D) пустую строку
2. *Что выведет данная программа при вызове h?*

```
x = 1
def g():  1 usage
    print(x)
def h():  1 usage
    x = 2
    g()
```

_____

3. *Что делает “global” внутри функции?*

_____

4. *Что выведет данная программа?*

```
def foo(a, L=[]):
    L.append(a)
    return L

print(foo(1))
print(foo(2))
```

_____

5. *Что выведет данная программа?*

```
prefix = "Hello"
greeter = lambda name, p=prefix: f"{p}, {name}!"

print(greeter("Ann"))
```

_____

Теория:

Работа с аргументами функции:

- Позиционные

Этот тип встречается чаще всего. Значения передаются в функцию в том порядке, в каком указано в функции. Иначе возникнет ошибка. Если функция принимает несколько аргументов, нужно разделить их запятой.

```
def greet(first_name, last_name):
    print(f"Hello, {first_name} {last_name}!")

greet("John", "Black")
```

- Именованные

```
def greet(first_name, last_name):
    print(f"Hello, {first_name} {last_name}!")

greet(last_name="Black", first_name="John")  # Порядок не имеет
значение
```

- Аргументы по умолчанию

```
def greet(name="Guest"):
    print(f"Hello, {name}!")

greet()  # Вывод: Hello, Guest!
greet("Alice")  # Вывод: Hello, Alice!

# Подводные камни
def bad(buf=[]):              # ❌ один общий список между вызовами
    buf.append(1)
    return buf

def good(buf=None):           # ✅ создаём новый при каждом вызове
    if buf is None:
        buf = []
    buf.append(1)
    return buf
```

- Произвольное количество аргументов

Для передачи произвольного количества аргументов используются *args и **kwargs.

***args** позволяет передавать произвольное количество позиционных аргументов. Это полезно, когда вы не знаете заранее, сколько аргументов будет передано функции.

```
def add(*args):
    return sum(args)
```

```
result = add(1, 2, 3, 4)
print(result)  # Вывод: 10
```

В этом примере функция **add** может принимать любое количество аргументов и возвращать их сумму.

****kwargs** позволяет передавать произвольное количество именованных аргументов. Это полезно для создания функций с гибкими параметрами.

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=30, city="New York")
```

Здесь функция **print_info** принимает произвольное количество именованных аргументов и выводит их на экран.

Аннотации типов

Аннотация типов — это способ явно указать тип данных для переменной, аргумента функции или возвращаемого значения. Это не влияет на выполнение программы, но помогает статическим анализаторам кода проверять соответствие и выявлять потенциальные ошибки.

- Пишутся в сигнатуре: name: Type, -> ReturnType.
- **Не** проверяются интерпретатором автоматически — это подсказки для IDE.
- Базовые типы: int, float, str, bool, list[int], dict[str, int], tuple[int, ...], и т.д. .

Проверить аннотации можно через __annotations__:

```
def add(a: int, b: int) -> int:
    return a + b
print(add.__annotations__)  # {'a': <class 'int'>, 'b': <class
'int'>, 'return': <class 'int'>}
```

Сигнатура функции в Python — это формальное описание типов аргументов и типа возвращаемого значения функции.

Обрасти видимости: global, nonlocal

global – переменная должна быть взята из глобальной области видимости, а не создана локально**.**

```
count = 0
def inc():
   global count
   count += 1
def f():
   inc()
   print(count)
f()
```

nonlocal – Указывает, что переменная относится к ранее определенной переменной в ближайшей охватывающей, но не глобальной области видимости.

```
def make_counter(start=0):
    val = start
    def inc():
        nonlocal val
        val += 1
        return val
    return inc
```

Анонимные функции:

Лямбда-функции в Python являются анонимными. Это означает, что функция безымянна. Как известно, ключевое слов def используется в Python для определения обычной функции. В свою очередь, ключевое слово lambda используется для определения анонимной функции.

```
double = lambda x: x*2
print(double(5))

#Эквивалентно
def double(x):
   return x * 2
```

В вышеуказанном коде lambda x: x*2 — это лямбда-функция. Здесь x — это аргумент, а x*2 — это выражение, которое вычисляется и возвращается.

Эта функция безымянная. Она возвращает функциональный объект с идентификатором double. Сейчас мы можем считать её обычной функцией.

Задачи(для решение используйте разные виды аргументов):

1. **clip** — обрезка строки
   `Сигнатура: (text: str, max_len: int, suffix: str = "…") -> str:`
   Требования: вернуть строку длиной ≤ max_len; если длиннее — обрезать и добавить suffix.
   Если max_len < len(suffix), вернуть suffix[:max_len].
   Примеры: clip("hello", 5) -> "hello", clip("hello!", 5) -> "hell…".
2. **dot** — скалярное произведение
   `Сигнатура: (x: list[float], y: list[float]) -> float:`
   Требования: длины равны, иначе ValueError. Вернуть сумму x[i]*y[i].
   Пример: dot([1,2,3],[4,5,6]) -> 32.
3. **slice_str** — подстрока по параметрам
   `Сигнатура: (s: str, start: int = 0, end: int = None, step: int = 1) -> str:`

Требования: вернуть s[start:end:step]; при step == 0 — ValueError.

Примеры: slice_str("abcdef", 1, 5) -> "bcde", slice_str("abcdef", 5, 1, -2) -> "fd".

4. **append_safe —** безопасное добавление в список с None по умолчанию.

```
Сигнатура: (x: int, buf: list[int] | None = None) ->
list[int]:
```

Добавить число x в список buf. Если buf не передан (или None) — создать новый список.

Вернуть сам список (чтобы видно было, что между вызовами без buf не идёт «общий» список).

Пример:

a = append_safe(3) # [3]

b = append_safe(4) # [4] — отдельный список, не общий с a

ext = [1]; append_safe(2, ext) # [1,2], ext тоже стал [1,2]

5. **safe_div** — безопасное деление

```
Сигнатура: (a: float, b: float) -> float | None:
```

Требования: вернуть a/b; если b == 0, вернуть None (не кидать исключение).

Примеры: safe_div(6,3)->2.0, safe_div(1,0)->None.

6. **stats** — сумма и среднее с настройками

```
Сигнатура: (vals: list[float], scale: float = 1.0, round_to: int
|  None = None) -> tuple[float, float]:
```

Требования: sum_scaled = sum(vals)*scale, mean_scaled = (sum(vals)/len(vals))*scale.

Если round_to задан — округлить оба значения. Если vals пуст — ValueError.

Примеры: stats([1,2,3], 2) -> (12.0, 4.0), stats([1.2345,2.3456], 1, 2) -> (3.58, 1.79).

7. **calc** — мини-калькулятор с использованием анонимных функций (lambda)

```
Сигнатура: (a: float, b: float, op: str) -> float:
```

Выполнить одну из операций: '+', '-', '*', '/' (в делении ZeroDivisionError для b==0). Использовать анонимные функции в таблице операций.