

Лабораторная работа 2.8 Работа с функциями в языке Python

Цель работы: приобретение навыков по работе с функциями при написании программ с помощью языка программирования Python версии 3.x.

Ход работы

Функции в программировании

Функция в программировании представляет собой обособленный участок кода, который можно вызывать, обратившись к нему по имени, которым он был назван. При вызове происходит выполнение команд тела функции.

Функции можно сравнить с небольшими программками, которые сами по себе, т. е. автономно, не исполняются, а встраиваются в обычную программу. Нередко их так и называют – подпрограммы. Других ключевых отличий функций от программ нет. Функции также при необходимости могут получать и возвращать данные. Только обычно они их получают не с ввода (клавиатуры, файла и др.), а из вызывающей программы. Сюда же они возвращают результат своей работы.

Существует множество встроенных в язык программирования функций. С некоторыми такими в Python мы уже сталкивались. Это `print()`, `input()`, `int()`, `float()`, `str()`, `type()`. Код их тела нам не виден, он где-то "спрятан внутри языка". Нам же предоставляется только интерфейс – имя функции.

С другой стороны, программист всегда может определять свои функции. Их называют пользовательскими. В данном случае под "пользователем" понимают программиста, а не того, кто пользуется программой. Разберемся, зачем нам эти функции, и как их создавать.

Предположим, надо три раза подряд запрашивать на ввод пару чисел и складывать их. С этой целью можно использовать цикл:

```
i = 0
while i < 3:
    a = int(input())
    b = int(input())
    print(a+b)
    i += 1
```

Однако, что если перед каждым запросом чисел, надо выводить надпись, зачем они нужны, и каждый раз эта надпись разная. Мы не можем прервать цикл, а затем вернуться к тому же циклу обратно. Придется отказаться от него, и тогда получится длинный код, содержащий в разных местах одинаковые участки:

```
print("Сколько бананов и ананасов для обезьян?")
a = int(input())
b = int(input())
print("Всего", a+b, "шт.")

print("Сколько жуков и червей для ежей?")
a = int(input())
b = int(input())
print("Всего", a+b, "шт.")
```

```
print("Сколько рыб и моллюсков для выдр?")
a = int(input())
b = int(input())
print("Всего", a+b, "шт.")
```

Пример исполнения программы:

```
Сколько бананов и ананасов для обезьян?
15
5
Всего 20 шт.
Сколько жуков и червей для ежей?
50
12
Всего 62 шт.
Сколько рыб и моллюсков для выдр?
16
8
Всего 24 шт.
```

Внедрение функций позволяет решить проблему дублирования кода в разных местах программы. Благодаря им можно исполнять один и тот же участок кода не сразу, а только тогда, когда он понадобится.

Определение функции. Оператор `def`

В языке программирования Python функции определяются с помощью оператора `def`.

Рассмотрим код:

```
def countFood():
    a = int(input())
    b = int(input())
    print("Всего", a+b, "шт.")
```

Это пример определения функции. Как и другие сложные инструкции вроде условного оператора и циклов функция состоит из заголовка и тела. Заголовок оканчивается двоеточием и переходом на новую строку. Тело имеет отступ.

Ключевое слово `def` сообщает интерпретатору, что перед ним определение функции. За `def` следует имя функции. Оно может быть любым, также как и всякий идентификатор, например, переменная. В программировании весьма желательно давать всему осмысленные имена. Так в данном случае функция названа "посчитатьЕду" в переводе на русский.

После имени функции ставятся скобки. В приведенном примере они пустые. Это значит, что функция не принимает никакие данные из вызывающей ее программы. Однако она могла бы их принимать, и тогда в скобках были бы указаны так называемые параметры.

После двоеточия следует тело, содержащее инструкции, которые выполняются при вызове функции. Следует различать определение функции и ее вызов. В программном коде они не рядом и не вместе. Можно определить функцию, но ни разу ее не вызвать. Нельзя вызвать функцию, которая не была определена. Определив функцию, но ни разу не вызвав ее, вы никогда не выполните ее тела.

Вызов функции

Рассмотрим полную версию программы с функцией:

```
def countFood():
    a = int(input())
    b = int(input())
    print("Всего", a+b, "шт.")

print("Сколько бананов и ананасов для обезьян?")
countFood()

print("Сколько жуков и червей для ежей?")
countFood()

print("Сколько рыб и моллюсков для выдр?")
countFood()
```

После вывода на экран каждого информационного сообщения осуществляется вызов функции, который выглядит просто как упоминание ее имени со скобками. Поскольку в функцию мы ничего не передаем скобки опять же пустые. В приведенном коде функция вызывается три раза.

Когда функция вызывается, поток выполнения программы переходит к ее определению и начинает исполнять ее тело. После того, как тело функции исполнено, поток выполнения возвращается в основной код в то место, где функция вызывалась. Далее исполняется следующее за вызовом выражение.

В языке Python определение функции должно предшествовать ее вызовам. Это связано с тем, что интерпретатор читает код строки за строкой и о том, что находится ниже по течению, ему еще неизвестно. Поэтому если вызов функции предшествует ее определению, то возникает ошибка (выбрасывается исключение `NameError`):

```
print("Сколько бананов и ананасов для обезьян?")
countFood()

print("Сколько жуков и червей для ежей?")
countFood()

print("Сколько рыб и моллюсков для выдр?")
countFood()

def countFood():
    a = int(input())
    b = int(input())
    print("Всего", a+b, "шт.")
```

Результат:

```
Сколько бананов и ананасов для обезьян?
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    countFood()
NameError: name 'countFood' is not defined
```

Для многих компилируемых языков это не обязательное условие. Там можно определять и вызывать функцию в произвольных местах программы. Однако для удобочитаемости кода программисты даже в этом случае предпочитают соблюдать определенные правила.

Функции придают программе структуру

Польза функций не только в возможности многократного вызова одного и того же кода из разных мест программы. Не менее важно, что благодаря им программа обретает истинную структуру. Функции как бы разделяют ее на обособленные части, каждая из которых выполняет свою конкретную задачу.

Пусть надо написать программу, вычисляющую площади разных фигур. Пользователь указывает, площадь какой фигуры он хочет вычислить. После этого вводит исходные данные. Например, длину и ширину в случае прямоугольника. Чтобы разделить поток выполнения на несколько ветвей, следует использовать оператор *if-elif-else*:

```
import math
import sys

figure = input("1-прямоугольник, 2-треугольник, 3-круг: ")

if figure == '1':
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    print(f"Площадь: {a * b}")
elif figure == '2':
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    print(f"Площадь: {0.5 * a * h}")
elif figure == '3':
    r = float(input("Радиус: "))
    print(f"Площадь: {math.pi * r**2}")
else:
    print("Ошибка ввода", file=sys.stderr)
```

Здесь нет никаких функций, и все прекрасно. Но напишем вариант с функциями:

```
import math
import sys

def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    print(f"Площадь: {a * b}")

def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    print(f"Площадь: {0.5 * a * h}")

def circle():
    r = float(input("Радиус: "))
```

```

print("Площадь: {math.pi * r**2}")

figure = input("1-прямоугольник, 2-треугольник, 3-круг: ")
if figure == '1':
    rectangle()
elif figure == '2':
    triangle()
elif figure == '3':
    circle()
else:
    print("Ошибка ввода", file=sys.stderr)

```

Он кажется сложнее, а каждая из трех функций вызывается всего один раз. Однако из общей логики программы как бы убраны и обособлены инструкции для нахождения площадей. Программа теперь состоит из отдельных "кирпичиков Лего". В основной ветке мы можем комбинировать их как угодно. Она играет роль управляющего механизма.

Если нам когда-нибудь захочется вычислять площадь треугольника по формуле Герона, а не через высоту, то не придется искать код во всей программе (представьте, что она состоит из тысяч строк кода как реальные программы). Мы пойдем к месту определения функций и изменим тело одной из них.

Если понадобиться использовать эти функции в какой-нибудь другой программе, то мы сможем импортировать их туда, сославшись на данный файл с кодом (как это делается в Python, будет рассмотрено позже).

Локальные и глобальные переменные

В программировании особое внимание уделяется концепции о локальных и глобальных переменных, а также связанное с ними представление об областях видимости. Соответственно, локальные переменные видны только в локальной области видимости, которой может выступать отдельно взятая функция. Глобальные переменные видны во всей программе. "Видны" – значит, известны, доступны. К ним можно обратиться по имени и получить связанное с ними значение.

К глобальной переменной можно обратиться из локальной области видимости. К локальной переменной нельзя обратиться из глобальной области видимости, потому что локальная переменная существует только в момент выполнения тела функции. При выходе из нее, локальные переменные исчезают. Компьютерная память, которая под них отводилась, освобождается. Когда функция будет снова вызвана, локальные переменные будут созданы заново.

Вернемся к нашей программе из прошлого урока, немного упростив ее для удобства:

```

def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    print(f"Площадь: {a * b}")

def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    print(f"Площадь: {0.5 * a * h}")

```

```
figure = input("1-прямоугольник, 2-треугольник: ")
if figure == '1':
    rectangle()
elif figure == '2':
    triangle()
```

Сколько здесь переменных? Какие из них являются глобальными, а какие – локальными?

Здесь пять переменных. Глобальной является только `figure`. Переменные `a` и `b` из функции `rectangle()`, а также `a` и `h` из `triangle()` – локальные. При этом локальные переменные с одним и тем же идентификатором `a`, но объявленные в разных функциях, – разные переменные.

Следует отметить, что идентификаторы `rectangle` и `triangle`, хотя и не являются именами переменных, а представляют собой имена функций, также имеют область видимости. В данном случае она глобальная, так как функции объявлены непосредственно в основной ветке программы.

В приведенной программе к глобальной области видимости относятся заголовки объявлений функций, объявление и присваивание переменной `figure`, конструкция условного оператора.

К локальной области относятся тела функций. Если, находясь в глобальной области видимости, мы попытаемся обратиться к локальной переменной, то возникнет ошибка:

```
...
elif figure == '2':
    triangle()

print(a)
```

Пример выполнения:

```
1-прямоугольник, 2-треугольник: 2
Основание: 4
Высота: 5
Площадь: 10.00
Traceback (most recent call last):
  File "test.py", line 17, in <module>
    print(a)
NameError: name 'a' is not defined
```

Однако мы можем обращаться из функций к глобальным переменным:

```
def rectangle():
    a = float(input("Ширина %s: " % figure))
    b = float(input("Высота %s: " % figure))
    print(f"Площадь: {a * b}")

def triangle():
    a = float(input("Основание %s: " % figure))
    h = float(input("Высота %s: " % figure))
    print(f"Площадь: {0.5 * a * h}")

figure = input("1-прямоугольник, 2-треугольник: ")
if figure == '1':
```

```
rectangle()
elif figure == '2':
    triangle()
```

Пример выполнения:

```
1-прямоугольник, 2-треугольник: 1
Ширина 1: 6.35
Высота 1: 2.75
Площадь: 17.46
```

В данном случае из тел функций происходит обращение к имени `figure`, которое, из-за того, что было объявлено в глобальной области видимости, видимо во всей программе.

Наши функции не совсем идеальны. Они должны вычислять площади фигур, но выводить результат на экран им не следовало бы. Вполне вероятна ситуация, когда результат нужен для внутренних нужд программы, для каких-то дальнейших вычислений, а выводить ли его на экран – вопрос второстепенный.

Если функции не будут выводить, а только вычислять результат, то его надо где-то сохранить для дальнейшего использования. Для этого подошли бы глобальные переменные. В них можно записать результат. Напишем программу вот так:

```
result = 0

def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    result = a*b

def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    result = 0.5 * a * h

figure = input("1-прямоугольник, 2-треугольник: ")
if figure == '1':
    rectangle()
elif figure == '2':
    triangle()

print("Площадь: %.2f" % result)
```

Итак, мы ввели в программу глобальную переменную `result` и инициировали ее нулем. В функциях ей присваивается результат вычислений. В конце программы ее значение выводится на экран. Мы ожидаем, что программа будет прекрасно работать. Однако...

```
1-прямоугольник, 2-треугольник: 2
Основание: 6
Высота: 4.5
Площадь: 0.00
```

... что-то пошло не так.

Дело в том, что в Python присвоение значения переменной совмещено с ее объявлением. (Во многих других языках это не так.) Поэтому, когда имя `result` впервые упоминается в локальной области видимости, и при этом происходит присваивание ей значения, то создается локальная переменная `result`. Это другая переменная, никак не связанная с глобальной `result`.

Когда функция завершает свою работу, то значение локальной `result` теряется, а глобальная не была изменена.

Когда мы вызывали внутри функции переменную `figure`, то ничего ей не присваивали.

Наоборот, мы запрашивали ее значение. Интерпретатор Питона искал ее значение сначала в локальной области видимости и не находил. После этого шел в глобальную и находил.

В случае с `result` он ничего не ищет. Он выполняет вычисления справа от знака присваивания, создает локальную переменную `result`, связывает ее с полученным значением.

На самом деле можно принудительно обратиться к глобальной переменной. Для этого существует команда `global`:

```
result = 0

def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    global result
    result = a*b

def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    global result
    result = 0.5 * a * h

figure = input("1-прямоугольник, 2-треугольник: ")
if figure == '1':
    rectangle()
elif figure == '2':
    triangle()

print(f"Площадь: {result}")
```

В таком варианте программа будет работать правильно.

Однако менять значения глобальных переменных в теле функции – плохая практика. В больших программах программисту трудно отследить, где, какая функция и почему изменила их значение. Программист смотрит на исходное значение глобальной переменной и может подумать, что оно остается таким же. Сложно заметить, что какая-то функция поменяла его. Подобное ведет к логическим ошибкам.

Чтобы избавиться от необходимости использовать глобальные переменные, для функций существует возможность возврата результата своей работы в основную ветку программы. И уже это полученное из функции значение можно присвоить глобальной переменной в глобальной области видимости. Это делает программу более понятной.

Возврат значений из функции. Оператор `return`

Функции могут передавать какие-либо данные из своих тел в основную ветку программы. Говорят, что функция возвращает значение. В большинстве языков программирования, в том числе Python, выход из функции и передача данных в то место, откуда она была вызвана, выполняется оператором `return`.

Если интерпретатор Питона, выполняя тело функции, встречает `return`, то он "забирает" значение, указанное после этой команды, и "уходит" из функции.

```
import math

def cylinder():
    r = float(input())
    h = float(input())
    # площадь боковой поверхности цилиндра:
    side = 2 * math.pi * r * h
    # площадь одного основания цилиндра:
    circle = math.pi * r**2
    # полная площадь цилиндра:
    full = side + 2 * circle
    return full

square = cylinder()
print(square)
```

Пример выполнения:

```
3
7
188.4
```

В данной программе в основную ветку из функции возвращается значение локальной переменной `full`. Не сама переменная, а ее значение, в данном случае – какое-либо число, полученное в результате вычисления площади цилиндра.

В основной ветке программы это значение присваивается глобальной переменной `square`. То есть выражение `square = cylinder()` выполняется так:

1. Вызывается функция `cylinder()`.
2. Из нее возвращается значение.
3. Это значение присваивается переменной `square`.

Не обязательно присваивать результат переменной, его можно сразу вывести на экран:

```
...
print(cylinder())
```

Здесь число, полученное из `cylinder()`, непосредственно передается функции `print()`. Если мы в программе просто напишем `cylinder()`, не присвоив полученные данные переменной или не передав их куда-либо дальше, то эти данные будут потеряны. Но синтаксической ошибки не будет.

В функции может быть несколько операторов `return`. Однако всегда выполняется только один из них. Тот, которого первым достигнет поток выполнения. Допустим, мы решили обработать исключение, возникающее на некорректный ввод. Пусть тогда в ветке `except` обработчика исключений происходит выход из функции без всяких вычислений и передачи значения:

```
import math

def cylinder():
    try:
        r = float(input())
        h = float(input())
    except ValueError:
        return

    side = 2 * math.pi * r * h
    circle = math.pi * r**2
    full = side + 2 * circle
    return full

print(cylinder())
```

Если попытаться вместо цифр ввести буквы, то сработает `return`, вложенный в `except`. Он завершит выполнение функции, так что все нижеследующие вычисления, в том числе `return full`, будут опущены. Пример выполнения:

```
r
None
```

Но постойте! Что это за слово `None`, которое нам вернул "пустой" `return`? Это ничего, такой объект – "ничто". Он принадлежит классу `NoneType`. До этого мы знали четыре типа данных, они же четыре класса: `int`, `float`, `str`, `bool`. Пришло время пятого.

Когда после `return` ничего не указывается, то по умолчанию считается, что там стоит объект `None`. Но никто вам не мешает явно написать `return None`.

Более того. Ранее мы рассматривали функции, которые вроде бы не возвращали никакого значения, потому что в них не было оператора `return`. На самом деле возвращали, просто мы не обращали на него внимание, не присваивали никакой переменной и не выводили на экран. В Python всякая функция что-либо возвращает. Если в ней нет оператора `return`, то она возвращает `None`. То же самое, как если в ней имеется "пустой" `return`.

Возврат нескольких значений

В Питоне позволительно возвращать из функции несколько объектов, перечислив их через запятую после команды `return`:

```
import math

def cylinder():
    r = float(input())
    h = float(input())
    side = 2 * math.pi * r * h
```

```
circle = math.pi * r**2
full = side + 2 * circle
return side, full

scyl, fcyl = cylinder()
print("Площадь боковой поверхности {scyl}")
print("Полная площадь {fcyl}")
```

Из функции `cylinder()` возвращаются два значения. Первое из них присваивается переменной `scyl`, второе – `fcyl`. Возможность такого группового присвоения – особенность Python, обычно не характерная для других языков:

```
>>> a, b, c = 10, 15, 19
>>> a
10
>>> b
15
>>> c
19
```

Фокус здесь в том, что перечисление значений через запятую (например, `10, 15, 19`) создает объект типа *tuple*. На русский переводится как "кортеж".

Когда же кортеж присваивается сразу нескольким переменным, то происходит сопоставление его элементов соответствующим в очереди переменным. Это называется распаковкой.

Таким образом, когда из функции возвращается несколько значений, на самом деле из нее возвращается один объект класса *tuple*. Перед возвратом эти несколько значений упаковываются в кортеж. Если же после оператора `return` стоит только одна переменная или объект, то ее/его тип сохраняется как есть.

Распаковка не является обязательной. Будет работать и так:

```
...
print(cylinder())
```

Пример выполнения:

```
4
3
(75.36, 175.84)
```

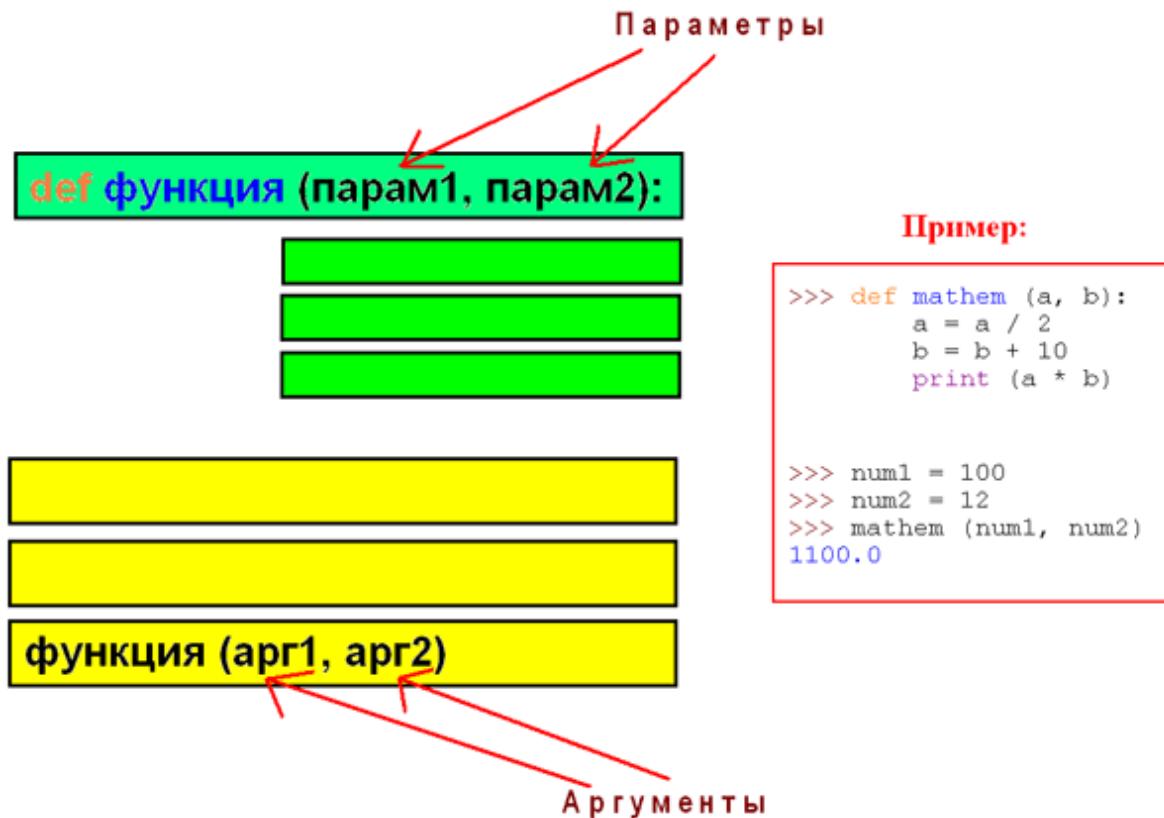
На экран выводится кортеж, о чём говорят круглые скобки. Его также можно присвоить одной переменной, а потом вывести ее значение на экран.

Параметры и аргументы функции

В программировании функции могут не только возвращать данные, но также принимать их, что реализуется с помощью так называемых параметров, которые указываются в скобках в заголовке функции. Количество параметров может быть любым.

Параметры представляют собой локальные переменные, которым присваиваются значения в момент вызова функции. Конкретные значения, которые передаются в функцию при ее вызове, будем называть аргументами. Следует иметь в виду, что встречается иная терминология. Например, формальные параметры и фактические параметры. В Python же обычно все называют аргументами.

Рассмотрим схему и поясняющий ее пример:



Когда функция вызывается, то ей передаются аргументы. В примере указаны глобальные переменные `num1` и `num2`. Однако на самом деле передаются не эти переменные, а их значения. В данном случае числа 100 и 12. Другими словами, мы могли бы писать `mathem(100, 12)`. Разницы не было бы.

Когда интерпретатор переходит к функции, чтобы начать ее исполнение, он присваивает переменным-параметрам переданные в функцию значения-аргументы. В примере переменной `a` будет присвоено 100, `b` будет присвоено 12.

Изменение значений `a` и `b` в теле функции никак не скажется на значениях переменных `num1` и `num2`. Они останутся прежними. В Python такое поведение характерно для неизменяемых типов данных, к которым относятся, например, числа и строки. Говорят, что в функцию данные передаются по значению. Так, когда `a` присваивалось число 100, то это было уже другое число, не то, на которое ссылается переменная `num1`. Число 100 было скопировано и помещено в отдельную ячейку памяти для переменной `a`.

Существуют изменяемые типы данных. Для Питона, это, например, списки и словари. В этом случае данные передаются по ссылке. В функцию передается ссылка на них, а не сами данные. И эта ссылка связывается с локальной переменной. Изменения таких данных через локальную переменную обнаруживаются при обращении к ним через глобальную. Это есть следствие того, что несколько переменныхзываются на одни и те же данные, на одну и ту же область памяти.

Необходимость передачи по ссылке связана в первую очередь с экономией памяти. Сложные типы данных, по сути представляющие собой структуры данных, обычно копировать не целесообразно. Однако, если надо, всегда можно сделать это принудительно.

Произвольное количество аргументов

Обратим внимание еще на один момент. Количество аргументов и параметров совпадает.

Нельзя передать три аргумента, если функция принимает только два. Нельзя передать один аргумент, если функция требует два обязательных. В рассмотренном примере они обязательные.

Однако в Python у функций бывают параметры, которым уже присвоено значение по-умолчанию. В таком случае, при вызове можно не передавать соответствующие этим параметрам аргументы. Хотя можно и передать. Тогда значение по умолчанию заменится на переданное.

```
import math

def cylinder(h, r=1):
    side = 2 * math.pi * r * h
    circle = math.pi * r**2
    full = side + 2 * circle
    return full

figure1 = cylinder(4, 3)
figure2 = cylinder(5)
print(figure1)
print(figure2)
```

При втором вызове `cylinder()` мы указываем только один аргумент. Он будет присвоен переменной-параметру `h`. Переменная `r` будет равна 1.

Согласно правилам синтаксиса Python при определении функции параметры, которым присваивается значение по-умолчанию должны следовать (находиться сзади) за параметрами, не имеющими значений по умолчанию.

А вот при вызове функции, можно явно указывать, какое значение соответствует какому параметру. В этом случае их порядок не играет роли:

```
...
figure3 = cylinder(10, 2)
figure4 = cylinder(r=2, h=10)
print(figure3)
print(figure4)
```

В данном случае оба вызова – это вызовы с одними и теми же аргументами-значениями. Просто в первом случае сопоставление параметрам-переменным идет в порядке следования. Во-втором случае – по ключам, которыми выступают имена параметров.

В Python определения и вызовы функций имеют и другие нюансы, рассмотрение которых мы пока опустим, так как они требуют более глубоких знаний, чем у нас есть на данный момент. Скажем лишь, что функции может быть определена так, что в нее можно передать хоть ни одного аргумента, хоть множество:

```
def one_or_many(*a):
    print(a)

one_or_many(1)
one_or_many('1', 1, 2, 'abc')
one_or_many()
```

Результат:

```
(1,)
('1', 1, 2, 'abc')
()
```

Lambda-функции

Python поддерживает интересный синтаксис, позволяющий определять небольшие однострочные функции на лету. Позаимствованные из Lisp, так называемые lambda-функции могут быть использованы везде, где требуется функция.

Небольшой пример

```
def func(x, y):
    return x**2 + y**2

func = lambda x, y: x**2 + y**2
```

С одной стороны "прикольно", вместо 2 строк - одна, но сложные конструкции на `Lambda` функциях не напишешь - плохо читаемы. Например

```
lambda x: (lambda y: x + y)
```

Так чем же отличаются так принципиально `def` и `Lambda`:

`Lambda` – это выражение, а не инструкция. По этой причине ключевое слово `Lambda` может появляться там, где синтаксис языка Python не позволяет использовать инструкцию `def`, – внутри литералов или в вызовах функций, например.

Из этого следует, что лямбды хорошо применять со встроенными функциями - `map`, `filter`:

```
foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]

print(list(filter(lambda x: x % 3 == 0, foo)))
# [18, 9, 24, 12, 27]

print(list(map(lambda x: x * 2 + 10, foo)))
# [14, 46, 28, 54, 44, 58, 26, 34, 64]
```

Есть и еще одно интересное применение - хранение списка обработчиков данных в списке/словаре:

```
plural_rules = [
    lambda n: 'all',
    lambda n: 'singular' if n == 1 else 'plural',
    lambda n: 'singular' if 0 <= n <= 1 else 'plural',
    ...
]
```

Документирование кода в Python. PEP 257

Документирование кода в python - достаточно важный аспект, ведь от нее порой зависит читаемость и быстрота понимания вашего кода, как другими людьми, так и вами через полгода. PEP 257 описывает соглашения, связанные со строками документации python, рассказывает о том, как нужно документировать python код. Цель этого PEP - стандартизировать структуру строк документации: что они должны в себя включать, и как это написать (не касаясь вопроса синтаксиса строк документации). Этот PEP описывает соглашения, а не правила или синтаксис.

При нарушении этих соглашений, самое худшее, чего можно ожидать - некоторых неодобрительных взглядов. Но некоторые программы (например, docutils), знают о соглашениях, поэтому следование им даст вам лучшие результаты.

Строки документации - строковые литералы, которые являются первым оператором в модуле, функции, классе или определении метода. Такая строка документации становится специальным атрибутом `__doc__` этого объекта.

Все модули должны, как правило, иметь строки документации, и все функции и классы, экспортруемые модулем также должны иметь строки документации. Публичные методы (в том числе `__init__`) также должны иметь строки документации. Пакет модулей может быть документирован в `__init__.py`.

Для согласованности, всегда используйте `"""triple double quotes"""` для строк документации. Используйте `r"""raw triple double quotes"""`, если вы будете использовать обратную косую черту в строке документации. Существует две формы строк документации: односторочная и многострочная.

Одиночные строки документации предназначены для действительно очевидных случаев. Они должны умещаться на одной строке. Например:

```
def kos_root():
    """Return the pathname of the kos root directory."""
    global _kos_root
    if _kos_root: return _kos_root
```

Используйте тройные кавычки, даже если документация умещается на одной строке. Потом будет проще её дополнить.

Односторонняя строка документации не должна быть "подписью" параметров функции / метода (которые могут быть получены с помощью интроспекции). Не делайте:

```
def function(a, b):
    """function(a, b) -> list""""
```

Этот тип строк документации подходит только для С функций (таких, как встроенные модули), где интроспекция не представляется возможной. Тем не менее, возвращаемое значение не может быть определено путем интроспекции. Предпочтительный вариант для такой строки документации будет что-то вроде:

```
def function(a, b):
    """Do X and return a list."""
```

Многострочные строки документации состоят из однострочной строки документации с последующей пустой строкой, а затем более подробным описанием. Первая строка может быть использована автоматическими средствами индексации, поэтому важно, чтобы она находилась на одной строке и была отделена от остальной документации пустой строкой. Первая строка может быть на той же строке, где и открывающие кавычки, или на следующей строке. Вся документация должна иметь такой же отступ, как кавычки на первой строке (см. пример ниже).

Вставляйте пустую строку до и после всех строк документации (однострочных или многострочных), которые документируют класс - вообще говоря, методы класса разделены друг от друга одной пустой строкой, а строка документации должна быть смещена от первого метода пустой строкой; для симметрии, поставьте пустую строку между заголовком класса и строкой документации. Строки документации функций и методов, как правило, не имеют этого требования.

Строки документации скрипта (самостоятельной программы) должны быть доступны в качестве "сообщения по использованию", напечатанной, когда программа вызывается с некорректными или отсутствующими аргументами (или, возможно, с опцией "-h", для помощи). Такая строка документации должна документировать функции программы и синтаксис командной строки, переменные окружения и файлы. Сообщение по использованию может быть довольно сложным (несколько экранов) и должно быть достаточным для нового пользователя для использования программы должным образом, а также полный справочник со всеми вариантами и аргументами для искушенного пользователя.

Строки документации модуля] должны, как правило, перечислять классы, исключения, функции (и любые другие объекты), которые экспортируются модулем, с краткими пояснениями (в одну строчку) каждого из них. (Эти строки, как правило, дают меньше деталей, чем первая строка документации к объекту). Строки документации пакета модулей (т. е. строка документации в `__init__.py`) также должны включать модули и подпакеты.

Строки документации функции или метода должны обобщить его поведение и документировать свои аргументы, возвращаемые значения, побочные эффекты, исключения, дополнительные аргументы, именованные аргументы, и ограничения на вызов функции.

Строки документации класса обобщают его поведение и перечисляют открытые методы и переменные экземпляра. Если класс предназначен для подклассов, и имеет дополнительный интерфейс для подклассов, этот интерфейс должен быть указан отдельно (в строке документации). Конструктор класса должен быть задокументирован в документации метода `__init__`. Отдельные методы должны иметь свои строки документации.

Если класс - подкласс другого класса, и его поведение в основном унаследовано от этого класса, строки документации должны отмечать это и обобщить различия. Используйте глагол "override", чтобы указать, что метод подкласса заменяет метод суперкласса и не вызывает его; используйте глагол "extend", чтобы указать, что метод подкласса вызывает метод суперкласса (в дополнение к собственному поведению).

Пример многострочной формы строк документации:

```

def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)

    """
    if imag == 0.0 and real == 0.0: return complex_zero
    ...

```

Пример 1. Для примера 1 лабораторной работы 2.6, оформить каждую команду в виде вызова отдельной функции.

Решение: на основание примера 1 лабораторной работы 2.6 напишем программу для решения поставленной задачи.

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sys
from datetime import date

def get_worker():
    """
    Запросить данные о работнике.
    """
    name = input("Фамилия и инициалы? ")
    post = input("Должность? ")
    year = int(input("Год поступления? "))

    # Создать словарь.
    return {
        'name': name,
        'post': post,
        'year': year,
    }

def display_workers(staff):
    """
    Отобразить список работников.
    """
    # Проверить, что список работников не пуст.
    if staff:
        # Заголовок таблицы.
        line = '+-{ }-+{ }-+{ }-+{ }-+'.format(
            '_' * 4,
            '_' * 30,
            '_' * 20,
            '_' * 8
        )
        print(line)
        print(
            '| {:>4} | {:>30} | {:>20} | {:>8} |'.format(
                "№",

```

```

        "Ф.И.О.",
        "должность",
        "Год"
    )
)
print(line)

# Вывести данные о всех сотрудниках.
for idx, worker in enumerate(workers, 1):
    print(
        '| {:>4} | {:<30} | {:<20} | {:>8} |'.format(
            idx,
            worker.get('name', ''),
            worker.get('post', ''),
            worker.get('year', 0)
        )
    )
)
print(line)

else:
    print("Список работников пуст.")

def select_workers(staff, period):
    """
    Выбрать работников с заданным стажем.
    """
    # Получить текущую дату.
    today = date.today()

    # Сформировать список работников.
    result = []
    for employee in staff:
        if today.year - employee.get('year', today.year) >= period:
            result.append(employee)

    # Возвратить список выбранных работников.
    return result

def main():
    """
    Главная функция программы.
    """
    # Список работников.
    workers = []

    # Организовать бесконечный цикл запроса команд.
    while True:
        # Запросить команду из терминала.
        command = input(">>> ").lower()

        # Выполнить действие в соответствие с командой.
        if command == 'exit':
            break

        elif command == 'add':
            # Запросить данные о работнике.

```

```

worker = get_worker()

# добавить словарь в список.
workers.append(worker)
# отсортировать список в случае необходимости.
if len(workers) > 1:
    workers.sort(key=lambda item: item.get('name', ''))

elif command == 'list':
    # отобразить всех работников.
    display_workers(workers)

elif command.startswith('select '):
    # Разбить команду на части для выделения стажа.
    parts = command.split(' ', maxsplit=1)
    # Получить требуемый стаж.
    period = int(parts[1])

    # выбрать работников с заданным стажем.
    selected = select_workers(workers, period)
    # отобразить выбранных работников.
    display_workers(selected)

elif command == 'help':
    # вывести справку о работе с программой.
    print("Список команд:\n")
    print("add - добавить работника;")
    print("list - вывести список работников;")
    print("select <стаж> - запросить работников со стажем;")
    print("help - отобразить справку;")
    print("exit - завершить работу с программой.")

else:
    print(f"Неизвестная команда {command}", file=sys.stderr)

if __name__ == '__main__':
    main()

```

Аппаратура и материалы

1. Компьютерный класс общего назначения с конфигурацией ПК не хуже рекомендованной для ОС Windows 10 с подключением к глобальной сети Интернет.
2. Операционная система Windows 10.
3. Система контроля версий Git.
4. Браузер для доступа к web-сервису GitHub, рекомендован к использованию Google Chrome.
5. Дистрибутив языка программирования Python, включающий набор популярных библиотек Anaconda.
6. Интегрированная среда разработки PyCharm Community Edition.

Указания по технике безопасности

При работе на ЭВМ без разрешения руководителя занятия запрещается:

- подавать (снимать) напряжение на ПЭВМ и электрические розетки с распределительного щита;
- включать и выключать блоки питания ПЭВМ и мониторы;
- извлекать ПЭВМ из защитного кожуха;
- устранять неисправности, возникшие в ходе выполнения лабораторной работы.

Методика и порядок выполнения работы

1. Изучить теоретический материал работы.
2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.
3. Выполните клонирование созданного репозитория.
4. Дополните файл `.gitignore` необходимыми правилами для работы с IDE PyCharm.
5. Организуйте свой репозиторий в соответствие с моделью ветвления git-flow.
6. Создайте проект PyCharm в папке репозитория.
7. Проработайте примеры лабораторной работы. Зафиксируйте изменения.
8. Решить следующую задачу: основная ветка программы, не считая заголовков функций, состоит из двух строки кода. Это вызов функции `test()` и инструкции `if __name__ == '__main__'`. В ней запрашивается на ввод целое число. Если оно положительное, то вызывается функция `positive()`, тело которой содержит команду вывода на экран слова "Положительное". Если число отрицательное, то вызывается функция `negative()`, ее тело содержит выражение вывода на экран слова "Отрицательное".
Понятно, что вызов `test()` должен следовать после определения функций. Однако имеет ли значение порядок определения самих функций? То есть должны ли определения `positive()` и `negative()` предшествовать `test()` или могут следовать после него? Проверьте вашу гипотезу, поменяв объявления функций местами. Попробуйте объяснить результат.
9. Зафиксируйте изменения в репозитории.
10. Решите следующую задачу: в основной ветке программы вызывается функция `cylinder()`, которая вычисляет площадь цилиндра. В теле `cylinder()` определена функция `circle()`, вычисляющая площадь круга по формуле πr^2 . В теле `cylinder()` у пользователя спрашивается, хочет ли он получить только площадь боковой поверхности цилиндра, которая вычисляется по формуле $2\pi rh$, или полную площадь цилиндра. В последнем случае к площади боковой поверхности цилиндра должен добавляться удвоенный результат вычислений функции `circle()`.
11. Зафиксируйте изменения в репозитории.
12. Решите следующую задачу: напишите функцию, которая считывает с клавиатуры числа и перемножает их до тех пор, пока не будет введен 0. Функция должна возвращать полученное произведение. Вызовите функцию и выведите на экран результат ее работы.
13. Зафиксируйте изменения в репозитории.
14. Решите следующую задачу: напишите программу, в которой определены следующие четыре функции:
 1. Функция `get_input()` не имеет параметров, запрашивает ввод с клавиатуры и возвращает в основную программу полученную строку.
 2. Функция `test_input()` имеет один параметр. В теле она проверяет, можно ли переданное ей значение преобразовать к целому числу. Если можно, возвращает логическое `True`. Если нельзя – `False`.

3. Функция `str_to_int()` имеет один параметр. В теле преобразовывает переданное значение к целочисленному типу. Возвращает полученное число.
4. Функция `print_int()` имеет один параметр. Она выводит переданное значение на экран и ничего не возвращает.

В основной ветке программы вызовите первую функцию. То, что она вернула, передайте во вторую функцию. Если вторая функция вернула `True`, то те же данные (из первой функции) передайте в третью функцию, а возвращенное третьей функцией значение – в четвертую.

15. Зафиксируйте изменения в репозитории.
16. Приведите в отчете скриншоты результатов выполнения примера при различных исходных данных вводимых с клавиатуры.
17. Приведите в отчете скриншоты работы программы решения индивидуального задания.
18. Зафиксируйте сделанные изменения в репозитории.
19. Выполните слияние ветки для разработки с веткой `master/main`.
20. Отправьте сделанные изменения на сервер GitHub.

Индивидуальное задание

Решить индивидуальное задание лабораторной работы 2.6, оформив каждую команду в виде отдельной функции.

Содержание отчета и его форма

Отчет по лабораторной работе оформляется письменно в рабочей тетради, должен содержать ответы на контрольные вопросы, ссылку на репозиторий с которым выполнялась работа, скриншоты IDE PyCharm, скриншоты результатов работы программ.

Вопросы для защиты работы

1. Каково назначение функций в языке программирования Python?
2. Каково назначение операторов `def` и `return`?
3. Каково назначение локальных и глобальных переменных при написании функций в Python?
4. Как вернуть несколько значений из функции Python?
5. Какие существуют способы передачи значений в функцию?
6. Как задать значение аргументов функции по умолчанию?
7. Каково назначение `lambda`-выражений в языке Python?
8. Как осуществляется документирование кода согласно PEP257?
9. В чем особенность односторонних и многострочных форм строк документации?