Программирование на Python

Базовый курс. Часть 2. Функции

Андрей Скороходов
udv|group 2023

Python. Функции

Функция – объект, принимающий аргументы и возвращающий значение.

Функции позволяют многократно использовать повторяющийся код, упростить код, избежать ошибок.

```
price = 1000
vat_rate = 20

if price < 0 or vat_rate < 0:
    print("Некорректные входные данные.")

vat = (price * vat_rate) / 100
print("Сумма НДС:", vat)

price = 200
vat_rate = 18

if price < 0 or vat_rate < 0:
    print("Некорректные входные данные.")

vat = (price * vat_rate) / 100
print("Сумма НДС:", vat)

price = 2120
vat_rate = 33

if price < 0 or vat_rate < 0:
    print("Некорректные входные данные.")

vat = (price * vat_rate) / 100
print("Сумма НДС:", vat)
```

✓ 0.0s

```
Сумма НДС: 200.0
Сумма НДС: 36.0
Сумма НДС: 699.6
```

С использованием функций

```
def calculate_vat(price, vat_rate):
    if price < 0 or vat_rate < 0:
        print("Некорректные входные данные.")
        return None

    vat = (price * vat_rate) / 100
    print("Сумма НДС:", vat)

    return vat

vat_rate = 20

vat_amount = calculate_vat(price=1000, vat_rate=20)
vat_amount = calculate_vat(price=200, vat_rate=18)
vat_amount = calculate_vat(price=2120, vat_rate=33)
```

✓ 0.0s

```
Сумма НДС: 200.0
Сумма НДС: 36.0
Сумма НДС: 699.6
```

Python. Встроенные функции

Python имеет огромный набор встроенных функций, например:

Функция	Описание
print()	Выводит данные на стандартный вывод (консоль).
len(iterable)	Возвращает длину (количество элементов) итерируемого объекта, такого как список, кортеж или строка.
type(object)	Возвращает тип объекта, например, int, str, list, и так далее.
input(prompt)	Считывает строку с консоли, часто используется для ввода данных пользователем.
range([start], stop[, step])	Создает последовательность чисел от start до stop (не включая stop) с заданным шагом step.
sum(iterable)	Возвращает сумму всех элементов итерируемого объекта, например, сумму всех элементов списка.
sorted(iterable[, key=функция, reverse=булево])	Возвращает новый список, отсортированный в порядке возрастания (или убывания). Может использовать функцию key для более сложных сортировок.
dir([object])	Возвращает список всех атрибутов и методов объекта (или текущего модуля, если объект не указан).
help([object])	Выводит справочную информацию об объекте (или текущем модуле, если объект не указан).
open(file, mode)	Открывает файл и возвращает объект файла, который может использоваться для чтения ('r'), записи ('w'), добавления ('a') и других операций над файлами.

```
s = 'Hello world'
print(s)
print(f"len(s): {len(s)}")
print(f"abs(-11): {abs(-11)}")
print(f'any[list]: {any([True, False, False, False])}')
print(f'max[list]: {max([1, 10, 2.5, 8])}')
```

✓ 0.0s

```
Hello world
len(s): 11
abs(-11): 11
any[list]: True
max[list]: 10
```

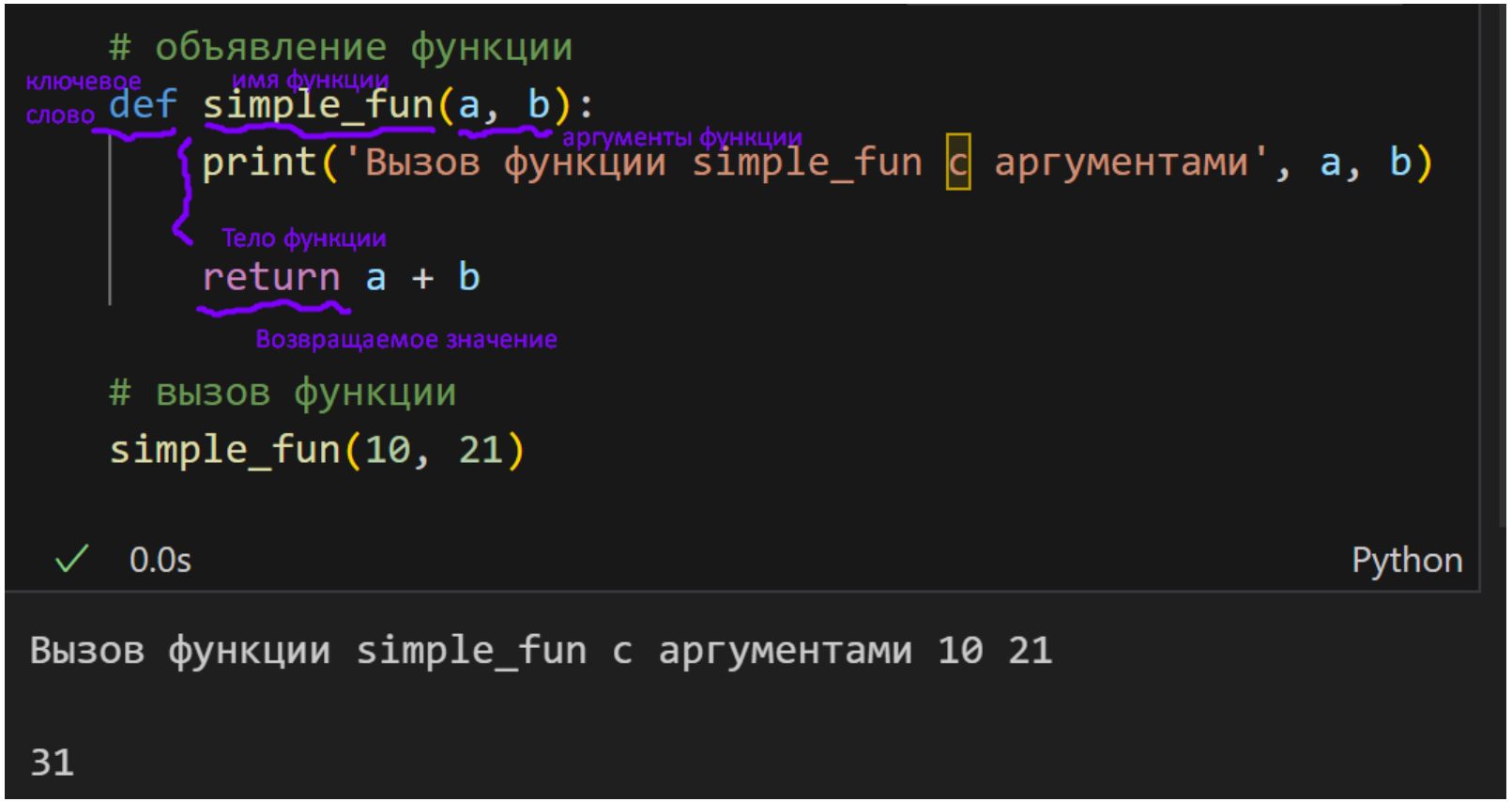
```
dir(s)
```

✓ 0.0s

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
...
```

Python. Пользовательские функции

Пример пользовательской функции



Python. Аргументы функции

Аргументы:

- Позиционные
- Именованные

Сначала в функцию передаются позиционные аргументы, а затем – именованные.

В случае с именованными аргументами порядок перечисления может не совпадать

```
def calculate_vat(price, discount=10, vat=20):
    ...

calculate_vat(1234)
calculate_vat(1234, discount=20)
calculate_vat(1234, vat=30)
calculate_vat(1234, vat=30, discount=20)
```

В функцию можно передавать неограниченное количество аргументов, при этом позиционные обозначаются ***agrs**, а именованные ****kwargs**, при этом ***args** распаковываются в кортеж, а ****kwargs** – в словарь.

Оператор * отвечает за распаковку итерируемого объекта:

```
lst = ['a', 'b', 'c', 'd']
print(lst[0], lst[1], lst[2], lst[3])
print(*lst)
```

✓ 0.0s

```
a b c d
a b c d
```

```
def calculate_vat(*args, **kwargs):
    print('*args:', [arg for arg in args])
    print('**kwargs:', {key: value for key, value in kwargs.items()})

calculate_vat('1', 1234)
calculate_vat('2', 1234, discount=20)
calculate_vat('3', 1234, vat=30)
calculate_vat('4', 1234, vat=30, discount=20)
```

✓ 0.0s

```
*args: ['1', 1234]
**kwargs: {}
*args: ['2', 1234]
**kwargs: {'discount': 20}
*args: ['3', 1234]
**kwargs: {'vat': 30}
*args: ['4', 1234]
**kwargs: {'vat': 30, 'discount': 20}
```

Python. Возвращаемое значение

Функция всегда возвращает значение, это может быть в т.ч. пустое значение (None).

Ключевое слово для возвращения значения – оператор **return**. Таких операторов в теле функции может быть несколько, после выполнения оператора **return** исполнение функции прекращается.

Возвращаемых значений может быть несколько, для этого нужно перечислить значения через запятую после оператора **return**. В этом случае функция будет возвращать кортеж (tuple):

```python
def calculate_vat(price, vat_rate=20):
    if price < 0 or vat_rate < 0:
        print("Некорректные входные данные.")
        return None

    vat = (price * vat_rate) / 100
    print("Сумма НДС:", vat)

    return vat, price

result = calculate_vat(price=1000)
print(result, type(result))
vat_amount, price = calculate_vat(price=1000)
print(vat_amount, price)
```

✓ 0.0s

```
Сумма НДС: 200.0
(200.0, 1000) <class 'tuple'>
Сумма НДС: 200.0
200.0 1000
```

```python
def calculate_vat(price, vat_rate=20):
    if price < 0 or vat_rate < 0:
        print("Некорректные входные данные.")
        return None

    vat = (price * vat_rate) / 100
    print("Сумма НДС:", vat)

    return vat

vat_rate = 20

vat_amount = calculate_vat(price=1000)
vat_amount = calculate_vat(price=200, vat_rate=18)
vat_amount = calculate_vat(price=2120, vat_rate=33)
vat_amount = calculate_vat(price=-2120, vat_rate=33)
```

✓ 0.0s

```
Сумма НДС: 200.0
Сумма НДС: 36.0
Сумма НДС: 699.6
Некорректные входные данные.
```

Python. Область видимости

Область видимости переменных:

- Локальная
- Глобальная

Переменные, объявленные в теле функции, создаются во время выполнения функции и остаются видимыми только внутри этой функции. Для доступа к глобальным области видимости необходимо использовать ключевое слово **global**, в этом случае локальная переменная создаваться не будет, а будет меняться значение глобальной переменной. Использование глобальных переменных усложняет анализ кода и в целом не рекомендуется.

```
var1 = 'value_1='

def modify_value(value):
    return value*2

print('Возвращаемое значение: ', modify_value(var1))
print('Исходное значение: ', var1)
```

✓ 0.0s

```
Возвращаемое значение:  value_1=value_1=
Исходное значение:  value_1=
```

```
var1 = 'value_1='

def modify_value(value):
    var1 = value*2
    return var1

print('Возвращаемое значение: ', modify_value(var1))
print('Исходное значение: ', var1)
```

✓ 0.0s

```
Возвращаемое значение:  value_1=value_1=
Исходное значение:  value_1=
```

```
var1 = 'value_1='

def modify_value(value):
    global var1
    var1 = value*2
    return var1

print('Возвращаемое значение: ', modify_value(var1))
print('Исходное значение: ', var1)
```

✓ 0.0s

```
Возвращаемое значение:  value_1=value_1=
Исходное значение:  value_1=value_1=
```

Python. Изменяемые аргументы

Если в качестве аргумента функции передается изменяемый параметр, то есть риск этот параметр изменить.

Следует очень аккуратно взаимодействовать с изменяемыми аргументами, т.к. они могут привести к неожиданному поведению алгоритма.

```python
lst = ['a', 'b', 'c']

def modify_list(lst):
    lst1 = lst
    lst1.append('new')
    return lst1

print('Возвращаемое значение: ', modify_list(lst))
print('Исходное значение: ', lst)

lst = ['a', 'b', 'c']

def create_modify_list(lst):
    lst1 = lst[:]
    lst1.append('new')
    return lst1

print('Возвращаемое значение: ', create_modify_list(lst))
print('Исходное значение: ', lst)
```

✓ 0.0s

```
Возвращаемое значение:  ['a', 'b', 'c', 'new']
Исходное значение:  ['a', 'b', 'c', 'new']
Возвращаемое значение:  ['a', 'b', 'c', 'new']
Исходное значение:  ['a', 'b', 'c']
```

Python. Задание 1

1. Создайте функцию *calculate_bmi()*, которая рассчитывает индекс массы тела (ИМТ, bmi). На вход функция будет принимать вес (*weight: float*) в килограммах и рост (*height: float*) в метрах.
2. Предусмотрите проверку отрицательных, нулевых и нечисловых значений для аргументов функции. В случае некорректных входных аргументов необходимо вывести на экран, например «Вес и/или рост имеют некорректные значения!», и остановить выполнение функции.
3. ИМТ – отношение массы человека к квадрату его роста. Функция должна возвращать значение ИМТ и выводить его на экран.

Python. Задание 2

1. Создайте функцию *group_types()*, которая будет принимать на вход любое количество позиционных аргументов и группировать их по типу. Аргументы могут быть типов *str*, *int*, *float*, другие типы игнорируются.
2. Функция должна возвращать словарь, в котором ключи – это типы данных, а значения – отсортированные по убыванию значения соответствующих типов.

```
group_types(4, 2, 3, 5.0, 6.5, 99.2, 'cat', 'dog', 'bee')
✓ 0.0s
{'int': [4, 3, 2], 'float': [99.2, 6.5, 5.0], 'str': ['dog', 'cat', 'bee']}
```

Python. Задание 3

1. Создайте функцию *print_shopping_list()*, которая будет принимать на вход имя (*name: str*) человека и список товаров (*shopping_list: dict{<название товара>: <количество>}*), а также любое количество именованных аргументов.
2. В случае если среди именованных аргументов существует аргумент «*hour*» (час дня), а среди покупок попалось «*пиво*», нужно проверить значение аргумента «*hour*». Если оно больше либо равно 23, необходимо вывести на экран сообщение о том, что данная покупка не может быть совершена, после чего прекратить выполнение функции.
3. В противном случае нужно вывести на экран сообщение «*<Имя человека>*, купи: *<список товаров>*», где *<список товаров>* – список товаров, в котором каждый товар находится на новой строчке, каждая строчка должна быть пронумерована.

```
print_shopping_list('Андрей', {'яйца': 10, 'хлеб': 1, 'молоко': 2})
✓ 0.0s

Андрей , купи:
1. яйца: 10
2. хлеб: 1
3. молоко: 2


print_shopping_list('Андрей', {'яйца': 10, 'хлеб': 1, 'пиво': 2}, hour=23)
✓ 0.0s

Слишком поздно для пива
```

Python. Задание 4

1. Для словаря *employees* из предыдущей части напишите функцию *change_currency()*, которая будет переводить зарплату работников в доллары.
2. В качестве параметров функции нужно передать данные сотрудника (“*employee*” – значение ключа словаря *employees*) валюту (“*currency*”) и обменный курс (“*exchange_rate*”), новое значение зарплаты будет вычисляться как текущее значение *salary*, умноженное на *exchange_rate*.
3. Для того, чтобы понимать в какой валюте работник получает зарплату, нужно добавить в словарь каждого сотрудника ключ *“currency”,* изначально равный *“rub”*. При смене валюты значение ключа меняется на соответствующую валюту (например, “*usd*”).
4. Также напишите функцию *show_salary()*, которая будет принимать на вход элемент словаря *employees* и выводить на экран сообщение формата: {должность} {фамилия} получает {зарплата} {валюта}.

```
for key, value in employees.items():
  employees[key] = change_currency(value, 'usd', 0.01)
  show_salary(key, employees[key])
```

```
Инженер Иванов получает  978.30 usd
Управляющий Петров получает  982.98 usd
Электрик Сидоров получает  935.10 usd
Рабочий Прохоров получает  455.36 usd
```

Python. Задание 5

1. Для словаря *employees* из задания 4 напишите функцию *add_employee()*, которая будет добавлять сотрудников в исходный словарь.
2. В качестве обязательных параметров в функцию нужно передать исходный словарь (*employees: dict*), имя (*name: str*), фамилию (*surname: str*), должность (*position: str*), возраст (*age: int*), в качестве необязательных параметров – зарплату (*salary: float, по умолчанию 50000*) и валюту (*currency: str, по умолчанию “rub”*), в которой выплачивается зарплата.
3. Функция должна добавлять сотрудника в исходный словарь в принятом для словаря формате. Если добавляемый сотрудник уже существует в исходном словаре, следует вывести на экран предупреждение и прекратить выполнение функции.
4. Также напишите функцию *fire_employee()*, которая будет удалять сотрудника по его имени и фамилии. Если указанный сотрудник отсутствует, следует вывести на экран предупреждение и прекратить выполнение функции.

Python. Рекурсивная функция

Рекурсивная функция – функция, вызывающая сама себя.

Рекурсивные функции обычно работают медленнее итеративных.

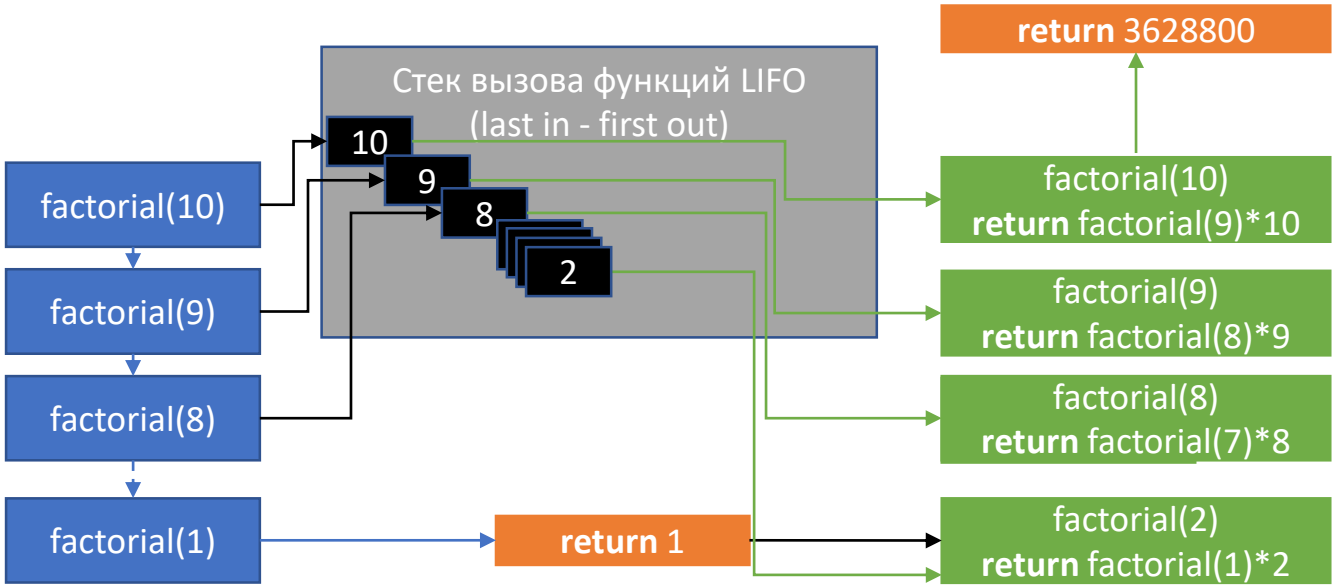
Рассмотрим, как работает рекурсивная функция, на примере вычисления факториала.

В рекурсивной функции должно быть определено условие завершения (**граничное условие**), иначе функция будет вызывать сама себя бесконечно, в нашем примере это *if n == 1*.

```
def factorial(n):
    print("Вычисление факториала, n = ", n)
    if n == 1:
        return n
    else:
        return n*factorial(n-1)

factorial(10)
```

✓ 0.0s

```
Вычисление факториала, n =  10
Вычисление факториала, n =  9
Вычисление факториала, n =  8
Вычисление факториала, n =  7
Вычисление факториала, n =  6
Вычисление факториала, n =  5
Вычисление факториала, n =  4
Вычисление факториала, n =  3
Вычисление факториала, n =  2
Вычисление факториала, n =  1

3628800
```

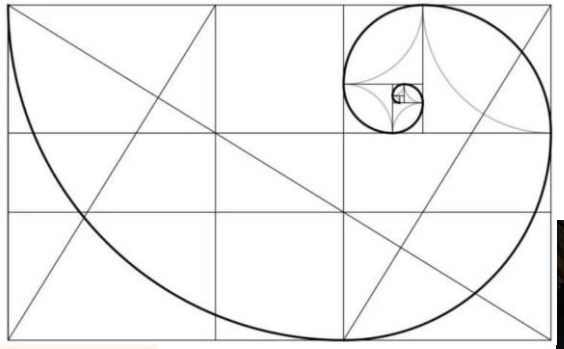


Рекурсия в Python имеет ограничение в 1000 слоев. Этот параметр можно изменить:

```
from sys import setrecursionlimit
setrecursionlimit(2000) # увеличивает лимит до 2000 вызовов
```

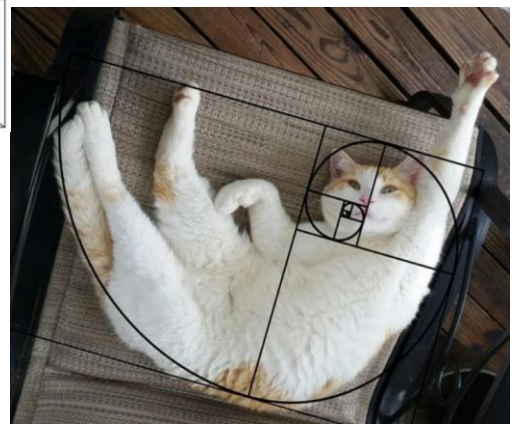
Python. Задание 6

1. Напишите рекурсивную функцию *fibonacci()*, которая возвращает число Фибоначчи для соответствующего порядкового номера. Числа Фибоначчи – это ряд чисел, в котором каждое следующее число равно сумме двух предыдущих: 1, 1, 2, 3, 5, 8, 13… *Каждое число из ряда Фибоначчи, разделенное на предыдущее, имеет значение, стремящееся к уникальному показателю, которое составляет 1,618. Первые числа ряда Фибоначчи не дают настолько точное значение, однако по мере нарастания, соотношение постепенно выравнивается и становится все более точным.*
2. В качестве параметров функция принимает порядковый номер числа Фибоначчи (*n: int*).
3. Функция возвращает число Фибоначчи.



Как вы считаете, является ли повсеместное применение числа Фибоначчи в природе совпадением или свидетельством наличия некоего вселенского разума? Давайте попробуем обсудить этот вопрос в нашем Telegram-чате.

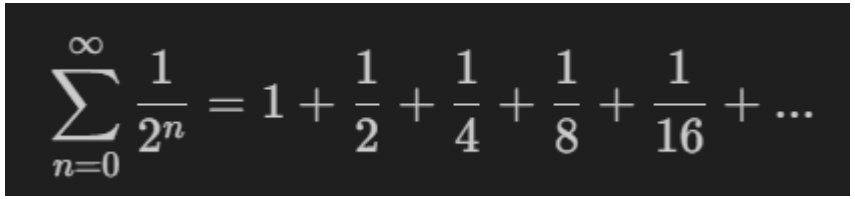
Используя основные принципы ряда Фибоначчи, растут семечки в центре подсолнуха, движется спираль ДНК, был построен Парфенон и написана самая знаменитая картина в мире — «Джоконда» Леонардо Да Винчи.



https://hi-news.ru/science/chislo-fibonachchi-pochemu-ono-tak-populyarno-v-prirode.html#i-2

Python. Задание 7

1. Напишите рекурсивную функцию *geometric_progression_sum()* для вычисления суммы *n* первых членов бесконечно убывающей геометрической прогрессии.



2. В качестве параметра функция принимает количество членов прогрессии (*n: int*).
3. Функция возвращает сумму первых членов прогрессии.

Python. Побочные эффекты и чистые функции

Детерминированность функции — это свойство, при котором функция всегда возвращает один и тот же результат для одних и тех же входных данных без побочных эффектов, связанных с состоянием программы.

Побочный эффект (side effects) — любые взаимодействия с внешней средой: изменения глобальных переменных, операции с файлами, прием данных по сети, вывод в консоль.

Чистые функции (pure functions) — функции, которые при вызове не влияют на состояние программы и не имеют побочных эффектов:

- Проще для тестирования. Результаты их работы можно легко предсказать и проверить.
- Более безопасны. Поскольку они не изменяют состояние программы, то не могут вызвать неожиданные побочные эффекты или ошибки в других частях программы.
- Легче поддаются оптимизации. Поскольку они не имеют побочных эффектов, их можно безопасно кэшировать или выполнять в многопоточной среде.

Python. Функции высшего порядка

Функции высшего порядка принимают одну (или более) функций в качестве аргументов и/или в качестве результата возвращают функцию.

В Python существуют 3 встроенные функции высшего порядка: **map(), filter() и reduce():**

- **map(function, iter) –** применяет функцию ко всем элементам итерируемого объекта.
- **filter(condition, iter) –** отбирает элементы итерируемого объекта согласно условию.
- **reduce(function, iter, [, initial]) –** применяет функцию двух аргументов кумулятивно к элементам итерируемого объекта, необязательно начиная с начального аргумента. (для использования необходим импорт модуля functools). По сути функция уменьшает итерируемый объект до одного значения.

```
def some_func(n):
    return n**2 + 10

numbers = (1, 2, 3, 4)
result = map(some_func, numbers)
print(list(result))
```

✓ 0.0s

```
[11, 14, 19, 26]
```

```
def some_condition(value):
    if value > 3:
        return True

    return False

numbers = (1, 2, 3, 4, 5, 6, 7, 8)
result = filter(some_condition, numbers)
print(list(result))
```

✓ 0.0s

```
[4, 5, 6, 7, 8]
```

```
from functools import reduce

def some_func(x, y):
    return x * y

numbers = (1, 2, 3, 4, 5, 6)
result = reduce(some_func, numbers)
print(result)
```

✓ 0.0s

```
720
```

Python. Лямбда-функции

Лямбда-функция – анонимная функция с неограниченным количеством элементов, но только с одним выражением, которое вычисляется и возвращается.

lambda arguments: expression

Лямбда-функции можно не объявлять и использовать, например, в качестве аргументов для функций высшего порядка (**map(), filter(), reduce()**) или для других операций с итерируемыми объектами, например, для сортировки словарей.

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

print(double(10), hex(id(double)), type(double))

double_lambda = lambda x: x*2

print(double_lambda(10), hex(id(double_lambda)), type(double_lambda))
```

✓ 0.0s

```
20 0x15886e48550 <class 'function'>
20 0x15886e480d0 <class 'function'>
```

```
dict1 = {
    (0, 0): {'parameter1': 10, 'parameter2': 21, 'parameter3': 30},
    (0, 1): {'parameter1': 20, 'parameter2': 23, 'parameter3': 30},
    (1, 1): {'parameter1': 30, 'parameter2': 2, 'parameter3': 30},
    (1, 2): {'parameter1': 40, 'parameter2': 0, 'parameter3': 30},
    (2, 1): {'parameter1': 50, 'parameter2': 10, 'parameter3': 30},
}


sorted(list(dict1.items()), key=lambda x: x[1]['parameter2'])
```

✓ 0.0s

```
[((1, 2), {'parameter1': 40, 'parameter2': 0, 'parameter3': 30}),
 ((1, 1), {'parameter1': 30, 'parameter2': 2, 'parameter3': 30}),
 ((2, 1), {'parameter1': 50, 'parameter2': 10, 'parameter3': 30}),
 ((0, 0), {'parameter1': 10, 'parameter2': 21, 'parameter3': 30}),
 ((0, 1), {'parameter1': 20, 'parameter2': 23, 'parameter3': 30})]
```

```
numbers = (1, 2, 3, 4)
result = map(lambda x: x**2+10, numbers)
print(list(result))
```

✓ 0.0s

```
[11, 14, 19, 26]
```

```
numbers = (1, 2, 3, 4, 5, 6, 7, 8)
result = filter(lambda x: x > 3, numbers)
print(list(result))
```

✓ 0.0s

```
[4, 5, 6, 7, 8]
```

Python. Задание 8

1. При помощи конструкции List Comprehension и модуля random создайте список *lst* из 1 миллиона случайных целочисленных значений.
2. При помощи функции map() и лямбда-выражений сгенерируйте список *lst1*, в котором каждый элемент будет равен квадратному корню соответствующего значения списка *lst*.
3. Выполните пункт 2, используя цикл for
4. Выполните пункт 2, используя List Comprehension
5. Сравните время выполнения формирования списка *lst1*, для этого можно использовать модуль datetime (datetime.datetime.now() – получение текущего времени в формате datetime, времена можно вычитать), выведите на экран время выполнения для каждого варианта.
6. При помощи функции filter() и лямбда-выражений сгенерируйте список *lst2*, в котором будут только те элементы списка *lst*, которые больше среднего значения списка *lst*.

Python. Задание 9

1. Выведите на экран список, состоящий из сотрудников, находящихся в словаре *employees* из предыдущих заданий, отсортированный по размеру зарплаты. Элемент списка – кортеж (ключ, значение), соответствующий конкретному сотруднику.

Python. Замыкание функции

Замыкание (closure) функции – концепция, в которой вложенная функция имеет доступ к локальным переменным функции более высокого порядка, после того, как внешняя функция уже завершила свою работу.

Вложенная функция

```
def print_some_text(text):
    print("Вызов функции print_some_text")

    def add_important_information():
        print("Ниже будет напечатан какой-то текст:")
        print(text)

    add_important_information()


print_some_text('Текст основной функции')
```

[219] ✓ 0.0s

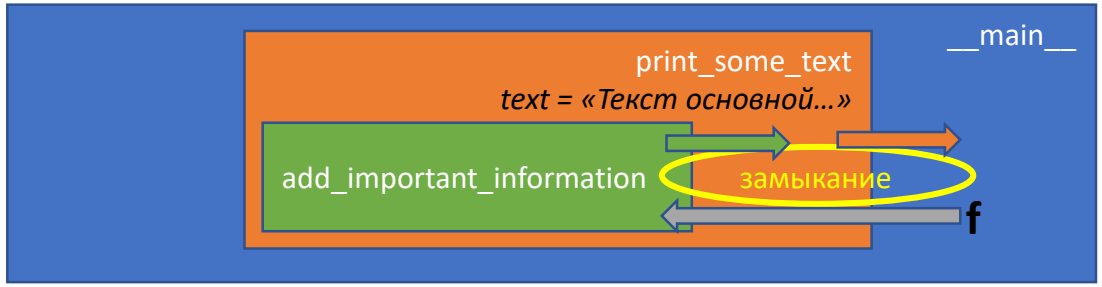
```
Вызов функции print_some_text
Ниже будет напечатан какой-то текст:
Текст основной функции
```

Замыкание функции

```
def print_some_text(text):
    print("Вызов функции print_some_text")

    def add_important_information():
        print("Ниже будет напечатан какой-то текст:")
        print(text)


    return add_important_information

f = print_some_text('Текст основной функции')
print(f)
f()
```

✓ 0.0s

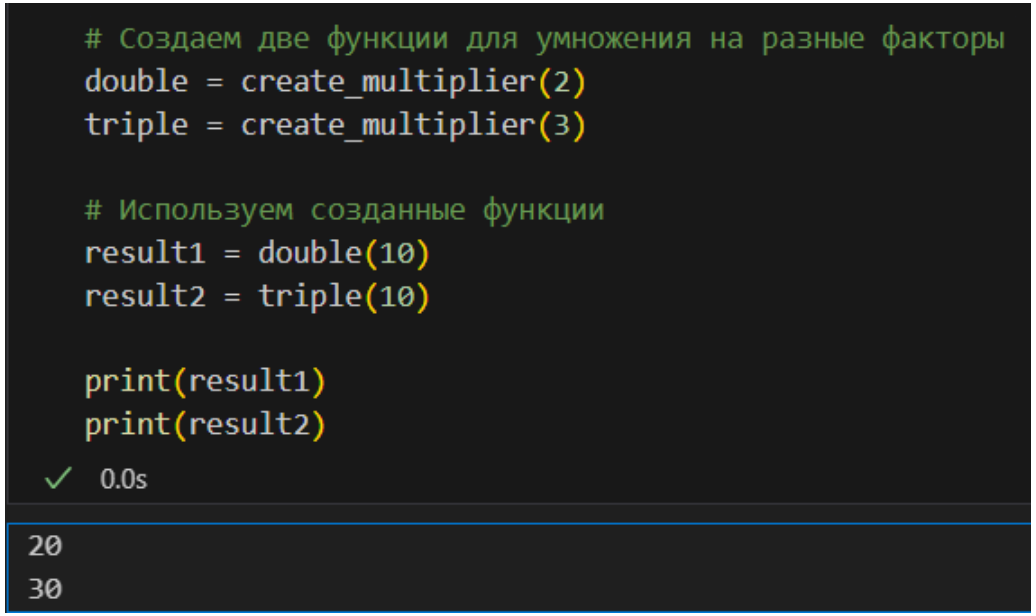
```
Вызов функции print_some_text
<function print_some_text.<locals>.add_important_information at 0x0000015884C91280>
Ниже будет напечатан какой-то текст:
Текст основной функции
```



Замыкания полезны, например, когда вам нужно создать функции с долгоживущими переменными или когда вы хотите скрыть некоторые данные от внешнего кода, делая их доступными только внутри функции.

Python. Задание 10

1. Создайте функцию *create_multiplier(factor)*, которая принимает на вход число, которое будет являться множителем.
2. Внутри функции *create_multiplier()* создайте функцию *multiplier(x)*, которая принимает на вход число, которое нужно умножить на множитель.
3. Используя концепцию замыкания функций объявите два объекта *double* и *triple*, которые будут ссылаться на функцию *create_multiplier*, но передавать в нее разные аргументы: 2 и 3 соответственно.
4. Присвойте значения функций *double* и *triple* переменным *result1* и *result2* соответственно. В функции *double* и *triple* в качестве аргумента передайте одинаковое число, например, 10.



Python. Декораторы

Декораторы – функции, которые позволяют изменить поведение произвольной функции без изменения ее кода.

Принцип работы декоратора основан на принципах механизме замыкания функций. Т.е. в функцию-декоратор передается функция, которую нужно декорировать: decorator(function). Для упрощения синтаксиса декоратор обычно обозначается с использованием «@» и указывается непосредственно перед объявлением функции. Для функции можно использовать несколько декораторов, при этом они указываются друг под другом, порядок декорирования важен.

В указанном примере декоратор сохраняет значение времени до запуска декорируемой функции и вычитает его из текущего времени при завершении работы декорируемой функции, что позволяет вычислить время работы функции. При этом декорируемая функция выполняет свой алгоритм без изменений.

```
import time

def some_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        print(f"Время работы функции: {time.time() - start_time: .2f} s")

        return result
    return wrapper
```

Декоратор

```
@some_decorator
def some_function(name, args):
    print(f"Выполнение '{name}' с аргументами '{args}'...")
    time.sleep(5)
    return 'Результат работы функции some_function'
```

Функция, которую нужно расширить

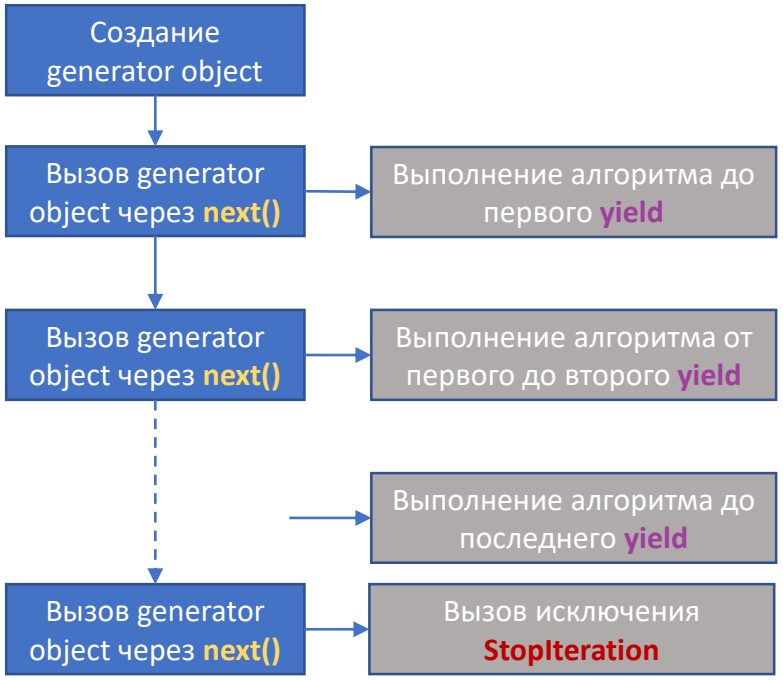
```
# some_function = some_decorator(some_function) -> @some_decorator


result = some_function('Имя функции', 'Аргумент')
result
```

вызов функции

```
Выполнение 'Имя функции' с аргументами 'Аргумент'...
Время работы функции:  5.00 s
'Результат работы функции some_function'
```

Python. Генераторы

Генератор – объект, который сразу при создании не вычисляет значения всех своих элементов, а хранит в памяти только последний вычисленный элемент, правило перехода к следующему и условие, при котором выполнение прерывается. Вычисление следующего значения происходит лишь при выполнении метода **next(),** код выполняется до следующего оператора yield. Предыдущее значение при этом теряется. Генератор уменьшает потребление памяти и ускоряет процесс обработки




```
a = generator_example(4)
next(a)
next(a)
next(a)
next(a)
```

0.0s

```
=== Вызов функции generator_example ===
Обработка итерации i = 1, count = 1
Обработка итерации i = 2, count = 2
Обработка итерации i = 3, count = 3

----------------------------------------
StopIteration                             Trace
C:\Users\ASKORO~1\AppData\Local\Temp/ipykernel_
      3 next(a)
      4 next(a)
----> 5 next(a)

StopIteration:
```

```
def generator_example(m):
    count = 1
    print('=== Вызов функции generator_example ===')

    for i in range(1, m):
        print(f'Обработка итерации i = {i}, count = {count}')
        yield i**2 + count
        count += 1

a = generator_example(8)

print("Генератор возвращает значение только при выполнении метода next()")
for i in a:
    print("Вычисленное значение: ", i)
```

0.0s

```
Генератор возвращает значение только при выполнении метода next()
=== Вызов функции generator_example ===
Обработка итерации i = 1, count = 1
Вычисленное значение:  2
Обработка итерации i = 2, count = 2
Вычисленное значение:  6
Обработка итерации i = 3, count = 3
Вычисленное значение:  12
Обработка итерации i = 4, count = 4
Вычисленное значение:  20
Обработка итерации i = 5, count = 5
Вычисленное значение:  30
Обработка итерации i = 6, count = 6
Вычисленное значение:  42
Обработка итерации i = 7, count = 7
Вычисленное значение:  56
```

```
generator = generator_example(4)
generator
```

0.0s

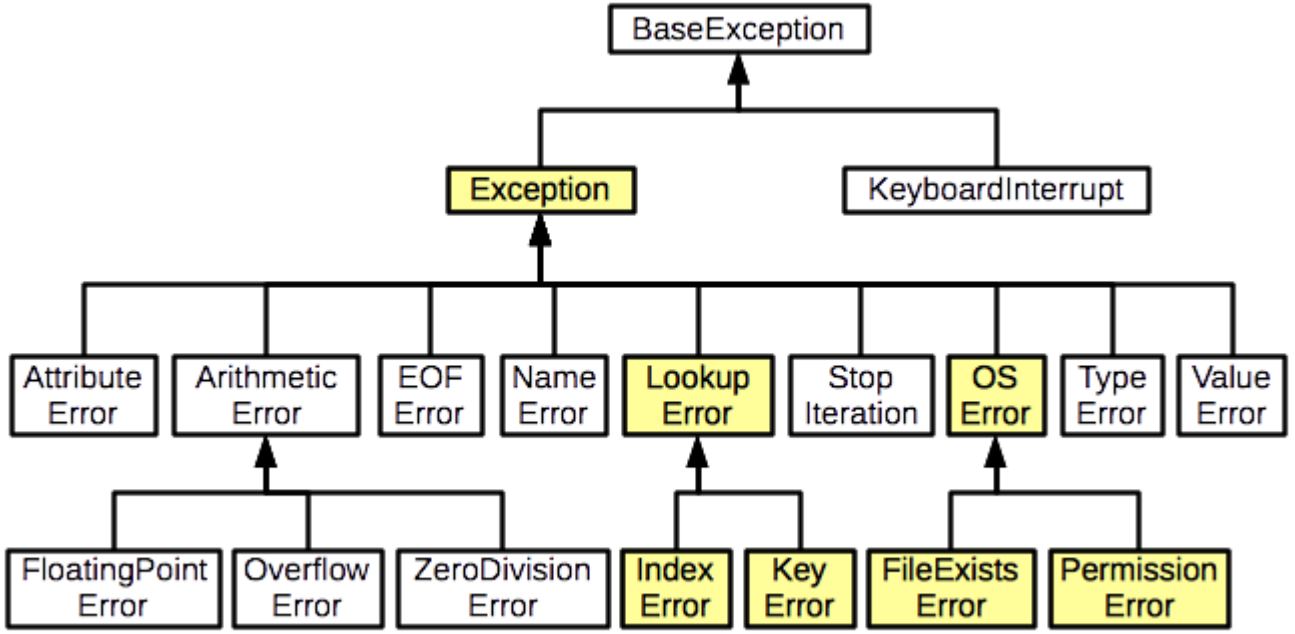
```
<generator object generator_example at 0x0000015886EC3BA0>
```

Python. Задание 11

1. Реализуйте вывод на экран последовательности Фибоначчи при помощи генератора fibonacci_gen().
2. Для вывода на экран воспользуйтесь циклом for или while, количество выводимых цифр будет определяться количеством итераций в цикле.
3. Напоминалка: последовательность Фибонначи состоит из таких чисел, где каждое последующее число равняется сумме двух предыдущих. Первое и второе число равняется единице: 1, 1, 2, 3, 5, 8, 13

Python. Исключения (class Exception)

Во время выполнения программы могут возникать различные ошибки, которые в Python вызывают сработку так называемых исключений – специального типа данных, в котором передается тип ошибки, ее описание и трейс вызова инструкции, вызвавшей ошибку.



https://betacode.net/11421/python-exception-handling

Если возникшее исключение никак не обработать, это приведет к завершению выполнения программы.

Python. Обработка исключений

Конструкция try-except нужна для обработки исключений. Базовый синтаксис конструкции:

try:
    <код, в котором может произойти ошибка>
except *<один тип исключения>:*
    <код, который выполнится при возникновении ошибки>
except <другой тип исключения>**:**
    <код, который выполнится при возникновении ошибки>
else:
    <код, который выполнится, если ошибки не было>
finally:
    <код, который выполнится в любом случае> В функции выполняется до оператора return!

После выполнения этих инструкций программа продолжает работу.

- В блоке **except** можно обрабатывать несколько типов исключений, например, except (ValueError, ZeroDevisionError)
- Блоки **except** обрабатываются последовательно, по аналогии с конструкцией **if-elif-elif-else.**
- Исключения имеют иерархию, например BaseException->Exception->ArithmeticError->ZeroDivisionError. Блок **except ArithmeticError** будет перехватывать все ошибки, связанные с арифметическими ошибками. Если мы поставим после него блок **except ZeroDivisionError**, то он уже не выполнится, т.к. выполнится except ArithmeticError**.**
- Если после **except** не указывать никакое исключение, то блок будет перехватывать абсолютно все исключения.
- Для получения исключения в виде объекта можно использовать конструкцию except *<исключение>* as *<имя переменной>*. Указанная переменная будет ссылаться на экземпляр исключения, это может пригодится, например, для вывода исключения в консоль

Python. Распространение исключений

Распространение исключений (propagation exceptions) – механизм, при котором полученное исключение распространяется на все уровни вызова программы.

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
C:\Users\ASKORO~1\AppData\Local\Temp/ipykernel_54132/3163494175.py in <module>
     10 print('1')
     11 print('2')
---> 12 print(func3())
     13 print('3')

C:\Users\ASKORO~1\AppData\Local\Temp/ipykernel_54132/3163494175.py in func3()
      6
      7 def func3():
----> 8     func2()
      9
     10 print('1')

C:\Users\ASKORO~1\AppData\Local\Temp/ipykernel_54132/3163494175.py in func2()
      3
      4 def func2():
----> 5     func1()
      6
      7 def func3():

C:\Users\ASKORO~1\AppData\Local\Temp/ipykernel_54132/3163494175.py in func1()
      1 def func1():
----> 2     return 1/0
      3
      4 def func2():
      5     func1()

ZeroDivisionError: division by zero
```

```python
def func1():
    return 1/0

def func2():
    func1()

def func3():
    func2()

print('1')
print('2')
print(func3())
print('3')
```

```python
def func1():
    try:
        return 1/0
    except:
        return 0

def func2():
    try:
        func1()
    except:
        return 0

def func3():
    func2()

print('1')
print('2')
print(func3())
print('3')
```

```
1
2
None
3
```

Можно обрабатывать исключения на любом уровне стека вызова функций.

Python. Генерация исключений

В коде можно самостоятельно генерировать исключения при помощи инструкции **raise**:

```
:
  1 def some_function(a):
  2     for i in range(a):
  3         if i==3:
  4             raise AttributeError('<Новое описание ошибки>')
  5
  6 some_function(10)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
C:\Users\ASKORO~1\AppData\Local\Temp/ipykernel_54132/1171433506.py in <module>
      4             raise AttributeError('<Новое описание ошибки>')
      5
----> 6 some_function(10)

C:\Users\ASKORO~1\AppData\Local\Temp/ipykernel_54132/1171433506.py in some_function(a)
      2     for i in range(a):
      3         if i==3:
----> 4             raise AttributeError('<Новое описание ошибки>')
      5
      6 some_function(10)

AttributeError: <Новое описание ошибки>
```

Также можно создавать собственные классы исключений, которые должны наследоваться от класса BaseException.

Python. Задание 12

1. Напишите собственную функцию get_value_by_key() для получения значения из словаря по ключу.
2. Функция должна принимать на вход словарь (dictionary), ключ (key) и значение (def_value, по умолчанию None), которое будет возвращаться, если ключ в словаре отсутствует. Т.е. вести себя схожим образом со стандартной функцией dict.get ().
3. В случае, если ключ есть, то функция возвращает его значение.
4. В случае, если на вход функции в качестве dictionary передается не словарь, в консоль должно выводиться сообщение о недопустимом значении функции, а функция должна возвращать None.
5. В случае, если ключ key в словаре отсутствует, функция должна выводить в консоль сообщение, что такого ключа нет, но все равно возвращать значение, указанное в def_value.

Python. Задание 13

1. Напишите функцию check_value(), которая будет проверять соответствие переданного ей значения определенным критериям и две функции check_number(), check_str(), в которых будет выполняться обработка исключений.
2. На вход функция check_value() будет принимать значение (value), а также неопределенное количество именованных аргументов.
3. В случае, если на вход пришло число и среди именованных аргументов есть аргумент limits (список из верхней и нижней границы разрешенного диапазона), должна вызываться функция check_number().
4. В случае если на вход пришла строка и среди именованных аргументов есть аргумент lenght (максимальная длина строки), должна вызываться функция check_str().
5. Функции check_str() и check_number() должны проверять строку и число на соответствие длине и диапазону соответственно. При превышении максимальной длины строки или выходе числа за диапазон функции должны генерировать исключение AttributeError.
6. При возникновении исключения в функции check_value(), исключение нужно обработать и вывести его текст в консоль.
7. В любом случае в конце выполнения функции она должна выводить в консоль сообщение о завершенной проверке, значение value и его тип.