**Изучаем Python**

**Марк Лутц 2011, издание O’ Reilly**

**Том 2**

***Часть. 4. Функции.*** *стр. 459*

**Глава 16. Основы функций.** стр. 461

Функция - это средство, позволяющее группировать наборы инструкций так, что в программе они могут запускаться неоднократно.

Они могут вычислять результат и иметь входные параметры. Оформление операций в виде функций - очень удобный инструмент, который мы можем использовать в самых разных ситуациях.

Т.о. функции устраняют необходимость вставлять в программу избыточные копии блоков одного и того же программного кода, он заменяется одной единственной функцией. Кстати, при изменении программы достаточно изменить только функцию, а не все повторяющиеся блоки.

Функции - самые основные программные структуры в ЯП Питон, обеспечивающие многократное использование программного кода и уменьшающие его избыточность. Также это средство проеткирования, разбивающее сложжную систему на простые и легкие части.

**Таблица 16.1 Инструкции и выражения, имеющие отношения к функциям.** стр. 461

**Зачем нужны функции?**

Это универсальное средство структурирования программы, в других ЯП функции могут называться - подпрограммами или процедурами.

Две основные роли:

- *Максимизировать многократное использование программного кода и минимизировать его избыточность.* Функции в Питон являют простейший способ упаковки логики выполнения, которая может использоваться в разных местах программы и более чем один раз. Ф-ции позволяют обощать и группировать код, который потом можно использ. много раз. Т.к. ф-ции позволяют поместить реализацию в одно место и использовать её в разных местах, они явл-ся основным инструментом структуризации - дают возможность уменьшить избыточность кода.

- *Процедурная декомпозиция.* Ф-ция помогает разбить сложную систему на части. Ведь гораздо проще создать решение малых задач по отдельности, чем реализовать большую и сложную задачу целиком и разом. Вообще ф-ции описывают "как делать", а не "зачем делать".

**Создание функций.** стр. 463

Ранее мы уже пользовались функциями, напр. создавали объект с помощью ф-ции open, использовали len и мн. др.

В этой главе мы узнаем как создавать новые функции. Написанные нами функции ведут себя так же ка ки встроенные - они могут вызываться в выражениях, получать значения и возвращать результаты.

Кстати, ф-ции в Питон ведут себя не так как в С.

Краткое описание основных концепций и понятий функций:

- **def - это испольняемый программный код**. Ф-ции в Питон создаются с помощью инструкции def. В отличие от компилирующихся ЯП, типа С, def относится к классу исполняемых инструкций - ф-ция не существует, пока интерпретатор не доберется до инструкции def и не выполнит ее. Допустимо (иногда и полезно) вкладывать def внутрь инструкций if, циклов while и даже других инструкций def. В самом типичном случае def вставляется внутрь модулей и генерируют функции во время первой операции импортирования модуля.

- **def создает объект и присваивает ему имя.** Интерпретатор встречает и выполняет def, при это создавая новый объект-функцию и связывая его с именем ф-ции. Т.е. имя становится ссылкой на объект-функцию. Объект-функция может быть свзязано с неск-ми именами, может сохраняться в списке и т.д. Функциям можно прикреплять атрибуты, с данными.

- **Выражение lambda создает объект и возвращает его в виде результата.** Ф-ции могут создаваться с пом. выражения lambda. Это дает нам создавать встроенные определения ф-ций там, где синтаксис не позволяет использ. инструкцию def.

- **return - передает объект результата вызывающей программе.** Когда ф-ция вызывается, вызывающая программа приостанавливает работу, пока ф-ция не завершится и не вернет управление. Ф-ции вычисляющие значения возвпращают ег ос помощью return - возвращаемое значение становится рез-том вызова ф-ции.

- **yield - передает объект результата вызывающей программе и запоминает, где был произвдеен возврат.** Ф-ции, называемые генераторами, могут использовать yield и сохранять состояние так, чтобы работа ф-ции могла быть возобновлена позднее.

- **Аргументы передаются посредством присваивания (в виде ссылок на объекты).** В Питон аргуменыт передаются ф-циям посредством операции присваивания (т.е. в виде ссылок на объекты). В Питон и вызывающая программа и и ф-ция совместно использ. ссылку на объект, но здесь нет никакого совмещения имен. Изменение имени аргумента не меняет имени в вызывающ. программе, но модификация изменямых объектоввнутри ф-ции может приводить к изменению объектов в вызывающей программе.

- **global - объявляет переменные, глобальные для модуля, без присваивания им значений.** По умолч. все имена, прсиваивание которым произв-ся внутри ф-ции, явл-ся локальными для этих ф-ций и существуют только во время выполнения этих ф-ций. Чтобы прсивоить значение имени в объемлющем модуле, ф-ция должна объявить его с помощью инструкции global. Поиск имен всегда производится в некоторой области видимости - там, где хранятся переменные, - а операция присваивания связывает имена с областями видимости.

- **nonlocal - объявляет переменные, находящиеся в в области видимости объемлющей функции, без присваивания им значений.** Это позволяет использ. объемлющие ф-ции, как место хранения информации о состоянии - информация восстанавливается в момент вызова ф-ции, при этом отпадает необходимость использ-ть глобальные переменные.

- **Аргументы получают свои значения (ссылки на объекты) в результате выполнения операции присваивания.** При передаче аргументво ф-ции выполняется операция прсиваивания значений (т.е. ссылок на объекты). Передача объектов в функции производится по ссылкам, но это не означает, что созд-ся псевдонимы имен. Изменение аргуменат фнутри ф-ции не влечет за собйо изменение соответствующего имени в вызывающей программе, но изменения в изменемых объектах отразятся в вызывающей программе.

- **Аргументы, возвращаемые значения и переменные не объявляются.** Как и во всем Питоне, на ф-ции не накладываются ограничения по типу. Фактически, никакие элементы ф-ций не требуют предварительного объявления, мы можем передавать аргументы любых типов, возвращать из ф-ции объекты любого типа и т.д. Т.о. одна и та же ф-ция может применяться к объектам различных типов. Допустимыми явл-ся любые объекты, поддерживающие совместимые интерфейсы (методы и выражения), независимо от конкретного типа.

**Инструкция def.**

def создает объект ф-ции и связывает его с именем. Общий вид таков:

def <name>(arg1, arg2, ... argN):

<statements>

Мы видим, что def явл-ся типичной составной инструкцией, с заголовком и блоком инструкций, т.н. - телом ф-ции, т.е. программным кодом, который выполняется интерпретатором всякий раз, когда ф-ция вызывается.

В заголовке определяется имя ф-ции и список из нуля и более аргументов (др. навазние - параметры). Имена аргументов в заголовке будут связаны с объектами, передаваемыми в ф-цию, в точке вызова.

Тело ф-ции часто содержит инструкцию return:

def <name>(arg1, arg2, ... argN):

...

return <value>

- return может располагаться в любом месте в теле ф-ции - она завершает работу ф-ции и передает результат вызывающей программе.

Инструкция return явл-ся необязательной, если ее нет ф-ция завершается когда достигается конец тела ф-ции. С технич. точки зрения ф-ция без return автоматически возвращает объект None, однако это значение обычно просто игнорируется.

**Инструкции def исполняются во время выполнения.**

def - исполняемая инструкция, когда она выполняется, она создает новый объект ф-ции и присваивает этот объект имени.

**Не забываем!** - Все, что есть в Питон относится к времени выполнения, здесь нет понятия времени компиляции.

Т.к. def инструкция, то можеть быть везде где могут быть инструкции - даже внутри других инструкций:

if test:

def func(): # Определяет функцию таким способом

...

else:

def func(): # Или таким способом

...

...

func() # Вызов выбранной версии

Чтобы понять этот фрагмент обратим внимание, что def напоминает инструкцию присваивания = т.к. она просто выполняет присваивание во время выполнения.

В отличие от С ф-ции в Питон не должны быть полностью поределены к моменту запуска программы. Т.е. инструкции def не интерпретируются, пока не будут достигнуты и выполнены потоком выполнения, а код внутри def не выполняется, пока ф-ция не будет вызвана позднее.

Важно не имя функции, а объект, на который ссылается имя:

othername = func # Связывание объекта функции с именем

othername() # Вызов функции

- мы связали нашу функцию с другим именем и вызвали с использованием нового имени.

Ф-ция - это обычные объекты, они явно записываются в память во время выполнения программы.

Ф-ции позволяют присвоединять любые атрибуты, с информацией, напр. для сохранения инф-ии:

def func(): ... # Создает объект функции

func() # Вызывает объект

func.attr = value # Присоединяет атрибут к объекту

**Первый пример: определения и вызовы.** стр. 466

Напишем первый пример функции. Как видим ф-ция имеет две стороны: определение (инструкция def, которая создает ф-цию) и вызов (выражение, которое предписывает интерпретатору выполнить тело функции).

**Определение.**

Ф-ция times, которая возвращает рез-т обработки двух аргументов:

>>> def times(x, y):

return x \* y

...

>>>

- когда интерпретатор достигает инструкции def и выполняет ее, он создает объект функции, в который упакует программнфый код ф-ции и свяжет объект с именем times. Как правило такие ф-ции размещаются в файлах модулей и выполняются по время импортирования, но наша небольшая и можно создать ее прямо в интерактивной оболочке.

**Вызов.**

После выполнения def появл-ся возможность вызвать ф-цию, добавив круглые скобки после ее имени. В скобках указ-ся аргументы:

>>> times(2, 4) # В круглых скобках аргументы: 2 и 4

8

- имени x присваивается значение 2, а имени y - значение 4, после чего запускается тело ф-ции. В нашем случае значение возвращается инструкцией return (собственно она и есть тело нашей ф-ции).

Теперь мы можем указать, что тело функции выполняется в момент ВЫЗОВА функции.

Результат ф-ции можно присвоить переменной:

>>> x = times(3.14, 4)

>>> x

12.56

А теперь передадим ф-ции объекты разных типов:

>>> times('Ni', 4) # Получилась операция повторения

'NiNiNiNi'

- т.о. мы видим, что смысл ф-ции times и тип возвращаемого значения определяется аргументами, которые ей передаются.

**Полиморфизм в языке Питон.**

Как мы уже подчеркнули, смысл выражения x\*y полностью зависит от типа объектов x и y - т.е. это может быть как умножение, так и повторение.

В Питон именно объекты определяют синтаксический смысл операции.

Такого рода зависимость от типов известна как полиморфизм - термин, означающий, что смысл операции зависит от типов обрабатываемых объектов. Фактически все операции в Питон явл-ся полиморфическими.

Т.о. одна функция может автоматически применяться к целой категории типов объектов. Если объект поддерживает ожидаемые методы и операторы выражений, он будет совместим с логикой функции.

Напр. наша ф-ция times, - любые два объекта, поддерживающие оператор \*, смогут обрабатываться ф-цией times и не важно что это за объекты и когда были созданы.

Если же будут переданы объекты, НЕ поддерживающие ожидаемый интерфейс, интерпретатор автоматически возбудит исключение.

Вообще интерфейс - это набор методов и операций, которые используются функцией.

Т.о. мы видим отличие Питона от статической типизации - программный код на Питоне не делает предположений о конкретных типах данных. Вообще, при программировании на Питон во внимание принимаются интерфейсы объектов, а не типы данных.

Конечно такая модель полиморфизма предполагает необходимость тестирования кода на наличие ошибок, напр. на соответствие типов и интерфейсов и т.п. Но тестирование оно много где применяется в больших масштабах, так что это не совсем даже и недостаток.

**Второй пример: пересечение последовательностей.** стр. 469

Вспомним цикл for, который выбирал элементы, общие для двух строк. Мы его усовершенствуем, сделав функцию.

**Определение.**

Плюсы оформления нашего цикла в виде функции:

- Сделав код в виде ф-ции, появится возможность использовать его столько раз сколько нужно;

- Т.к. вызывающая программа может передавать ф-ции произвольные аргументы, ф-ция может использ-ся с любыми двумя последовательностями (или итерируемыми объектами) для полученяи их пересечения;

- Когда логика работы в виде ф-ции достаточно изменить код всего в одном месте, чтобы изменить способ полученяи пересечения;

- Поместив ф-цию в файл модуля, её можно импортировать и использовать в любой программе на нашем ПК.

Итак, наш код, обернутый в функцию, превращается в утилиту нахождения пересечения:

def intersect(seq1, seq2):

res = []

for x in seq1:

if x in seq2:

res.append(x)

return res

- все понятно: имя, аргументы, тело ф-ции, return. Двигаемся дальше.

**Вызов.**

Мы поместили нашу ф-цию в файл и для её создания импортируем наш модуль. Как только def будет выполнена, можно будет вызывать ф-цию и передавать ей два объекта последовательностей в круглых скобках:

>>> def inter(seq1, seq2):

res = []

for x in seq1:

if x in seq2:

res.append(x)

return res

>>> inter(s1, s2)

['S', 'A', 'M']

- Как видим я набрал функцию прямо в IDLE, хотя в книге написано, что можно создать её в модуле и импортировать его, но тут начинается самое интересное...

- Я создал файл tnt.py и импортировал его, вызвав dir(tnt) я увидел, что атрибут inter присутствует у модуля tnt, НО - при попытке вызвать ф-цию inter:

>>>inter(s1, s2)

...

NameError: name 'inter' is not defined

- я получал сообщение об ошибке "NameError: name 'inter' is not defined", не представляю с чем это связано, возможно с тем, что tnt.py расположен в моей домашней директории.

- Самое печальное то, что я не знаю как с этим бороться.

*Замечание! –* Я все-таки разобрался с этой проблемой!!! см. далее

**Разобрался!!!!!**

Мы импортируем модуль, а потом указывая имя модуля и его аргументы мы их используем. А в нашем случае ф-ция inter() как раз явл-ся аргументом модуля tnt (как мы уже писали выше о проверке dir()) и в конечном итоге всё работает:

>>> import tnt

>>> s1 = 'spam'

>>> s2 = 'scam'

>>> tnt.inter(s1, s2)

['s', 'a', 'm']

- это **рекомендованный** способ использования модулей.

Или, чтобы не писать каждый раз имя модуля, используем конструкцию from import:

>>> s1 = 'spam'

>>> s2 = 'scam'

>>> from tnt import inter

>>> inter(s1, s2)

['s', 'a', 'm']

- Вот такие вот подводные камни. Но всё-таки рекомендуется использовать просто import.

Все-таки нужно повторить первую часть моих записок, а то вот уже забыл правила использования атрибутов модулей. А если бы всё это помнил, не пришлось бы целый час сидеть в инете и искать ответ на "загадочную" ошибку с импортированным модулем.

Кстати наша функция, если честно, работает слишком медленно и в принципе не особо нужна, её можно заменить единственным выражением генератора списков:

>>> [x for x in s1 if x in s2]

['S', 'A', 'M']

- но всё же это простой пример применения функций к различному диапазону типов (различных типов последовательностей).

**Еще о полиморфизме.**

Как и другие ф-ции, наша inter тоже полиморфна, пример:

>>> import tnt

>>> x = tnt.inter([1, 2, 3], (1, 4)) # Смешивание типов

>>> x # Объект с результатом

[1]

- мы передали ф-ции список и кортеж, два разных типа - и она все равно сработала!

Т.е. inter будет выполнять итерацию по объектам послетовательностей любых типов, если они поддерживают ожидаемый интерфейс.

В нашем случае для inter, это означает, что первый объект должен поддерживать циклы for, а второй - поддерживать оператор in, выполняющий проверку на вхождение.

Так наша ф-ция поддерживает любые итераторы с ожидаемымми интерфейсами, даже файлы, словари и объекты на основе классов с перегрузкой операторов.

Опять же - если объекты не поддерживают ожидаемые интерфейсы, то будет возбуждено исключение.

**Локальные переменные.**

Рассмотрим переменные в нашем примере, и соотв. функции.

Переменная res внутри ф-ции inter - это то, что в Питон называется - локальной переменной, - имя, которое доступно только программному коду внутри инструкции def и существует только во время выполнения ф-ции.

Фактически, любые имена, которым были присвоены какие-либо значения внутри ф-ции, по умолчанию классифицируются как локальные переменные.

Почти все имена в ф-ции inter явл-ся локальными переменными:

- Переменная res явно участвует в операции присваивания, - поэтому она локальная;

- Аргументы передаются через присваивание, - поэтому seq1 и seq2 тоже локальные переменные;

- Цикл for присваивает элементы переменной, поэтому имя х также явл-ся локальным.

Все эти локальные переменные появл-ся в момент вызова ф-ции и исчезают, когда ф-ция возвращает управление - return возвращает объект результата, а имя res исчезает.