**Модуль 2. Базовый**

**Урок 4. Функции**

**Ключевые понятия: def, lambda и return, создание и вызов функций. Создание мини-программ на основе функций. Встроенные функции Python и рекурсия**

Функции можно назвать небольшими подпрограммами, куда можно вынести повторяющийся код и обращаться к нему, когда это будет нужно. Функции значительно облегчают построение программ, так как нам не надо копировать однотипный код множество раз, а можно просто воспользоваться одной общей функцией.

Функции очень часто используются в программировании. В предыдущих занятиях мы уже встречались с встроенными функциями такими как ( print(), input(), len() и т.д. ). Попробуем разобраться что такое функции и как написать свою функцию.

**Что такое функция?**

* Часть кода, для которого мы (даем) присваиваем имя.
* Написав один раз, можно вызывать из разных мест программы.
* Функции позволяют структурировать код, разбив его на отдельные действия.
* Функции делают код более читабельным и упрощают поиск ошибок.

Многие путают функции и методы в классах (ООП) и не понимают отличий между ними. На самом деле отличий нет, так как что методы, что функции являются одним и тем же. Функции что записаны вне классов называют функциями, а функции что записаны внутри классов называются методами.

Точно такая же ситуация обстоит с переменным. В классах переменные называются полями, а вне классов - переменными.

В Python функции создаются при помощи ключевого слова def. Каждая функция может иметь какие-либо параметры или же не иметь их вовсе. Функции способны что-либо возвращать в ходе выполнения кода, если это требуется.

def some\_test(): # Функция без параметров

 x = 23

 return x # Функция возвращает что-либо

def some\_test\_2(a, b, c): # Функция с 3 параметрами

 return a \* b \* c # Функция возвращает результат умножение чисел

def print\_something(word, prefix): # Функция с 2 параметрами

 print (prefix, "-", word)

 pass # Функция ничего не возвращает

**Пустая функция (stub)**

Иногда, когда вы пишете какой-нибудь код, вам нужно просто ввести определения функции, которое не содержит в себе код.

def empty\_function():

 pass

Оператор pass, который тут использован - это пустая операция, это означает, что когда оператор pass выполняется, не происходит ничего.

**Аргументы функции**

 Функции могут возвращать другие функции, тем самым вызывая их. Чтобы обратиться к функции необходимо прописать её название и передать параметры, если таковы имеются:

res = some\_test\_2 (2, 67, 12)

В примере выше результат выполнения функции будет помещен в переменную res. Далее с переменной можно работать как с обычным значением в программе.

В качестве параметров в функции могут быть переданы и арифметические выражения.

**\*args и \*\*kwargs**

Вы также можете настроить функцию на прием любого количества аргументов, или ключевых аргументов, при помощи особого синтаксиса. Чтобы получить бесконечное количество аргументов, мы используем \*args, а чтобы получить бесконечное количество ключевых аргументов, мы используем \*kwargs. Сами слова “args” и “kwargs” не так важны. Это просто сокращение. Вы можете назвать их \*lol и \*omg, и они будут работать таким же образом. Главное здесь – это количество звездочек. Обратите внимание: в дополнение к конвенциям \*args и \*kwargs, вы также, время от времени, будете видеть andkw. Давайте взглянем на следующий пример:

def many(\*args, \*\*kwargs):

 print( args )

 print( kwargs )

many(1, 2, 3, name="Mike", job="programmer")

# Результат:

# (1, 2, 3)

# {'job': 'programmer', 'name': 'Mike'}

Сначала мы создали нашу функцию, при помощи нового синтаксиса, после чего мы вызвали его при помощи трех обычных аргументов, и двух ключевых аргументов. Функция показывает нам два типа аргументов. Как мы видим, параметр args превращается в кортеж, а kwargs – в словарь. Вы встретите такой тип кодинга, если взгляните на исходный код Пайтона, или в один из сторонних пакетов Пайтон.

**Область видимость и глобальные переменные**

Концепт области (scope) в Пайтон такой же, как и в большей части языков программирования. Область видимости указывает нам, когда и где переменная может быть использована. Если мы определяем переменные внутри функции, эти переменные могут быть использованы только внутри это функции. Когда функция заканчиваются, их можно больше не использовать, так как они находятся вне области видимости.

**Анонимные функции**

Если функция не должна выполнять большой объем кода, то можно воспользоваться анонимной функцией. Для этого потребуется функция lambda.

Пример создания «lambda» функции:

multiple = lambda a, b: a \* b

multiple(7, 12)

Подобная функция не имеет названия, но её можно присвоить к переменной, которую в дальнейшем необходимо вызывать как обычную функцию.

**Рекурсия функций**

Рекурсия появляется когда вызов функции повторно вызывает ту же функцию до завершения первоначального вызова функции. Например, рассмотрим известное математическое выражение x! (т.е. факториал). Факториал определяется для всех неотрицательных целых чисел следующим образом:

Если число равно 0, то будет 1.

В противном случае ответом будет то, что число умножается на факториал на единицу меньше этого числа.

В Python наивная реализация факториала может быть определена как функция следующим образом:

def factorial(n):

 if n == 0:

 return 1

 else:

 return n \* factorial(n - 1)

Одна из самых больших проблем для молодых программистов – это усвоить правило «не повторяй сам себя». Суть в том, что вы не должны писать один и тот же код несколько раз. Когда вы это делаете, вы знаете, что кусок кода должен идти в функцию. Одна из основных причин для этого заключается в том, что вам, вероятно, придется снова изменить этот фрагмент кода в будущем, и если он будет находиться в нескольких местах, вам нужно будет помнить, где все эти местоположения и изменить их. Использование функций значительно упрощает работу с повторяющимся кодом.

**Задание:**

Написать функцию month\_to\_season(), которая принимает 1 аргумент - номер месяца - и возвращает название сезона, к которому относится этот месяц. Например, передаем 2, на выходе получаем 'Зима'.