

Программирование —»Python на Symbian S60: изучение функций, написание программы +пример Calculator.py

Номер журнала:

Автор:

Газетдинов Альберт

Python на Symbian S60: изучение функций, написание программы + пример Calculator.py

Скачать пример: программа Calculator.py

Начнём углубленное изучение языка Python с функций

Функции в Python, как и в любом другом языке программирования, предусмотрены **для упрощения работы** с часто повторяющимися действиями в программе.

Функция состоит из **имени**, **аргументов** и **тела**, выглядит она так:

```
def name(arguments):  
    тело функции
```

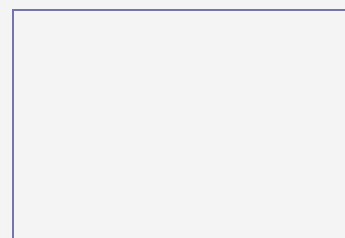
Как видно, после def указывается имя функции, а в скобках – её аргументы. Если аргументов для функции не предусмотрено, то скобки оставляются пустыми. В конце строки обязательно ставят двоеточие.

При каждом вызове функции выполняются записанные в её тело выражения. Следует отметить, что **тело функции надо выделять**. В языке программирования Pascal это делают с помощью инструкций begin...end, а в C – с помощью фигурных скобок {...}.

Наберём в интерактивной консоли нашу первую функцию. Сначала вводится "def hello():". После нажатия на джойстик произойдет переход на новую строку и вместо привычного ">>>" появится приглашение вида "...". На этой строке необходимо сделать отступ, обычно 4-8 пробелов (однако число пробелов может быть любым). Только затем можно печатать тело функции «print "Hello!"». Если нажать еще раз на джойстик, то снова выйдет приглашение "...". Так как выражений, принадлежащих функции, больше нет, нажимаем на джойстик. Переход на новую строку без наличия отступа уведомляет интерпретатор о завершении ввода тела функции. Теперь введите "hello()" и нажмите на джойстик, что приведет к выводу в консоль текста "Hello!" (на экране смартфона это выглядит так, как представлено на Рис.1).

Пример:

```
>>> def hello():  
...     print "Hello!"  
...  
>>> hello()  
Hello!
```



```
>>>
```

Рис. 1.

Наша первая функция не содержала **аргументов**. Следующий шаг – научиться их использовать!

Пример:

```
>>> def hello(name):  
...     print "Hello",name,"!"  
...
```

```
>>> hello("Albert")
```

```
Hello Albert!
```

```
>>>
```

Или еще один пример, уже с числами:

```
>>> def summa(a,b):  
...     print "a + b =",a+b  
...
```

```
>>> summa(1,2)
```

```
a + b = 3
```

```
>>>
```

Переменные, которые используются в функции, называются **локальными**. Это означает, что после того, как функция отработала, её переменные попросту уничтожаются. Любопытно, что заданные в функции переменные не конфликтуют с **глобальными**. Поясним на примере:

```
>>> c=5  
>>> def check():
```

```
...     c=10  
...     print c  
...
```

```
>>> check()
```

```
10
```

```
>>> print c
```

```
5
```

```
>>>
```

Как видно, работа с переменной внутри функции никак не сказывается на её однофамильца вне этой функции. В случае, когда внутри функции необходимо использовать глобальные переменные, используется оператор **global**, за которым следует их список через запятую.

Пример:

```
>>> pi=3.14
```

```
>>> def len(radius):
```

```
...     global pi  
...     print "Dlina kruga =",pi*radius  
...
```

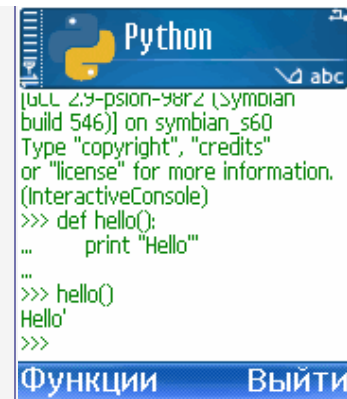


Рис. 1

```
>>> len(2)
>>> Dlina kruga = 6.28
>>>
```

Нетрудно представить, что результат работы функции, который был сохранен в переменной, может понадобиться для дальнейшей работы, но после завершения функции, как было сказано, эта переменная будет удалена. Для этого случая предназначена команда **return**, возвращающая указанные переменные. Пример:

```
>>> def summa(a,b):
...     c=a+b
...     return c
...
>>> summa(1,2)
3
>>>
```

В нашем примере функция возвращает "с" в любом случае, однако если вдруг понадобится, в зависимости от каких-либо условий, совершить другое действие? Вот пример с решением проблемы:

```
>>> def summa(a,b):
...     if a<0:
...         print "Error! a",a,"< 0"
...         return
...     elif b<0:
...         print "Error! b",b,"< 0"
...         return
...     else:
...         c=a+b
...         return c
...
>>> summa(1,2)
3
>>> summa(-1,2)
'Error! -1 < 0'
>>> summa(1,-2)
'Error! -2 < 0'
>>>
```

В примере функция `summa()` успешно возвращает результат, только если оба аргумента больше нуля. Для определения этого используется конструкция `if...elif...else` и условия.

Начинается конструкция с `if`, и если условие, записываемое после него, истинно (например, `a<0`), то выполняется нижеследующий блок выражений (например, `print "Error! a",a,"< 0"`). Эта часть конструкции должна присутствовать всегда.

Если условие является ложным, то проверяются все elif. В нашем случае он был один, однако их может быть неограниченное количество и интерпретатор будет проверять их все, пока в каком-либо из них условие не станет истинным. Т.е. если бы параметров было на два больше - `def summa(a,b,c,d)` - то необходимо было бы дописать еще две elif: `>>> def summa(a,b,c,d):`

```
...     if a<0:
...         print "Error! a",a,"< 0"
...         return
...     elif b<0:
...         print "Error! b",b,"< 0"
...         return
...     elif c<0:
...         print "Error! c",c,"< 0"
...         return
...     elif d<0:
...         print "Error! d",d,"< 0"
...         return
```

И т.д...

Если же ни в одном из if и elif условие не будет истинно, то выполняется блок выражений, следующий за else. В нашем случае это происходит, только если все аргументы больше нуля. И еще два замечания:

1) команда return может быть использована для выхода в любом месте тела функции, причем даже без аргументов (в этом случае функция по умолчанию возвращает значение None);

2) в зависимости от ситуации, в if и elif могут быть использованы условия проверки "<" для меньше, ">" для больше, ">=" для больше или равно, "<=" для меньше или равно, "==" для равно, "<>" или "!=" для не равно.

Перейдем к рассмотрению **типов данных** и начнем со списков.

Список в языке Python предназначен для группирования нескольких значений. Наиболее близким аналогом из других языков программирования является понятие массив. Создается список путем заключения нужных значений, перечисленных через запятую, в квадратные скобки:

```
>>> [1, 2, 3]
[1, 2, 3]
>>> ["one", "two", "three"]
["one", "two", "three"]
>>>
```

Первый пример – список четырех целых чисел, а второй – список трех строк. Элементы списков вовсе не обязательно должны быть одного типа. Следующий список содержит строку, целое, вещественное число и другой список:

```
>>> ["hello", 5, 2.0, [10, 20]]
```

```
["hello", 5, 2.0, [10, 20]]
```

```
>>>
```

Список, являющийся элементом другого списка, называют **вложенным**.

Кроме того, Python предоставляет возможность быстрого создания списков целых

значений, без необходимости их надо перечислять:

```
>>> range(1,5)
```

```
[1, 2, 3, 4]
```

```
>>>
```

В данном примере функция **range()** принимает два целых аргумента и возвращает список, который содержит все целые числа в промежутке между заданными значениями, включая первое и исключая второе.

Существует еще два способа вызова функции **range()**. Если ей передано только одно значение, то в результате она вернет список с целыми значениями от 0 до N, где N – значение параметра:

```
>>> range(10)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>>
```

Если же **range()** вызвана с тремя аргументами, то последний из них рассматривается как размер шага. Т.е. в полученном списке значения будут идти не подряд, а через промежутки:

```
>>> range(1, 10, 2)
```

```
[1, 3, 5, 7, 9]
```

```
>>>
```

Список, не содержащий ни одного элемента, называют пустым. Он обозначается пустыми квадратными скобками **[]**.

Как и любые другие значения, списки могут сохраняться в переменных:

```
>>> numbers = [17, 123, 537]
```

```
>>> empty = []
```

```
>>> print numbers, empty
```

```
[17, 123, 537] []
```

```
>>>
```

Значение в списке может быть получено по порядковому номеру – **индексу**. Индексом может быть любое выражение, возвращающее целое число, в том числе отрицательное. Нумерация индексов начинается с нуля. Если индекс меньше нуля, то отсчет индекса будет начат с конца списка:

```
>>> numbers[0]
```

```
17
```

```
>>> numbers[-1]
```

```
537
```

```
>>>
```

После создания списка его элементы можно изменять:

```
>>> numbers[1] = 5
```

```
>>> numbers
```

```
[17, 5, 537]
```

```
>>>
```

Для вычисления длины списка используется функция `len()`:

```
>>>mylist = ["one", "two", "three", "four", [1, 2, 3, 4]]
```

```
>>>len(mylist)
```

```
5
```

```
>>>
```

Обратите внимание, что если список содержит в качестве элемента другой список, то этот вложенный список будет считаться как один элемент.

В Python есть еще один тип данных, называемый **кортеж**. Он полностью аналогичен списку – за исключением того, что элементы изменять не допускается.

Для того чтобы создать кортеж, необходимо заключить в обычные скобки нужные значения через запятую:

```
>>> tuple = ("a", "b", "c", "d", "e")
```

```
>>> len(tuple)
```

```
5
```

```
>>> tuple[0]
```

```
'a'
```

```
>>>
```

Как видно, можно получить размер кортежа и значение любого элемента по индексу – так же, как и в случае со списком. Однако выражение вида `tuple[0]="z"` приведет к ошибке – повторяю, элементы кортежа изменять нельзя.

Таким образом, было рассмотрены понятия список и кортеж. Необходимость изучения была продиктована тем, что эффективное их использование делает программу элегантной и простой в понимании. К тому же множество функций принимают списки и кортежи в виде аргументов или возвращают их как результат.

Теперь перейдем к нашей программе `my_program`. Там было две строки:

```
import appuifw
```

```
appuifw.note(u"Hello World!")
```

Если выполнить (т.е. ввести и нажать на джойстик) сначала первую строку, то ничего не произойдет. На самом деле инструкция **import** подключает модуль `appuifw`, после чего его возможности будут доступны в программе. А направлены они на создание **интерфейса** программы, т.е. инициирование всплывающих сообщений, окошек с запросами для ввода данных, холста для рисования и многое другое. Для вызова функций данного модуля сначала необходимо указать собственно имя модуля, затем точку и, наконец, имя функции.

После выполнения второй строки появится окошко с сообщением "Hello World!". Ответственно за это функция **note**, имеющая один обязательный аргумент – текст сообщения. На данном этапе обучения примем, что текст должен быть на английском и обязательно начинаться с символа "u".

У этой функции есть и второй необязательный аргумент, который указывает характер выводимого сообщения. По умолчанию он равен "info", т.е. подразумевается, что сообщение информирует пользователя о чем-либо. Кроме него возможны значения "error" (часто используется при выводе сообщения об ошибке) и "conf" (используется, например, когда пользователю необходимо сообщить о завершении операции). Подставьте эти значения сами и посмотрите на результат.

Также данный модуль предоставляет функцию **query**, которая пригодится в случае, когда программе требуется получить какую-то информацию от пользователя. Например, в результате выполнения `appuifw.query(u"Give UID", "text", u"0xxxxxxxxx")` появится абсолютно то же окно запроса, что и при вводе UID в программе AppMgr при создании приложения (Рис. 2). После нажатия на «ОК» в консоли отобразятся введенные данные. Т.е. функция вернула результат своей работы, и эти данные впоследствии могут быть использованы по ходу работы программы.



Рис. 2

Разберем параметры функции **query**: первый – имя окна запроса (заголовок), второй – тип вводимых данных, третий – значение по умолчанию, которое будет отображено в окне при его создании. Сразу отмечу, третий аргумент необязателен, т.е. если он будет отсутствовать, то окно появится пустое изначально. Теперь подробнее о втором аргументе. Кроме "text", существуют варианты:

1) "code" – предназначено для запроса пароля, т.е. в окне введенные данные будут отображены звездочками;

2) "number" – предназначено для запроса целых чисел;

3) "date" – предназначено для запроса даты (день, месяц, год);

4) "time" – предназначено для запроса времени (час, минута);

5) "float" – предназначено для запроса вещественных чисел;

6) "query" – предназначено для запроса подтверждения.

При этом:

1) "code" и "text" возвращают набранный пользователем текст;

2) "date" и "time" возвращают набранную дату и время в секундах;

3) "number" возвращает целое число, а "float" – вещественное;

4) если тип данных указан "float", то значение по умолчанию не может быть установлено.

Остановлюсь на последнем. Выполнение `appuifw.query(u"You want exit?", "query")` приведет к появлению окна выбора с заголовком "You want exit?" и

с двумя вариантами действия: "ОК" (над левой софт-клавишей) и "Отмена" (над правой софт-клавишей). Т.е. данный вид запроса может быть использован, например, для подтверждения действий пользователя (Рис. 3). В случае нажатия на "ОК" `query` вернет число 1, в противном случае ничего не возвращается. Пример:

```
>>> def exit():
...     if appuifw.query(u"You want exit?", "query")==1:
...         appuifw.note(u"OK")
...     else:
...         appuifw.note(u "Cancel")
...
>>>
```

Вызов `exit()` в любом месте программы инициирует появление окна с вопросом `You want exit?`, причем результат выбора будет немедленно отмечен соответствующим сообщением.

В случае, когда имеется необходимость предусмотреть выход из программы, модуль `appuifw` предлагает использовать функцию `app.set_exit()`. Если предыдущий пример переписать и вместо `appuifw.note(u"OK")` ввести `appuifw.app.set_exit()`, то после вызова `exit()` и нажатия на «ОК», произойдет ожидаемый выход из программы.

Последняя описываемая в этой статье возможность – создание меню. Для этого используется переменная `app.menu`. Пример:

```
>>> def exit():
...     if appuifw.query(u"You want exit?", "query")==1:
...         appuifw.app.set_exit()
...
>>> appuifw.app.menu([(u"Exit",exit)])
>>>
```

Теперь при нажатии на «Функции» будет появляться список предлагаемых действий с одним пунктом – `Exit` (смотри рис. 4). Его выбор приведет к появлению запроса о выходе, что и произойдет при нажатии на «ОК».



Рис. 3



Рис. 4

Подробно о значении, которое может быть присвоено переменной `appuifw.app.menu`. Как видно из примера, для создания меню используется список кортежей. В нашем случае список состоял из одного кортежа, которое обеспечило создание одного пункта меню – “Exit”. Нажатие по этому пункту приводит к вызову функции `exit`. Если необходимо создать несколько пунктов, то указывайте кортежи через запятую. В общем, переменной может быть присвоено значение вида `[(title1, callback1), (title2, callback2), ..., (titleN, callbackN)]`. Каждый кортеж состоит из одного элемента в виде строки, отвечающего за имя пункта, и второго элемента в виде имени функции, вызываемого сразу после выбора пункта.

Полученной информации достаточно для создания простейших программ – например, калькулятора на четыре арифметических действия: сложение, вычитание, умножение и деление. Данный пример называется `Calculator` и находится в архиве, идущем вместе со статьей. Описывать принцип его работы я не буду, так как в файле `Calculator.py` имеются комментарии (Рис. 5-7).

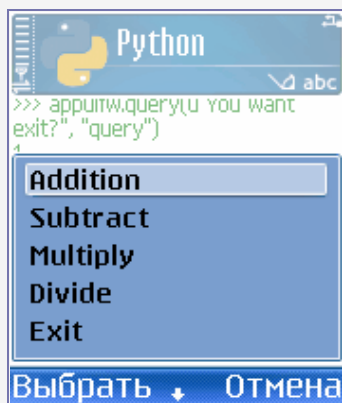


Рис. 5

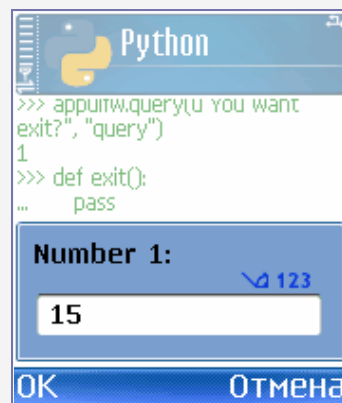


Рис. 6

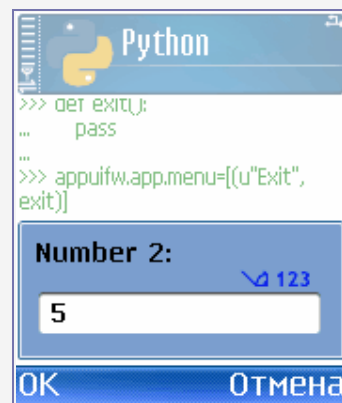


Рис. 7

Пример также ярко доказывает, возможно, главное преимущество Python – при должном опыте можно написать программу с любым нужным функционалом, что поднимает удобство работы с Symbian смартфоном до невообразимых высот. Например, тот же Calculator в умелых руках способен превратиться в мощный инженерный инструмент, подстроенный под нужды определенного пользователя. Причем за аналог в виде готового приложения *.sis пришлось бы отдать разработчику свои кровные деньги.

Чтобы достичь такого уровня программирования, читайте продолжение серии статей и подкрепляйте полученные знания собственными программами.

Скачать пример: программа Calculator.py