Компьютерная математика II

Дисциплина для студентов 1-го курса специальности «Компьютерная математика и системный анализ» ММФ БГУ

Тема 4. Функция. Область видимости переменной

доц. Лаврова О.А., кафедра дифференциальных уравнений и системного анализа (ауд. 329)

март, 2024

4.1 Определение функции, вызов функции, объект функции

Функция является способом группирования операторов и выражений для многократного выполнения сгруппированного кода за счет вызовов функции в разных местах программы.

Для функции в коде возможны два действия: *определение функции* и *вызов функции*. Вызвать функцию можно только после ее определения.

Определение функции, операторы `def` и `return`

Определение функции осуществляется с помощью оператора `def` или с помощью выражения, содержащего зарезервированное имя `lambda`.

Оператор `def` является составным. Общий формат определения функции с помощью оператора `def`:

```
def <имя функции>(<имя_аргумента1>, <имя_аргумента2>, ...,
<имя_аргументаn>):
    <тело функции из операторов и выражений>
```

`def <имя функции>(<имя_аргумента1>, <имя_аргумента2>, ..., <имя_аргументаn>):` будем называть заголовком функции.

Важно отметить, что при определении функции код тела функции не выполняется, а выполняется только при вызове функции.

Результатом вызова функции является `<объект-значение>`, которое посылается посредством оператора `return` вызывающему коду в виде `return <объект-значение>`, или результатом вызова функции является объект `None`, если оператор `return` отсутствует в теле функции

In [1]:
```
def f(x):
    return x

def g(x):
    x = 10

f(1), g(1)
```

Out[1]:
```
(1, None)
```

При необходимости возвращения из функции нескольких объектов-значений их можно возвращать как элементы кортежа в виде `return obj1, obj2, obj3 ...`

In [2]:
```
def f(x):
    return "Return value:", x

f(1)
```

Out[2]:
```
('Return value:', 1)
```

Выход из тела функции осуществляется сразу после выполнения оператора `return` или при достижении конца тела функции, если оператор `return` отсутствует в теле функции

In [3]:
```
def f(x):
    return x
    print(f'{x = }')

def g(x, y):
    print(f'{x = }, {y = }')

f(x=1), g(y=2, x=1)
```

Out[3]:
```
x = 1, y = 2
(1, None)
```

Вызов функции

Вызов функции осуществляется с помощью выражения вида `<имя функции>(аргумент1, аргумент2, ..., аргументn)`, где `аргумент1, аргумент2, ..., аргументn` являются передаваемыми в функцию объектами-значениями, которые используются для выполнения кода, записанного в теле функции.

При каждом вызове функции тело функции выполняется заново.

При вызове функции все аргументы можно разделить на два типа: **позиционные аргументы** и **ключевые аргументы** (именованные, named, keyword).

Если при вызове функции значение аргумента задается только объектом, то он называется **позиционный аргумент**. Позиционные аргументы должны передаваться в том же порядке, в котором указаны при определении функции.

Ключевые аргументы при вызове указываются в виде пары `<ключ = объект>`, где *ключом является имя аргумента, указанное при определении функции*. Это позволяет при вызове функции указывать ключевые аргументы в любом порядке.

При вызове функции аргументы должны указываться в следующем порядке: сначала позиционные аргументы, затем ключевые аргументы:

In [4]:
```
print(1, 2, 3, sep=' between ')
```

```
1 between 2 between 3
```

Примеры различных вызовов функции:

In [5]:
```
def f(arg1, arg2, arg3):
    print(arg1, arg2, arg3)

f(1, 2, 3)                    # все аргументы позиционные
f(arg1=1, arg2=2, arg3=3)     # все аргументы ключевые
f(arg3=3, arg2=2, arg1=1)     # порядок ключевых аргументов не важен
f(1, 2, arg3=3)               # сначала позиционные аргументы, затем ключевой
```

```
1 2 3
1 2 3
1 2 3
1 2 3
```

Вызов функции с ключевыми аргументами является более информативным, чем вызов с позиционными аргументами.

Следующие вызовы функции будут некорректными:

```
def f(arg1, arg2):
    print(arg1, arg2)

f()
f(arg1=2, {1})
f('second', arg1='first')
f(3, arg3=4)
```

```
def f(arg1, arg2):
    print(arg1, arg2)

f()                         # должно быть два аргумента
f(arg1=2, {1})              # позиционный аргумент должен указываться до
ключевого арг.
f('second', arg1='first') # одному позиционному аргументу соответствуют
два значения
f(3, arg3=4)                # неизвестный ключевой аргумент
```

Передача значений аргументов при вызове функции

При вызове функции значения аргументов передаются путем *автоматического неявного присваивания* аргументам-переменным, указанным в заголовке функции при ее определении, соответствующих объектов-значений.

In [6]:
```
def f(arg1, arg2):
    ...

f(1, 2) # неявные присваивания arg1 = 1 и arg2 = 2
```

При этом в период выполнения функции аргументы-переменные будут ассоциироваться с пространством имен функции и являться **локальными переменными**. По завершению выполнения функции аргументы-переменные удаляются из памяти, а объекты-значения, на которые они ссылались, могут быть подвергнуты автоматической сборке мусора, если на них нету других ссылок.

Важно : При вызове функции копирования объектов, указанных в качестве значений аргументов, не происходит. Другими словами, значения аргументов функций в Python *передаются по ссылке, а не путем создания новых объектов*.

Это приводит к тому, что *изменяемые объекты* (списки, словари, множества), указанные в качестве значений аргументов при вызове функции, могут быть изменены внутри функции и, как следствие, оказать влияние на вызывающий код

In [7]:
```
def f(arg1):
    arg1.pop()

list1 = [1,2,3]; set1 = set(list1)
f(list1); f(set1)
print(list1, set1)
```

```
[1, 2] {2, 3}
```

Для того, чтобы не оказывать влияния на изменяемые объекты, передаваемые при вызове функции, необходимо *внутри функции создавать копию передаваемого объекта*:

In [8]:
```
def f(arg1):
    arg1 = arg1[:] # создание копии списка
    arg1.pop()

list1 = [1,2,3]
f(list1)           # вызов функции со списком в качестве аргумента
print(list1)
```

```
[1, 2, 3]
```

Режимы сопоставления объектов-значений и аргументов-переменных при вызове функции

При вызове функции возникает **задача сопоставления** передаваемых объектов-значений и аргументов функции, указанных в заголовке оператора `def` .

Позиционный режим сопоставления при *вызове функции* только с позиционными аргументами -- *автоматическое неявное присваивание* аргументам-переменным объектов-значений по позиции слева направо

In [9]:
```
def f(arg1, arg2, arg3, arg4, arg5):
    ...

f(1, 2, 3, 4, 5) # arg1 = 1, arg2 = 2, arg3 = 3, arg4 = 4, arg5 = 5
```

Позиционный режим сопоставления **с распаковкой последовательностей**: когда при *вызове функции* только с позиционными аргументами указывается символ `*` перед последовательностью с объектами-значениями, то происходит распаковка последовательности на элементы и *автоматическое неявное присваивание* объектов-значений из последовательности аргументам-переменным по позиции слева направо

In [10]:
```
def f(arg1, arg2, arg3, arg4, arg5):
    print(f'{arg1=}, {arg2=}, {arg3=}, {arg4=}, {arg5=}')

f(*range(1,6)) # arg1 = 1, arg2 = 2, arg3 = 3, arg4 = 4, arg5 = 5
# совмещение позиционных режимов с распаковкой и без распаковки
f(2, *"st", 5, 6)
```

```
arg1=1, arg2=2, arg3=3, arg4=4, arg5=5
arg1=2, arg2='s', arg3='t', arg4=5, arg5=6
```

Ключевой режим сопоставления, когда при *вызове функции* явно указывается связь между аргументом и передаваемым значением `аргумент = значение`

In [11]:
```
def f(arg1, arg2, arg3, arg4, arg5):
    print(f'{arg1=}, {arg2=}, {arg3=}, {arg4=}, {arg5=}')

f(arg5=1, arg3=2, arg1=3, arg2=4, arg4=5)
```

```
arg1=3, arg2=4, arg3=2, arg4=5, arg5=1
```

Ключевой режим сопоставления **с распаковкой словарей**: когда при *вызове функции* указывается `**` перед словарем с элементами `аргумент: значение`, то происходит *автоматическое неявное присваивание* аргументам, указанным ключами словаря, соответствующих объектов-значений из словаря

In [12]:
```
def f(arg1, arg2, arg3, arg4, arg5):
    print(f'{arg1=}, {arg2=}, {arg3=}, {arg4=}, {arg5=}')

myDict = {'arg2':2,'arg3':3,'arg5':5,'arg4':4}

f(**myDict, arg1=100)
```

```
arg1=100, arg2=2, arg3=3, arg4=4, arg5=5
```

Возможно совмещение ключевого и позиционного режимов сопоставления объектов-значений и аргументов-переменных при вызове функции.

In [13]:
```
def f(arg1, arg2, arg3, arg4, arg5):
    print(f'{arg1=}, {arg2=}, {arg3=}, {arg4=}, {arg5=}')

f(1, 2, 3, arg5=5, arg4=4)
```

```
arg1=1, arg2=2, arg3=3, arg4=4, arg5=5
```

Определение функции со значениями по умолчанию для аргументов

При определении функции в заголовке оператора **def**, помимо имени аргумента, можно указать его значение, например, `def <имя функции>(<имя_аргумента>=<значение>):` .

Значение аргумента функции, заданное при ее определении, называют *значением по умолчанию*, или **стандартным значением** аргумента. Аргумент, имеющий значение по умолчанию, является *необязательным* при вызове функции: если он отсутствует, функция будет выполнена со стандартным значением аргумента.

In [14]:
```
def f(arg1, arg2=2, arg3=3):
    return arg1, arg2, arg3

f(100), f(100, 0)
```

Out[14]:
```
((100, 2, 3), (100, 0, 3))
```

Использование символа равенства `=` при задании стандартного значения во время определения функции и при задании ключевого аргумента во время вызова функции НЕ является вызовом оператора присваивания.

Значения по умолчанию создаются только один раз при ОПРЕДЕЛЕНИИ функции. Использование изменяемых объектов в качестве значений по умолчанию могут вызвать проблемы при повторных вызовах одной и той же функции. Рекомендация: не используйте изменяемые объекты (списки, словари, множества) в качестве значений по умолчанию

In [15]:
```
def f(arg=[]):
    arg.append(1)
    return arg
print(f())
print(f())
```

```
[1]
[1, 1]
```

Все аргументы в заголовке функции, находящиеся после первого аргумента со стандартным значением, также должны иметь стандартные значения.

In [16]:
```
def f(arg1, arg2=2, arg3):
    return arg1, arg2, arg3

f(100,arg3=300), f(arg1=100)
```

```
  Cell In[16], line 1
    def f(arg1, arg2=2, arg3):
                        ^
SyntaxError: non-default argument follows default argument
```

Определение функции с произвольным количеством аргументов

Если при определении функции в заголовке оператора `def` указать символ `*` перед именем аргумента, например, `def <имя функции>(*<имя_аргумента>):` , то *при вызове* функции на месте этого аргумента может быть указано **произвольное количество позиционных аргументов**, которые *при выполнении* функции будут доступны как элементы *кортежа* `<имя_аргумента>` :

In [17]:
```
def f(*pargs):
    print(pargs)

f() # аргументы со * в определении являются необязательными при вызове!
f(1,2,3)
f(1,2,3,4,5,6)
```

```
()
(1, 2, 3)
(1, 2, 3, 4, 5, 6)
```

Пример функции с произвольным количеством позиционных аргументов

In [18]:
```
def adder(*pargs):
    """ применяет операцию сложения ко всем аргументам и возвращает результат опера
    s = pargs[0]
    for i in pargs[1:]:
        s += i
    return s
```

In [19]:
```
adder(1,2,3,4,5,6), adder('word1', 'word2', 'word3'), adder([1],[2],[3],[4])
```

Out[19]:
```
(21, 'word1word2word3', [1, 2, 3, 4])
```

Если при определении функции в заголовке оператора `def` указать `**` перед именем аргумента, например, `def <имя функции>(**<имя_аргумента>):` , то *при вызове* функции на месте этого аргумента может быть указано **произвольное количество ключевых аргументов**, которые *при выполнении* функции будут доступны как элементы *словаря* `<имя_аргумента>` :

In [20]:
```
def f(x,**kargs):
    print(kargs)

f(10) ## аргументы с ** в определении являются необязательными при вызове!
f(10, arg1=1, arg2=2, arg3=3)
f(10, x1=2, x2=3, x3=4, x5=5, x6=6)
```

```
{}
{'arg1': 1, 'arg2': 2, 'arg3': 3}
{'x1': 2, 'x2': 3, 'x3': 4, 'x5': 5, 'x6': 6}
```

Порядок следования аргументов при определении функции

При определении функции аргументы должны указываться в следующем порядке: позиционные аргументы, далее форма `*pargs` , затем аргументы со стандартными значениями, затем аргументы с передачей только по ключу и, наконец, форма `**kargs` :

In [21]:
```
def f(a, *pargs, b=50, c, **kargs):
    print(f'{a=}, {pargs=}, {b=},  {c=}, {kargs=}')

f(1, 2, 3, 4, 5, c=0, x=1, y=2)
```

```
a=1, pargs=(2, 3, 4, 5), b=50,  c=0, kargs={'x': 1, 'y': 2}
```

```
print(*args, sep=' ', end='\n', file=None, flush=False)
```

В Python 3.X при определении функции аргументы можно указывать также в следующем порядке: позиционные аргументы, затем аргументы со стандартными значениями, далее форма `*` , затем аргументы с передачей только по ключу и, наконец, форма `**kargs` :

In [22]:
```
def f(a, b=5, *, c, **kargs):
    print(f'{a=}, {b=}, {c=}, {kargs=}')

f(1, c=10, x=1, y=2)
```

```
a=1, b=5, c=10, kargs={'x': 1, 'y': 2}
```

Такой синтаксис означает, что при вызове фукции все аргументы, следующие за символом `*` , должны быть заданы ключевыми.

```
FuncAnimation(fig, func, frames=None, init_func=None, fargs=None,
save_count=None, *, cache_frame_data=True, **kwargs)
```

Объект функции

При выполнении оператора `def` происходит *автоматическое неявное присваивание* объекта функции имени функции. Имя функции становится ссылкой на объект функции.

In [23]:
```
def f(x):
    ...
    return "function f is called"
f
```

Out[23]:
```
<function __main__.f(x)>
```

Объект функции имеет тип `function`

In [24]:
```
f.__class__
```

Out[24]:
```
function
```

С помощью атрибута `__code__` объекта функции можно получить, в частности, информацию об аргументах функции

In [25]:
```
f.__code__,f.__code__.co_argcount, f.__code__.co_varnames
```

Out[25]:
```
(<code object f at 0x000001B7BB12E5B0, file "C:\Users\Olga\AppData\Local\Temp\ipyke
rnel_15828\3391401968.py", line 1>,
 1,
 ('x',))
```

В дальнейшем в коде объекты функций можно присваивать другим переменным, передавать другим функциям в качестве аргумента, использовать в коллекциях и т.д. Обращаться с объектами функций можно так же, как и с объектами встроенных типов.

In [26]:
```
g = f;
g(f), [1, f, "string"]
```

Out[26]:
```
('function f is called', [1, <function __main__.f(x)>, 'string'])
```

Аннотация функции

Аннотация типов -- это информация о типах данных для аргументов функции и для значений функции, которая указывается при определении функции в строке заголовка оператора `def` . Синтаксис определения функции с использованием аннотаций:

```
def <имя функции>(<имя_аргумента1> [: type1], <имя_аргумента2> [: type2],
..., <имя_аргументаN> [: typeN]) [-> typeN+1]:
    <тело функции из операторов и выражений>
```

In [27]:
```
def conc(s1: str, s2: str = 'second') -> str:
    return ' and '.join([s1,s2])
```

In [28]:
```
conc('first')
```

Out[28]:
```
'first and second'
```

Аннотации хранятся в атрибуте `__annotations__` объекта функции в виде словаря. Ключами словаря являются строковые объекты с именами атрибутов, а также строковый объект `'return'` . Значениями словаря являются имена типов, указанные в аннотациях.

In [29]:
```
conc.__annotations__
```

Out[29]:
```
{'s1': str, 's2': str, 'return': str}
```

Аннотации НЕ имеет никакого эффекта на выполнение функции и являются не обязательными при определении функции.

В качестве аннотаций может быть указано ЛЮБОЕ выражение, которое может быть в дальнейшем использовано

In [30]:
```
def compile(source: "something compilable",
            filename: "where the compilable thing comes from",
            mode: "is this a single statement or a suite?"):
    ...
```

4.2 Строки документации

Строки документации в Python -- это строковый литерал, который документирует функциональность *функции*, *модуля*, *класса* или *метода класса*, его спецификацию. Строки документации являются атрибутом объекта, для которого они определены.

Комментарии -- это текст внутри программы, документирующий функциональность не объекта, а фрагментов кода. Комментарии начинаются с символа `#` и заканчиваются концом текущей строки. При выполнении кода комментарии игнорируются интерпретатором.

Документация в Python -- это совокупнсть строк документации для объектов и комментариев для фрагментов кода. Документация является важной частью хорошо написанных программ на Python.

Строки документации *автоматически* записываются в атрибут `__doc__` объекта при его создании

In [31]:
```
import math
print(math.__doc__, math.floor.__doc__)
```

```
This module provides access to the mathematical functions
defined by the C standard. Return the floor of x as an Integral.

This is the largest integer <= x.
```

PyDoc — это инструмент для извлечения строк документации. Интерфейсом является вызов функции `help` с указанием объекта в качестве аргумента функции

In [32]:
```
help(math.ceil)
```

```
Help on built-in function ceil in module math:

ceil(x, /)
    Return the ceiling of x as an Integral.

    This is the smallest integer >= x.
```

In [33]:
```
# help(math)
```

Строки документации можно получить с помощью средств оболочки блокнота Jupyter Notebook, вводя символ знака вопроса `?` до или после имени объекта:

In [34]:
```
?math.dist
```

```
Signature: math.dist(p, q, /)
Docstring:
Return the Euclidean distance between two points p and q.

The points should be specified as sequences (or iterables) of
coordinates.  Both inputs must have the same dimension.

Roughly equivalent to:
    sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))
Type:      builtin_function_or_method
```

Правила оформления строк документации

PEP 257 — это руководство по стандартам для оформления строк документации на Python, см. https://www.python.org/dev/peps/pep-0257/ (2001 год).

Синтаксически строки документации представляют собой многострочный строковый литерал, который записывается в начале определения объекта (функции, модуля, класса, метода класса) ПЕРЕД исполняемым кодом.

Многострочные строковые литералы создаются с помощью утроенных кавычек, составленных из двойных кавычек:

In [35]:
```
def f():
    """Do ... and return ..."""
    ...
```

Пустые строки до и после строк документации оставлять нельзя.

Строки документации могут состоять только из одной строки или из нескольких строк.

Рекомендованная форма для *однострочных строк документации*: `"""Делает ... и возвращает ..."""`

In [36]:
```
def f():
    """Do ... and return ..."""
    ...
```

Многострочные строки документации состоят из:

- строки вида: `"""делает ...` ,
- пустая строка,
- развернутое описание функции,
- закрывающие кавычки `"""` в отдельной строке.

В развернутом описании указываются аргументы, ограничения на аргументы, необязательные аргументы, возвращаемые значения, обработка исключений. Каждый аргумент описывается в отдельной строке.

In [37]:
```
def f(t):
    """Do ...

    Arguments :

    t : ...

    Returns : ...
    """
    ...
```

Строковый литерал записывается в атрибут `__doc__` объекта функции при его создании:

In [38]:
```
f.__doc__
```

Out[38]:
```
'Do ... \n     \n     Arguments :\n     \n     t : ...\n     \n     Returns : ...\n     '
```

4.3 Встроенная функция вывода `print`

`print` в Python 3.X является встроенной функцией, в более ранних версиях `print` был оператором.

Функция `print` обычно вызывается в отдельной строке, так как она возвращает `None`.

In [39]:
```
print({1})
1, print({2})
```

Out[39]:
```
{1}
{2}
(1, None)
```

В общем случае функции, которые возвращают `None`, рассматриваются как разновидности операторов.

Назначение и синтаксис функции `print`

```
print(obj1, obj2, ...[, sep=' '] [, end='\n'][, file=sys.stdout][,
flush=False])
```

Функция `print` выводит в поток данных `file` строковые представления одного или большего числа объектов `obj1`, `obj2`, ..., которые при выводе разделяются строкой `sep`. За строковыми представлениями объектов `obj1`, `obj2`, ... следует строка `end`. Буферизированный вывод сбрасывается или нет согласно значению аргумента `flush`.

In [40]:
```
print(1, [1], {1: 1}, sep='\t', end=' end')
```

```
1       [1]     {1: 1} end
```

```
print(obj1, obj2, ...[, sep=' '] [, end='\n'][, file=sys.stdout][,
flush=False])
```

По умолчанию функция `print` записывает произвольные объекты в *стандартный поток вывода* `sys.stdout`, *автоматически* добавляя к объектам некоторое форматирование, разделяя объекты символом пробела и добавляя в конце строки вывода управляющий символ перехода на новую строку. При вызове функции `print` **преобразовывать объекты в строки не требуется** в отличие от методов записи файловых объектов `write` и `writelines`. При вызове функции `print` без аргументов в поток стандартного вывода записывается символ новой строки `\n`.

In [41]:
```
print(1, {1}, {1:1})
print()
```

```
1 {1} {1: 1}
```

```
print(obj1, obj2, ...[, sep=' '] [, end='\n'][, file=sys.stdout][,
flush=False])
```

В Python 3.X указание аргумента `file` при вызове функции `print` неявно вызывает метод `write` файлового объекта, указанного в качестве значения аргумента `file`, для строковых представлений объектов `obj1`, `obj2`, ...

In [42]:
```
file = open("tmp.txt",'a')
print(1, [1], file=file)
file.close()
```

Последующие вызовы функции `print` продолжают выводить объекты в стандартный поток вывода

In [43]:
```
print('print in sys.stdout')
```

```
print in sys.stdout
```

4.4 lambda-функция

Определение функции можно осуществлять с помощью выражения, содержащего `lambda`.

Общий формат определения выражения `lambda`-функции:

```
lambda <арг1>, <арг2>, ..., <аргn>: <ОДНО выражение, зависящее от
аргументов>
```

Тело `lambda`-функции является одиночным выражением, т.е. НЕ может содержать операторы. Например, оператор `if` внутри тела `lambda`-функции использовать нельзя. Выражение `lambda`-функции возвращает объект функции. Имя функции не создается, на объект функции нет ссылок в коде.

In [44]:
```
def f(x):                # объект функции создан и неявно присвоен переменной f
    return (x-5)**2

lambda x: (x-5)**2       # объект функции создан
```

Out[44]:
```
<function __main__.<lambda>(x)>
```

Тело `lambda` -функции похоже на то, что указывается в операторе `return` внутри тела функции оператора `def` .

Тело `lambda` -функции может быть ТОЛЬКО выражением, поэтому `lambda` -функция менее универсальна, чем функция, определенная с помощью оператора `def` . `lambda` -функция используется в основном для определения простых функций небольшого размера.

Так как `lambda` -функция является выражением, а не оператором, она может находиться в местах, где оператор `def` не разрешен синтаксисом языка Python, например, внутри коллекций или в качестве аргументов при вызове функции:

In [45]:
```
[lambda x: (x-5)**2, lambda x: abs(x)] # объект функции как элемент коллекции
```

Out[45]:
```
[<function __main__.<lambda>(x)>, <function __main__.<lambda>(x)>]
```

In [46]:
```
list(map((lambda x: 1), range(10))) # объект функции как аргумент функции
```

Out[46]:
```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

`lambda` -функцию полезно определять тогда, когда она используется однократно. Чаще всего `lambda` -функции используются со встроенными функциями `map` , `filter` , `sorted` , `min` , `max` и др.

In [47]:
```
z = ['abC','BCA']
sorted(z), sorted(z, key=lambda x: x.lower())
```

Out[47]:
```
(['BCA', 'abC'], ['abC', 'BCA'])
```

In [48]:
```
l = list(range(10))
l.sort(key=lambda x: x%2)
l
```

Out[48]:
```
[0, 2, 4, 6, 8, 1, 3, 5, 7, 9]
```

Так как `lambda` -функция является выражением, это позволяет совместить *определение функции* и *применение функции* в одной строке кода.

4.5 Типы областей видимости переменной

При ссылке на переменную в коде, когда для переменной явно не указывается объект, которому переменная принадлежит, осуществляется поиск значения переменной в четырех возможных лексических областях видимости:

- **локальная** область видимости (**L**ocal),
- областИ видимости **объемлющих функций** (**E**nclosing),
- **глобальная** область видимости (**G**lobal),
- **встроенная** область видимости (**B**uilt-in).

Правило лексических областей LEGB означает, что переменная ищется сначала в области **L**ocal, затем в областях **E**nclosing от внутренней объемлющей функции до наружной, затем в **G**lobal и, наконец, в области **B**iult-in.

Поиск останавливается в первой области видимости, где обнаруживается переменная. Если в результате поиска переменную найти не удалось, то возникает ошибка типа `NameError`.

In [49]:
```
new_variable
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[49], line 1
----> 1 new_variable

NameError: name 'new_variable' is not defined
```

При ссылке на переменную в коде, когда для переменной явно указывается объект, которому переменная принадлежит, например, `obj.X`, значение переменной `X` извлекается из объекта `obj` без поиска переменной `X` в областях видимости **LEGB**.

Локальная область видимости функции

Переменная расположена в локальной области видимости функции, если она ассоциирована с пространством имен функции, т.е. переменной выполнено *присваивание внутри функции*. В локальную область видимости попадают также имена аргументов функции.

Локальные переменные функции — это переменные, расположенные в локальной области видимости некоторой функции. Локальные переменные видимы только операторам и выражениям внутри тела функции. Использование локальных переменных за пределами тела функции невозможно.

Локальные переменные существуют в памяти только в период, когда функция *выполняется*. По завершению выполнения функции локальные переменные автоматически удаляются из памяти, а объекты, на которые они ссылаются, могут быть подвергнуты сборке мусора, если на них нет ссылок где-то в другом месте.

Локальная переменная не конфликтует с переменными за пределами функции (в модуле, в другой функции), даже если переменные имеют одинаковые имена:

In [50]:
```
x = 1 # поиск только в областях видимости G и B

def f(x):
    x = 2 # локальная переменная x
    return x

x, f(x), x
```

Out[50]:
```
(1, 2, 1)
```

Кроме локальной области видимости функции в Python есть другие локальные области видимости:

- локальная область видимости в выражениях включений,
- локальная область видимости в операторах `class` (Тема 8) и др.

In [51]:
```
[i**2 for i in range(10)]
i
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[51], line 2
      1 [i**2 for i in range(10)]
----> 2 i

NameError: name 'i' is not defined
```

Глобальная область видимости

Переменная расположена в глобальной области видимости, если она ассоциирована с пространством имен модуля верхнего уровня, т.е. переменной выполнено *присваивание внутри модуля*.

Глобальные переменные — это переменные, расположенные в глобальной области видимости модуля верхнего уровня.

Глобальная область видимости охватывает только одиночный файл с модулем верхнего уровня. Имена переменных распределяются по модулям. Модуль должен явно импортироваться, чтобы появилась возможность работы с именами, определенными в его файле. В Python нет понятия единственной всеобъемлющей глобальной области видимости.

В функциях можно ссылаться на переменные из области видимости объемлющих функций и из глобальной области видимости, но изменять их значения нельзя:

In [52]:
```
x = 1

def f():
    return x+1
x, f()
```

Out[52]:
```
(1, 2)
```

Встроенные функции `globals` и `locals` возвращают глобальные и локальные переменные, соответственно, в виде словаря, доступные в текущей области видимости

In [53]:
```
x = 1

def f(a):
    print(locals())
    return x+a
x, f(1)
```

Out[53]:
```
{'a': 1}
(1, 2)
```

Операторы объявления `global` и `nonlocal`

Операторы объявления `global` и `nonlocal` используются только внутри функций.

В функциях можно использовать переменные из глобальной области видимости, но изменять их значения нельзя. Для изменения таких переменных внутри функции они должны быть *объявлены внутри функции* как *глобальные переменные* с помощью оператора `global` :

In [54]:
```
x = 1

def f():
    global x, y  # переменная x объявлена глобальной
    x, y = 2, 3

x, f(), x, y
```

Out[54]:
```
(1, None, 2, 3)
```

Оператор `global` *объявляет*, что функция планирует *изменять* одну или более переменных, которые существуют в пространстве имен модуля, содержащего эту функцию, или переменные будут созданы внутри функции, как новые переменные модуля.

В функциях можно использовать переменные из областей видимости объемлющих функций, но изменять их значения нельзя. Для изменения таких переменных внутри функции они должны быть *объявлены* внутри функции как **нелокальные переменные** с помощью оператора `nonlocal` (появился в Python 3.x):

In [55]:
```
x = 1

def f_out():
    x = 2
    def f_in():
        nonlocal x  # переменная x объявлена нелокальной
        x = 3
    f_in()
    return x

x, f_out(), x
```

Out[55]:
```
(1, 3, 1)
```

Переменная, объявленная глобальной с помощью оператора `global`, ищется сразу в глобальной области видимости **G**lobal, затем во встроенной области видимости **B**uilt-in.

Переменная, объявленная нелокальной с помощью оператора `nonlocal`, ищется ТОЛЬКО в объемлющих областях видимости **E**nclosing, нарушая правило поиска **LEGB**.

Старайтесь избегать использования глобальных переменных внутри пользовательских функций. Обменивайтесь данными с помощью передаваемых значений аргументов и возвращаемых значений функций. Использование глобальных переменных делает код более сложным для понимания и многократного применения, чем код, состоящий из функций, которые используют только на локальные переменные.

Встроенная область видимости

Встроенная область видимости состоит из имен встроенных исключений, из имен встроенных функций (например, `abs`, `len`, `min`) и из имен переменных (`True`, `False`, `None`, `Ellipses`, `NotImplemented`). Имена из встроенной области видимости видны ВЕЗДЕ в коде Python.

Все имена из встроенной области видимости перечислены в модуле `builtins`

In [56]:
```
import builtins
len(dir(builtins))
```

Out[56]:
```
161
```

Обратите внимание, что при переопределении встроенного имени предупреждение будет выдаваться не всегда:

In [57]:
```
print(abs(-5))
abs = 1
```

```
5
```

In [58]:
```
abs(-5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[58], line 1
----> 1 abs(-5)

TypeError: 'int' object is not callable
```