```
>>> def func(**kwards):
        return kwards

>>> func(a=1, b=2, c=3)
{'a': 1, 'b': 2, 'c': 3}
>>> func()
{}
>>> func(a='python')
{'a': 'python'}
```

В переменной kwargs у нас хранится словарь, с которым мы, опять-таки, можем делать все, что нам заблагорассудится.

Анонимные функции, инструкция lambda.

Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции lambda. Кроме этого, их не обязательно присваивать переменной, как делали мы инструкцией def func():

```
>>> func = lambda x,y: x+y
>>> func(1, 2)
3
>>> func('a', 'b')
'ab'
>>> (lambda x,y: x+y)(1, 2)
3
>>> (lambda x,y: x+y)('a', 'b')
'ab'
```

Lambda-функции, в отличие от обычной, не требуется инструкция return, а в остальном, ведет себя точно так же:

```
>>> func = lambda *args: args
>>> func(1, 2, 3, 4)
(1, 2, 3, 4)
```

Далее применение функций будет рассмотрено в практических заданиях.

Оборудование и материалы.

Персональный компьютер, среда разработки Python.

Указания по технике безопасности:

Соответствуют технике безопасности по работе с компьютерной техникой.

Задания

Давайте действовать пошагово. Сначала определим и вызовем функцию, которая не имеет параметров. Перед вами пример простейшей функции:

```
>>> def do_nothing():
        pass
```

Даже если функции не нужны параметры, вам все равно необходимо указать круглые скобки и двоеточие в ее определении. Следующую строку необходимо выделить пробелами точно так же, как если бы это был оператор if. Python требует использовать выражение pass, чтобы показать, что функция ничего не делает. Это эквивалентно утверждению «Эта страница специально оставлена пустой» (несмотря на то что теперь это не так).

Функцию можно вызвать, просто написав ее имя и скобки. Она сработает так, как я и обещал, вполне успешно не сделав ничего:

```
>>> do_nothing()
>>>
```

Теперь определим и вызовем другую функцию, которая не имеет параметров и выводит на экран одно слово:

```
>>> def make_a_sound():
        print('quack')


>>> make_a_sound()
quack
```

Когда вы вызываете функцию make_a_sound(), Python выполняет код, расположенный внутри ее описания. В этом случае он выводит одно слово и возвращает управление основной программе.

Попробуем написать функцию, которая не имеет параметров, но возвращает значение:

```
>>> def agree():
        return True
```

Вы можете вызвать эту функцию и проверить возвращаемое ею значение с помощью if:

```
>>> if agree():
        print('Отлично!')
else:
        print('Что-то пошло не так!')


Отлично!
```

Комбинация функций с проверками вроде if и циклами вроде while позволяет вам делать ранее недоступные вещи.

Теперь пришло время поместить аргументы в скобки после названия функции. Определим функцию echo(), имеющую один параметр anything. Она использует оператор return, чтобы отправить значение anything вызывающей стороне дважды, разделив их пробелом:

```
>>> def echo(anything):
        return anything + ' ' + anything
```

Теперь вызовем функцию echo(), передав ей строку 'Эгегей':

```
>>> echo('Эгегей')
'Эгегей Эгегей'
>>>
```

Значения, которые вы передаете в функцию при вызове, называются аргументами. Когда вы вызываете функцию с аргументами, значения этих аргументов копируются в соответствующие параметры внутри функций. В предыдущем примере функции echo() передавалась строка 'Эгегей'. Это значение копировалось внутри функции echo() в параметр anything, а затем возвращалось (в этом случае оно удваивалось и разделялось пробелом) вызывающей стороне.

Эти примеры функций довольно просты. Напишем функцию, которая принимает аргумент и что-то с ним делает. Мы адаптируем предыдущий фрагмент кода, который комментировал цвета. Назовем его commentary и сделаем так, чтобы он принимал в качестве аргумента строку color. Сделаем так, чтобы он возвращал описание строки вызывающей стороне, которая может решить, что с ним делать дальше:

```
>>> def commentary(color):
        if color=='красный':
                return "Это помидорка"
        elif color=='зеленый':
                return "Это огурчик"
        elif color=='ультрамарин':
                return "Я не знаю такого овоща"
        else:
                return "Я не знаю цвет " + color + "."


>>>
```

Вызовем функцию commentary(), передав ей в качестве аргумента строку 'blue'
Функция сделает следующее:

- присвоит значение 'blue' параметру функции color;
- пройдет по логической цепочке if-elif-else;
- вернет строку;
- присвоит строку переменной comment.

Что мы получим в результате?

```
>>> commentary('голубой')
'Я не знаю цвет голубой.'
```

Функция может принимать любое количество аргументов (включая нуль) любого типа. Она может возвращать любое количество результатов (также включая нуль) любого типа. Если функция не вызывает return явно, вызывающая сторона получит результат None.

```
>>> print(do_nothing())
None
```

Использование значения None

None — это специальное значение в Python, которое заполняет собой пустое место, если функция ничего не возвращает. Оно не является булевым значением False, несмотря на то что похоже на него при проверке булевой переменной. Рассмотрим пример:

```
>>> thing = None
>>> if thing:
        print("It's some thing")
else:
        print("It's no thing")


It's no thing
```

Для того чтобы понять важность отличия None от булева значения False, используйте оператор is:

```
>>> if thing is None:
        print("It's nothing")
else:
        print("It's something")


It's nothing
```

Разница кажется небольшой, однако она важна в Python. None потребуется вам, чтобы отличить отсутствующее значение от пустого. Помните, что целочисленные нули, нули с плавающей точкой, пустые строки (''), списки ([]), кортежи ((,)), словари ({}) и множества (set()) все равны False, но не равны None.

Напишем небольшую функцию, которая выводит на экран проверку на равенство None:

```
>>> def is_none(thing):
        if thing is None:
                print("It's None")
        elif thing:
                print("It's True")
        else:
                print("It's False")


>>>
```

Теперь выполним несколько проверок:

```
>>> is_none(None)
It's None
>>> is_none(True)
It's True
>>> is_none(False)
It's False
>>> is_none(0)
It's False
>>> is_none(0.0)
It's False
>>> is_none(())
It's False
>>> is_none([])
It's False
>>> is_none({})
It's False
>>> is_none(set())
It's False
```

Позиционные аргументы

Python довольно гибко обрабатывает аргументы функций в сравнении с многими языками программирования. Наиболее распространенный тип аргументов — это позиционные аргументы, чьи значения копируются в соответствующие параметры согласно порядку следования.

Эта функция создает словарь из позиционных входных аргументов и возвращает его:

```
>>> def menu(wine, entree, dessert):
        return {'напиток': wine, 'основное блюдо': entree, 'десерт': dessert}

>>> menu('шардоне', 'цыплёнок табака', 'печеньки')
{'напиток': 'шардоне', 'основное блюдо': 'цыплёнок табака', 'десерт': 'печеньк
и'}
>>>
```

Несмотря на распространенность аргументов такого типа, у них есть недостаток, который заключается в том, что вам нужно запоминать значение каждой позиции.

Если бы мы вызвали функцию menu(), передав в качестве последнего аргумента марку вина, обед вышел бы совершенно другим:

```
>>> menu('говядина', 'мороженка', 'портвейн 777')
{'напиток': 'говядина', 'основное блюдо': 'мороженка', 'десерт': 'портвейн 777
'}
```

Аргументы — ключевые слова

Для того чтобы избежать путаницы с позиционными аргументами, вы можете указать аргументы с помощью имен соответствующих параметров. Порядок следования аргументов в этом случае может быть иным:

```
>>> menu(entree='шашлык', dessert='пирожок с вишней', wine='кефир')
{'напиток': 'кефир', 'основное блюдо': 'шашлык', 'дессерт': 'пирожок с вишней'}
```

Вы можете объединять позиционные аргументы и аргументы — ключевые слова. Сначала выберем вино, а для десерта и основного блюда используем аргументы — ключевые слова.

```
>>> menu('вдова Клико', dessert='тирамису', entree='рыба')
{'напиток': 'вдова Клико', 'основное блюдо': 'рыба', 'дессерт': 'тирамису'}
```

Если вы вызываете функцию, имеющую как позиционные аргументы, так и аргументы — ключевые слова, то позиционные аргументы необходимо указывать первыми.

Указываем значение параметра по умолчанию

Вы можете указать значения по умолчанию для параметров. Значения по умолчанию используются в том случае, если вызывающая сторона не предоставила соответствующий аргумент. Эта приятная особенность может оказаться довольно полезной. Воспользуемся предыдущим примером:

```
def menu(wine, entree, dessert='вкусняшка'):
    return{'напиток': wine, 'основное блюдо' : entree, 'десерт' : dessert}
```

В этот раз мы вызовем функцию menu(), не передав ей аргумент dessert:

```
>>> menu('Кисель', 'Курица с нежным сливочным соусом')
{'напиток': 'Кисель', 'основное блюдо': 'Курица с нежным сливочным соусом', 'де
серт': 'вкусняшка'}
```

Если вы предоставите аргумент, он будет использован вместо аргумента по умолчанию:

```
>>> menu('Компот', 'Коржик', 'Карамелька')
{'напиток': 'Компот', 'основное блюдо': 'Коржик', 'десерт': 'Карамелька'}
```

> Значение аргументов по умолчанию высчитывается, когда функция определяется, а не выполняется. Распространенной ошибкой новичков (и иногда не совсем новичков) является использование изменяемого типа данных вроде списка или словаря в качестве аргумента по умолчанию.

В следующей проверке ожидается, что функция buggy() будет каждый раз запускаться с новым пустым списком result, добавлять в него аргумент arg, а затем выводить на экран список, состоящий из одного элемента. Однако в этой функции есть баг: список будет пуст только при первом вызове. Во второй раз список result будет содержать элемент, оставшийся после предыдущего вызова:

```
>>> def buggy(arg, result=[]):
        result.append(arg)
        print(result)


>>> buggy('a')
['a']
>>> buggy('b') #ожидаем увидеть ['b']
['a', 'b']
```

Функция работала бы корректно, если бы код выглядел так:

```
>>> def works(arg):
        result = []
        result.append(arg)
        return result

>>> works('a')
['a']
>>> works('b')
['b']
```

Решить проблему можно, передав в функцию что-то еще, чтобы указать на то, что вызов является первым:

```
>>> def nonbuggy(arg, result=None):
        if result is None:
                result = []
        result.append(arg)
        print(result)


>>> nonbuggy('a')
['a']
>>> nonbuggy('b')
['b']
```

Получаем позиционные аргументы с помощью *

Если вы работали с языками программирования C или C++, то можете предположить, что астериск (*) в Python как-то относится к указателям. Это не так, Python не имеет указателей.

Если символ * будет использован внутри функции с параметром, произвольное количество позиционных аргументов будет сгруппировано в кортеж. В следующем примере args является кортежем параметров, который был создан из аргументов, переданных в функцию print_args():

```
>>> def print_args(*args):
        print('Кортеж позиционных аргументов:', args)
```

Если вы вызовете функцию без аргументов, то получите пустой кортеж:

```
>>> print_args()
Кортеж позиционных аргументов: ()
```

Все аргументы, которые вы передадите, будут выведены на экран как кортеж args:

```
>>> print_args(3, 2, 1, 'Леген...', 'подождите...', '...дарно!!!')
Кортеж позиционных аргументов: (3, 2, 1, 'Леген...', 'подождите...', '...дарно!!!')
```

Это полезно при написании функций вроде print(), которые принимают произвольное количество аргументов. Если в вашей функции имеются также обязательные позиционные аргументы, *args отправится в конец списка и получит все остальные аргументы:

```
>>> def print_more(required1, required2, *args):
...     print('Need this one:', required1)
...     print('Need this one too:', required2)
...     print('All the rest:', args)
...
>>> print_more('cap', 'gloves', 'scarf', 'monocle', 'mustache wax')
Need this one: cap
Need this one too: gloves
All the rest: ('scarf', 'monocle', 'mustache wax')
```

При использовании * вам не нужно обязательно называть кортеж параметров args, однако это распространенная идиома в Python.

Получение аргументов — ключевых слов с помощью **

Вы можете использовать два астериска (**), чтобы сгруппировать аргументы — ключевые слова в словарь, где имена аргументов станут ключами, а их значения — соответствующими значениями в словаре. В следующем примере определяется функция print_kwargs(), в которой выводятся ее аргументы — ключевые слова:

```
>>> def print_kwargs(**kwargs):
...     print('Keyword arguments:', kwargs)
...
```

Теперь попробуйте вызвать ее, передав несколько аргументов:

```
>>> print_kwargs(wine='merlot', entree='mutton', dessert='macaroon')
Keyword arguments: {'dessert': 'macaroon', 'wine': 'merlot', 'entree': 'mutton'}
```

Внутри функции kwargs является словарем.

Если вы используете позиционные аргументы и аргументы — ключевые слова (*args и **kwargs), они должны следовать в этом же порядке. Как и в случае с args, вам не обязательно называть этот словарь kwargs, но это опять же является распространенной практикой.

Строки документации

Дзен Python гласит: удобочитаемость имеет значение. Вы можете прикрепить документацию к определению функции, включив строку в начало ее тела. Она называется строкой документации:

```
>>> def echo(anything):
...     'echo returns its input argument'
...     return anything
```

Вы можете сделать строку документации довольно длинной и даже, если хотите, применить к ней форматирование, что показано в следующем примере:

```
def print_if_true(thing, check):
    '''
    Prints the first argument if a second argument is true.
    The operation is:
    1. Check whether the *second* argument is true.
    2. If it is, print the *first* argument.
'''
if check:
    print(thing)
```

Для того чтобы вывести строку документации некоторой функции, вам следует вызвать функцию help(). Передайте ей имя функции, чтобы получить список всех аргументов и красиво отформатированную строку документации:

```
>>> help(echo)
Help on function echo in module __main__:
echo(anything)
    echo returns its input argument
```

Если вы хотите увидеть только строку документации без форматирования:

```
>>> print(echo.__doc__)
echo returns its input argument
```

Подозрительно выглядящая строка __doc__ является внутренним именем строки документации как переменной внутри функции. В пункте «Использование _ и __ в именах» в разделе «Пространства имен и область определения» данной работы объясняется причина появления всех этих нижних подчеркиваний.

Функции — это объекты первого класса

В Python объектами является все, включая числа, строки, кортежи, списки, словари и даже функции. Функции в Python являются объектами первого класса. Вы можете присвоить их переменным, использовать как аргументы для других функций и возвращать из функций. Это дает вам возможность решать с помощью Python такие задачи, справиться с которыми средствами многих других языков сложно, если не невозможно.

Для того чтобы убедиться в этом, определим простую функцию answer(), которая не имеет аргументов и просто выводит число 42:

```
>>> def answer():
...     print(42)
```

Вы знаете, что получите в качестве результата, если запустите эту функцию:

```
>>> answer()
42
```

Теперь определим еще одну функцию с именем run_something. Она имеет один аргумент, который называется func и представляет собой функцию, которую нужно запустить. Эта функция просто вызывает другую функцию:

```
>>> def run_something(func):
...     func()
```

Если мы передадим answer в функцию run_something(), то используем ее как данные, прямо как и другие объекты:

```
>>> run_something(answer)
42
```

Обратите внимание: вы передали строку answer, а не answer(). В Python круглые скобки означают «вызови эту функцию». Если скобок нет, Python относится к функции как к любому другому объекту. Это происходит потому, что, как и все остальное в Python, функция является объектом:

```
>>> type(run_something)
<class 'function'>
```

Попробуем запустить функцию с аргументами. Определим функцию add_args(), которая выводит на экран сумму двух числовых аргументов, arg1 и arg2:

```
>>> def add_args(arg1, arg2):
...     print(arg1 + arg2)
```

Чем является add_args()?

```
>>> type(add_args)
<class 'function'>
```

Теперь определим функцию, которая называется run_something_with_args() и принимает три аргумента:

- func — функция, которую нужно запустить;
- arg1 — первый аргумент функции func;
- arg2 — второй аргумент функции func:

```
>>> def run_something_with_args(func, arg1, arg2):
...     func(arg1, arg2)
```

Когда вы вызываете функцию run_something_with_args(), та функция, что передается вызывающей стороной, присваивается параметру func, а переменные arg1 и arg2 получают значения, которые следуют далее в списке аргументов. Вызов func(arg1, arg2) выполняет данную функцию с этими аргументами, потому что круглые скобки указывают Python сделать это.

Проверим функцию run_something_with_args(), передав ей имя функции add_args и аргументы 5 и 9:

```
>>> run_something_with_args(add_args, 5, 9)
14
```

Внутри функции run_something_with_args() аргумент add_args, представляющий собой имя функции, был присвоен параметру func, 5 — параметру arg1, а 9 — параметру arg2. В итоге получается следующая конструкция:

```
add_args(5, 9)
```

Вы можете объединить этот прием с использованием *args и **kwargs.

Определим тестовую функцию, которая принимает любое количество позиционных аргументов, определяет их сумму с помощью функции sum() и возвращает ее:

Функция sum() - это встроенная в Python функция, которая высчитывает сумму значений итерабельного числового (целочисленного или с плавающей точкой) аргумента.

Мы определим новую функцию run_with_positional_args(), принимающую функцию и произвольное количество позиционных аргументов, которые нужно будет передать в нее:

```
>>> def run_with_positional_args(func, *args):
...     return func(*args)
```

Теперь вызовем ее:

```
>>> run_with_positional_args(sum_args, 1, 2, 3, 4)
10
```

Вы можете использовать функции как элементы списков, кортежей, множеств и словарей. Функции неизменяемы, поэтому вы можете даже применять их как ключи для словарей.

Контрольные вопросы

1. Определение и вызов функций.
2. Синтаксис функций.
3. Использование параметров функций.
4. Использование оператора return.
5. Использование значения None.
6. Позиционные аргументы.
7. Аргументы — ключевые слова.
8. Значение параметра по умолчанию.
9. Получение аргументов — ключевых слов с помощью **.

Список литературы, рекомендуемый к использованию по данной теме:

1. Орлов, С. А. Программная инженерия. Технологии разработки программного обеспечения : учебник / С.А. Орлов. - 5-е изд., обновл. и доп. - СПб. : Питер, 2017. - 640 с.
2. Гагарина, Л. Г. Современные проблемы информатики и вычислительной техники : [учеб. пособие] / Л.Г. Гагарина, А.А. Петров. - М. : Форум, 2016. - 368 с.
3. Шкляр, М. Ф. Основы научных исследований : учеб. пособие / М.Ф. Шкляр. - 6-е изд. - М. : Дашков и Ко, 2016. - 208 с.
4. Михеева, Е. В. Информационные технологии в профессиональной деятельности : учеб. пособие / Е.В. Михеева. - 14-е изд., стер. - М. : Академия, 2016. - 384 с.
5. Любанович Билл. Простой Python. Современный стиль программирования. — СПб.: Питер, 2016. — 480 с.