Подробное введение в Python часть 2

1. Инструкция if-elif-else, проверка истинности, трехместное выражение if/else

Условная инструкция if-elif-else (её ещё иногда называют оператором ветвления) - основной инструмент выбора в Python. Проще говоря, она выбирает, какое действие следует выполнить, в зависимости от значения переменных в момент проверки условия.

1.1 Синтаксис инструкции if

Сначала записывается часть if с условным выражением, далее могут следовать одна или более необязательных частей elif, и, наконец, необязательная часть else. Общая форма записи условной инструкции if выглядит следующим образом:

```
if test1:
    state1
elif test2:
    state2
else:
    state3
```

Простой пример (напечатает 'true', так как 1 - истина):

```
>>> if 1:
...     print('true')
... else:
...     print('false')
...
true
```

Чуть более сложный пример (его результат будет зависеть от того, что ввёл пользователь):

```
a = int(input())
if a < -5:
    print('Low')
elif -5 <= a <= 5:
    print('Mid')
else:
```

```
    print('High')
```

Конструкция с несколькими elif может также служить отличной заменой конструкции switch - case в других языках программирования.

1.2 Проверка истинности в Python

- Любое число, не равное 0, или непустой объект - истина.
- Числа, равные 0, пустые объекты и значение None - ложь
- Операции сравнения применяются к структурам данных рекурсивно
- Операции сравнения возвращают True или False
- Логические операторы and и or возвращают истинный или ложный объект-операнд

Логические операторы:

```
X and Y
```

Истина, если оба значения X и Y истинны.

```
X or Y
```

Истина, если хотя бы одно из значений X или Y истинно.

```
not X
```

Истина, если X ложно.

1.3 Трехместное выражение if/else

Следующая инструкция:

```
if X:
    A = Y
else:
    A = Z
```

довольно короткая, но, тем не менее, занимает целых 4 строки. Специально для таких случаев и было придумано выражение if/else:

```
A = Y if X else Z
```

В данной инструкции интерпретатор выполнит выражение Y, если X истинно, в противном случае выполнится выражение Z.

```
>>> A = 't' if 'spam' else 'f'
```

```
>>> A
't'
```

2. Циклы for и while, операторы break и continue

2.1 Цикл while

While - один из самых универсальных циклов в Python, поэтому довольно медленный. Выполняет тело цикла до тех пор, пока условие цикла истинно.

```
>>> i = 5
>>> while i < 15:
...     print(i)
...     i = i + 2
...
5
7
9
11
13
```

2.2 Цикл for

Цикл for уже чуточку сложнее, чуть менее универсальный, но выполняется гораздо быстрее цикла while. Этот цикл проходится по любому итерируемому объекту (например строке или списку), и во время каждого прохода выполняет тело цикла.

```
>>> for i in 'hello world':
...     print(i * 2, end='')
...
hheellllloo  wwoorrlldd
```

2.3 Оператор continue

Оператор continue начинает следующий проход цикла, минуя оставшееся тело цикла (for или while)

```
>>> for i in 'hello world':
...     if i == 'o':
...         continue
```

```
...     print(i * 2, end='')
...
hheellll  wwrrlldd
```

2.4 Оператор break

Оператор break досрочно прерывает цикл.

```
>>>
```

```
>>> for i in 'hello world':
...     if i == 'o':
...         break
...     print(i * 2, end='')
...
hheellll
```

2.5 else

Слово else, примененное в цикле for или while, проверяет, был ли произведен выход из цикла инструкцией break, или же "естественным" образом. Блок инструкций внутри else выполнится только в том случае, если выход из цикла произошел без помощи break.

```
>>>
```

```
>>> for i in 'hello world':
...     if i == 'a':
...         break
... else:
...     print('Буквы a в строке нет')
...
Буквы a в строке нет
```

3. Словари (dict) и работа с ними. Методы словарей

3.1 Словари (dict)

Словари в Python - неупорядоченные коллекции произвольных объектов с доступом по ключу. Их иногда ещё называют ассоциативными массивами или хеш-таблицами.

Чтобы работать со словарём, его нужно создать. Сделать это можно несколькими способами. Во-первых, с помощью литерала:

```
>>> d = {}
>>> d
{}
>>> d = {'dict': 1, 'dictionary': 2}
>>> d
{'dict': 1, 'dictionary': 2}
```

Во-вторых, с помощью функции **dict**:

```
>>>
```

```
>>> d = dict(short='dict', long='dictionary')
>>> d
{'short': 'dict', 'long': 'dictionary'}
>>> d = dict([(1, 1), (2, 4)])
>>> d
{1: 1, 2: 4}
```

В-третьих, с помощью метода fromkeys:

```
>>> d = dict.fromkeys(['a', 'b'])
>>> d
{'a': None, 'b': None}
>>> d = dict.fromkeys(['a', 'b'], 100)
>>> d
{'a': 100, 'b': 100}
```

В-четвертых, с помощью генераторов словарей, которые очень похожи на генераторы списков.

```
>>> d = {a: a ** 2 for a in range(7)}
>>> d
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

Теперь попробуем добавить записей в словарь и извлечь значения ключей:

```
>>> d = {1: 2, 2: 4, 3: 9}
>>> d[1]
2
>>> d[4] = 4 ** 2
```

```
>>> d
{1: 2, 2: 4, 3: 9, 4: 16}
>>> d['1']
Traceback (most recent call last):
  File "", line 1, in
    d['1']
KeyError: '1'
```

Как видно из примера, присвоение по новому ключу расширяет словарь, присвоение по существующему ключу перезаписывает его, а попытка извлечения несуществующего ключа порождает исключение. Для избежания исключения есть специальный метод (см. ниже), или можно перехватывать исключение.

Что же можно еще делать со словарями? Да то же самое, что и с другими объектами: встроенные функции, ключевые слова (например, циклы for и while), а также специальные методы словарей.

3.2 Методы словарей

dict.clear() - очищает словарь.

dict.copy() - возвращает копию словаря.

classmethod **dict.fromkeys**(seq[, value]) - создает словарь с ключами из seq и значением value (по умолчанию None).

dict.get(key[, default]) - возвращает значение ключа, но если его нет, не бросает исключение, а возвращает default (по умолчанию None).

dict.items() - возвращает пары (ключ, значение).

dict.keys() - возвращает ключи в словаре.

dict.pop(key[, default]) - удаляет ключ и возвращает значение. Если ключа нет, возвращает default (по умолчанию бросает исключение).

dict.popitem() - удаляет и возвращает пару (ключ, значение). Если словарь пуст, бросает исключение KeyError. Помните, что словари неупорядочены.

dict.setdefault(key[, default]) - возвращает значение ключа, но если его нет, не бросает исключение, а создает ключ со значением default (по умолчанию None).

dict.update([other]) - обновляет словарь, добавляя пары (ключ, значение) из other. Существующие ключи перезаписываются. Возвращает None (не новый словарь!).

dict.values() - возвращает значения в словаре.

4. Списки (list). Функции и методы списков

4.1 Списки

Списки в Python - упорядоченные изменяемые коллекции объектов произвольных типов (почти как массив, но типы могут отличаться).

Чтобы использовать списки, их нужно создать. Создать список можно несколькими способами. Например, можно обработать любой итерируемый объект (например, строку) встроенной функцией **list**:

```
>>> list('список')
['с', 'п', 'и', 'с', 'о', 'к']
```

Список можно создать и при помощи литерала:

```
>>> s = []  # Пустой список
>>> l = ['s', 'p', ['isok'], 2]
>>> s
[]
>>> l
['s', 'p', ['isok'], 2]
```

Как видно из примера, список может содержать любое количество любых объектов (в том числе и вложенные списки), или не содержать ничего.

И еще один способ создать список - это **генераторы списков**. Генератор списков - способ построить новый список, применяя выражение к каждому элементу последовательности. Генераторы списков очень похожи на цикл for.

```
>>> c = [c * 3 for c in 'list']
>>> c
['lll', 'iii', 'sss', 'ttt']
```

Возможна и более сложная конструкция генератора списков:

```
>>> c = [c * 3 for c in 'list' if c != 'i']
>>> c
['lll', 'sss', 'ttt']
>>> c = [c + d for c in 'list' if c != 'i' for d in 'spam' if d != 'a']
>>> c
```

```
['ls', 'lp', 'lm', 'ss', 'sp', 'sm', 'ts', 'tp', 'tm']
```

Но в сложных случаях лучше пользоваться обычным циклом for для генерации списков.

3.2 Функции и методы списков

Создать создали, теперь нужно со списком что-то делать. Для списков доступны основные встроенные функции, а также методы списков.

Таблица "методы списков"

Метод	**Что делает**
list.append(x)	Добавляет элемент в конец списка
list.extend(L)	Расширяет список list, добавляя в конец все элементы списка L
list.insert(i, x)	Вставляет на i-ый элемент значение x
list.remove(x)	Удаляет первый элемент в списке, имеющий значение x. ValueError, если такого элемента не существует
list.pop([i])	Удаляет i-ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент
list.index(x, [start [, end]])	Возвращает положение первого элемента со значением x (при этом поиск ведется от start до end)
list.count(x)	Возвращает количество элементов со значением x
list.sort([key=функция])	Сортирует список на основе функции
list.reverse()	Разворачивает список
list.copy()	Поверхностная копия списка
list.clear()	Очищает список

Нужно отметить, что методы списков, в отличие от строковых методов, изменяют сам список, а потому результат выполнения не нужно записывать в эту переменную.

```
>>> l = [1, 2, 3, 5, 7]
```

```
>>> l.sort()
>>> l
[1, 2, 3, 5, 7]
>>> l = l.sort()
>>> print(l)
None
```

И, напоследок, примеры работы со списками:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

Изредка, для увеличения производительности, списки заменяют гораздо менее гибкими массивами (хотя в таких случаях обычно используют сторонние библиотеки, например NumPy).

5. Индексы и срезы

5.1 Взятие элемента по индексу

Как и в других языках программирования, взятие по индексу:

```
>>> a = [1, 3, 8, 7]
>>> a[0]
1
```

```
>>> a[3]
7
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Как и во многих других языках, нумерация элементов начинается с нуля. При попытке доступа к несуществующему индексу возникает исключение IndexError.

В данном примере переменная a являлась списком, однако взять элемент по индексу можно и у других типов: строк, кортежей.

В Python также поддерживаются отрицательные индексы, при этом нумерация идёт с конца, например:

```
>>> a = [1, 3, 8, 7]
>>> a[-1]
7
>>> a[-4]
1
>>> a[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

5.2 Срезы

В Python, кроме индексов, существуют ещё и **срезы**.

item[START:STOP:STEP] - берёт срез от номера START, до STOP (не включая его), с шагом STEP. По умолчанию START = 0, STOP = длине объекта, STEP = 1. Соответственно, какие-нибудь (а возможно, и все) параметры могут быть опущены.

```
>>> a = [1, 3, 8, 7]
>>> a[:]
[1, 3, 8, 7]
>>> a[1:]
[3, 8, 7]
>>> a[:3]
```

```
[1, 3, 8]
>>> a[::2]
[1, 8]
```

Также все эти параметры могут быть и отрицательными:

```
>>> a = [1, 3, 8, 7]
>>> a[::-1]
[7, 8, 3, 1]
>>> a[:-2]
[1, 3]
>>> a[-2::-1]
[8, 3, 1]
>>> a[1:4:-1]
[]
```

В последнем примере получился пустой список, так как START < STOP, а STEP отрицательный. То же самое произойдёт, если диапазон значений окажется за пределами объекта:

```
>>>
```

```
>>> a = [1, 3, 8, 7]
>>> a[10:20]
[]
```

Также с помощью срезов можно не только извлекать элементы, но и добавлять и удалять элементы (разумеется, только для изменяемых последовательностей).

```
>>> a = [1, 3, 8, 7]
>>> a[1:3] = [0, 0, 0]
>>> a
[1, 0, 0, 0, 7]
>>> del a[:-3]
>>> a
[0, 0, 7]
```

6. Кортежи (tuple)

Кортеж, по сути - неизменяемый список.

6.1 Зачем нужны кортежи, если есть списки?

- Защита от дурака. То есть кортеж защищен от изменений, как намеренных (что плохо), так и случайных (что хорошо).
- Меньший размер. Дабы не быть голословным:

```
>>> a = (1, 2, 3, 4, 5, 6)
>>> b = [1, 2, 3, 4, 5, 6]
>>> a.__sizeof__()
36
>>> b.__sizeof__()
44
```

- Возможность использовать кортежи в качестве ключей словаря:

```
>>> d = {(1, 1, 1) : 1}
>>> d
{(1, 1, 1): 1}
>>> d = {[1, 1, 1] : 1}
Traceback (most recent call last):
  File "", line 1, in
    d = {[1, 1, 1] : 1}
TypeError: unhashable type: 'list'
```

6.2 Как работать с кортежами?

С преимуществами кортежей разобрались, теперь встает вопрос - а как с ними работать. Примерно так же, как и со списками.

Создаем пустой кортеж:

```
>>> a = tuple() # С помощью встроенной функции tuple()
>>> a
()
>>> a = () # С помощью литерала кортежа
>>> a
()
```

```
>>>
```

Создаем кортеж из одного элемента:

```
>>>
```

```
>>> a = ('s')
>>> a
's'
```

Стоп. Получилась строка. Но как же так? Мы же кортеж хотели! Как же нам кортеж получить?

```
>>>
```

```
>>> a = ('s', )
>>> a
('s',)
```

Ура! Заработало! Все дело - в запятой. Сами по себе скобки ничего не значат, точнее, значат то, что внутри них находится одна инструкция, которая может быть отделена пробелами, переносом строк и прочим мусором. Кстати, кортеж можно создать и так:

```
>>>
```

```
>>> a = 's',
>>> a
('s',)
```

Но все же не увлекайтесь, и ставьте скобки, тем более, что бывают случаи, когда скобки необходимы.

Ну и создать кортеж из итерируемого объекта можно с помощью все той же пресловутой функции tuple()

```
>>>
```

```
>>> a = tuple('hello, world!')
>>> a
('h', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!')
```

6. 3 Операции с кортежами

Все операции над списками, не изменяющие список (сложение, умножение на число, методы index() и count() и некоторые другие операции). Можно также по-разному менять элементы местами и так далее.

Например, гордость программистов на python - поменять местами значения двух переменных:

```
a, b = b, a
```

7. Множества (set и frozenset)

7.1 Множество

Множество в python - "контейнер", содержащий не повторяющиеся элементы в случайном порядке.

Создаём множества:

```
>>> a = set()
>>> a
set()
>>> a = set('hello')
>>> a
{'h', 'o', 'l', 'e'}
>>> a = {'a', 'b', 'c', 'd'}
>>> a
{'b', 'c', 'a', 'd'}
>>> a = {i ** 2 for i in range(10)} # генератор множеств
>>> a
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> a = {}  # А так нельзя!
>>> type(a)
<class 'dict'>
```

Как видно из примера, множества имеет тот же литерал, что и словарь, но пустое множество с помощью литерала создать нельзя.

Множества удобно использовать для удаления повторяющихся элементов:

```
>>> words = ['hello', 'daddy', 'hello', 'mum']
>>> set(words)
{'hello', 'daddy', 'mum'}
```

С множествами можно выполнять множество операций: находить объединение, пересечение...

- len(s) - число элементов в множестве (размер множества).

- x in s - принадлежит ли x множеству s.
- **set.isdisjoint**(other) - истина, если set и other не имеют общих элементов.
- **set == other** - все элементы set принадлежат other, все элементы other принадлежат set.
- **set.issubset**(other) или **set <= other** - все элементы set принадлежат other.
- **set.issuperset**(other) или **set >= other** - аналогично.
- **set.union**(other, ...) или **set | other | ...** - объединение нескольких множеств.
- **set.intersection**(other, ...) или **set & other & ...** - пересечение.
- **set.difference**(other, ...) или **set - other - ...** - множество из всех элементов set, не принадлежащие ни одному из other.
- **set.symmetric_difference**(other); **set ^ other** - множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих.
- **set.copy**() - копия множества.

  И операции, непосредственно изменяющие множество:
- **set.update**(other, ...); set |= other | ... - объединение.
- **set.intersection_update**(other, ...); set &= other & ... - пересечение.
- **set.difference_update**(other, ...); set -= other | ... - вычитание.
- **set.symmetric_difference_update**(other); set ^= other - множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих.
- **set.add**(elem) - добавляет элемент в множество.
- **set.remove**(elem) - удаляет элемент из множества. KeyError, если такого элемента не существует.
- **set.discard**(elem) - удаляет элемент, если он находится в множестве.
- **set.pop**() - удаляет первый элемент из множества. Так как множества не упорядочены, нельзя точно сказать, какой элемент будет первым.
- **set.clear**() - очистка множества.

7.2 frozenset

Единственное отличие set от frozenset заключается в том, что set - изменяемый тип данных, а frozenset - нет. Примерно похожая ситуация с списками и кортежами.

```
>>> a = set('qwerty')
>>> b = frozenset('qwerty')
>>> a == b
True
>>> True
```

```
True
>>> type(a - b)
<class 'set'>
>>> type(a | b)
<class 'set'>
>>> a.add(1)
>>> b.add(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

8.Функции в Python

8.1 Определение

Вот пример простой функции:

```
def compute_surface(radius):
    from math import pi
    return pi * radius * radius
```

Для определения функции нужно всего лишь написать ключевое слово def перед ее именем, а после — поставить двоеточие. Следом идет блок инструкций. Последняя строка в блоке инструкций может начинаться с return, если нужно вернуть какое-то значение. Если инструкции return нет, тогда по умолчанию функция будет возвращать объект None. Как в этом примере:

```
i = 0
def increment():
    global i
    i += 1
```

Функция инкрементирует глобальную переменную i и возвращает None (по умолчанию).

8.2 Вызовы

Для вызова функции, которая возвращает переменную, нужно ввести:

```
surface = compute_surface(1.)
```

Для вызова функции, которая ничего не возвращает:

```
increment()
```

Функцию можно записать в одну строку, если блок инструкций представляет собой простое выражение:

```
def sum(a, b): return a + b
```

Функции могут быть вложенными:

```
def func1(a, b):
    def inner_func(x):
        return x*x*x
    return inner_func(a) + inner_func(b)
```

Функции — это объекты, поэтому их можно присваивать переменным.

8.3 Инструкция return

Возврат простого значения

Аргументы можно использовать для изменения ввода и таким образом получать вывод функции. Но куда удобнее использовать инструкцию return, примеры которой уже встречались ранее. Если ее не написать, функция вернет значение None.

Возврат нескольких значений

Пока что функция возвращала только одно значение или не возвращала ничего (объект None). А как насчет нескольких значений? Этого можно добиться с помощью массива. Технически, это все еще один объект. Например:

```
def stats(data):
    """данные должны быть списком"""
    _sum = sum(data) # обратите внимание на подчеркивание, чтобы
избежать переименования встроенной функции sum
    mean = _sum / float(len(data)) # обратите внимание на
использование функции float, чтобы избежать деления на целое число
    variance = sum([(x-mean)**2/len(data) for x in data])
    return mean,variance   # возвращаем x,y — кортеж!

m, v = stats([1, 2, 1])
```

8.4 Аргументы и параметры

В функции можно использовать неограниченное количество параметров, но число аргументов должно точно соответствовать параметрам. Эти параметры представляют собой позиционные аргументы. Также Python предоставляет возможность определять значения по умолчанию, которые можно задавать с помощью аргументов-ключевых слов.

Параметр — это имя в списке параметров в первой строке определения функции. Он получает свое значение при вызове. Аргумент — это реальное значение или ссылка на него, переданное функции при вызове. В этой функции:

```
def sum(x, y):
    return x + y
```

x и y — это параметры, а в этой:

```
sum(1, 2)
```

1 и 2 — аргументы.

При определении функции параметры со значениями по умолчанию нужно указывать до позиционных аргументов:

```
def compute_surface(radius, pi=3.14159):
    return pi * radius * radius
```

Если использовать необязательный параметр, тогда все, что указаны справа, должны быть параметрами по умолчанию.

Выходит, что в следующем примере допущена ошибка:

```
def compute_surface(radius=1, pi):
    return pi * radius * radius
```

Для вызовов это работает похожим образом. Сначала нужно указывать все позиционные аргументы, а только потом необязательные:

```
S = compute_surface(10, pi=3.14)
```

На самом деле, следующий вызов корректен (можно конкретно указывать имя позиционного аргумента), но этот способ не пользуется популярностью:

```
S = compute_surface(radius=10, pi=3.14)
```

А этот вызов **некорректен:**

```
S = compute_surface(pi=3.14, 10)
```

При вызове функции с аргументами по умолчанию можно указать один или несколько, и порядок не будет иметь значения:

```
def compute_surface2(radius=1, pi=3.14159):
    return pi * radius * radius
S = compute_surface2(radius=1, pi=3.14)
S = compute_surface2(pi=3.14, radius=10.)
S = compute_surface2(radius=10.)
```

Можно не указывать ключевые слова, но тогда порядок имеет значение. Он должен соответствовать порядку параметров в определении:

```
S = compute_surface2(10., 3.14)
S = compute_surface2(10.)
```

Если ключевые слова не используются, тогда нужно указывать все аргументы:

```
def f(a=1,b=2, c=3):
    return a + b + c
```

Второй аргумент можно пропустить:

```
f(1,,3)
```

Чтобы обойти эту проблему, можно использовать словарь:

```
params = {'a':10, 'b':20}
S = f(**params)
```

Значение по умолчанию оценивается и сохраняется только один раз при определении функции (не при вызове). Следовательно, если значение по умолчанию — это изменяемый объект, например, список или словарь, он будет меняться каждый раз при вызове функции. Чтобы избежать такого поведения, инициализацию нужно проводить внутри функции или использовать неизменяемый объект:

```
def inplace(x, mutable=[]):
   mutable.append(x)
   return mutable
res = inplace(1)
```

```
res = inplace(2)
print(inplace(3))
[1, 2, 3]
def inplace(x, lst=None):
   if lst is None: lst=[]
   lst.append()
   return lst
```

Еще один пример изменяемого объекта, значение которого поменялось при вызове:

```
def change_list(seq):
    seq[0] = 100
original = [0, 1, 2]
change_list(original)
original
[100, 1, 2]
```

Дабы не допустить изменения оригинальной последовательности, нужно передать копию изменяемого объекта:

```
original = [0, 1, 2]
change_list(original[:])
original
[0, 1, 2]
```

Указание произвольного количества аргументов

8.5 Позиционные аргументы

Иногда количество позиционных аргументов может быть переменным. Примерами таких функций могут быть max() и min(). Синтаксис для определения таких функций следующий:

```
def func(pos_params, *args):
    block statememt
```

При вызове функции нужно вводить команду следующим образом:

```
func(pos_params, arg1, arg2, ...)
```

Python обрабатывает позиционные аргументы следующим образом: подставляет обычные позиционные аргументы слева направо, а затем помещает остальные

позиционные аргументы в кортеж (*args), который можно использовать в функции.

Вот так:

```
def add_mean(x, *data):
    return x + sum(data)/float(len(data))
add_mean(10,0,1,2,-1,0,-1,1,2)
10.5
```

Если лишние аргументы не указаны, значением по умолчанию будет пустой кортеж.

8.6 Произвольное количество аргументов-ключевых слов

Как и в случае с позиционными аргументами можно определять произвольное количество аргументов-ключевых слов следующим образом (в сочетании с произвольным числом необязательных аргументов из прошлого раздела):

```
def func(pos_params, *args, **kwargs):
    block statememt
```

При вызове функции нужно писать так:

```
func(pos_params, kw1=arg1, kw2=arg2, ...)
```

Python обрабатывает аргументы-ключевые слова следующим образом: подставляет обычные позиционные аргументы слева направо, а затем помещает другие позиционные аргументы в кортеж (*args), который можно использовать в функции (см. предыдущий раздел). В конце концов, он добавляет все лишние аргументы в словарь (**kwargs), который сможет использовать функция.

Есть функция:

```
def print_mean_sequences(**kwargs):
    def mean(data):
        return sum(data)/float(len(data))
    for k, v in kwargs.items():
        print k, mean(v)
print_mean_sequences(x=[1,2,3], y=[3,3,0])
y 2.0
```

```
x 2.0
```

Важно, что пользователь также может использовать словарь, но перед ним нужно ставить две звездочки (**):

```
print_mean_sequences(**{'x':[1,2,3], 'y':[3,3,0]})
y 2.0
x 2.0
```

Порядок вывода также не определен, потому что словарь не отсортирован.

8.7 Документирование функции

Определим функцию:

```
def sum(s,y): return x + y
```

Если изучить ее, обнаружатся два скрытых метода (которые начинаются с двух знаков нижнего подчеркивания), среди которых есть __doc__. Он нужен для настройки документации функции. Документация в Python называется docstring и может быть объединена с функцией следующим образом:

```
def sum(x, y):
    """Первая срока - заголовок
    Затем следует необязательная пустая строка и текст
    документации.
    """
    return x+y
```

Команда docstring должна быть первой инструкцией после объявления функции. Ее потом можно будет извлекать или дополнять:

```
print(sum.__doc__)
sum.__doc__ += "some additional text"
```

8.8 Методы, функции и атрибуты, связанные с объектами функции

Если поискать доступные для функции атрибуты, то в списке окажутся следующие методы (в Python все является объектом — даже функция):

```
sum.func_closure   sum.func_defaults  sum.func_doc       sum.func_name
```

```
sum.func_code       sum.func_dict       sum.func_globals
```

И несколько скрытых методов, функций и атрибутов. Например, можно получить имя функции или модуля, в котором она определена:

```
>>> sum.__name__
"sum"
>>> sum.__module
"__main__"
```

Есть и другие. Вот те, которые не обсуждались:

```
sum.__call__          sum.__delattr__       sum.__getattribute__
sum.__setattr__

sum.__class__         sum.__dict__          sum.__globals__       sum.__new__
sum.__sizeof__

sum.__closure__       sum.__hash__          sum.__reduce__        sum.__str__

sum.__code__          sum.__format__        sum.__init__
sum.__reduce_ex__     sum.__subclasshook__

sum.__defaults__      sum.__get__           sum.__repr__
```

8.9 Рекурсивные функции

Рекурсия — это не особенность Python. Это общепринятая и часто используемая техника в Computer Science, когда функция вызывает сама себя. Самый известный пример — вычисление факториала n! = n * n — 1 * n -2 * … 2 *1. Зная, что 0! = 1, факториал можно записать следующим образом:

```
def factorial(n):
    if n != 0:
        return n * factorial(n-1)
    else:
        return 1
```

Другой распространенный пример — определение последовательности Фибоначчи:

```
f(0) = 1
f(1) = 1
f(n) = f(n-1) + f(n-2)
```

Рекурсивную функцию можно записать так:

```
def fibbonacci(n):
    if n >= 2:
        else:
    return 1
```

Важно, чтобы в ней было была конечная инструкция, иначе она никогда не закончится. Реализация вычисления факториала выше, например, не является надежной. Если указать отрицательное значение, функция будет вызывать себя бесконечно. Нужно написать так:

```
def factorial(n):
    assert n > 0
    if n != 0:
        return n * factorial(n-1)
    else:
        return 1
```

Рекурсия позволяет писать простые и элегантные функции, но это не гарантирует эффективность и высокую скорость исполнения.

Если рекурсия содержит баги (например, длится бесконечно), функции может не хватить памяти. Задать максимальное значение рекурсий можно с помощью модуля sys.

8.10 Глобальная переменная

Вот уже знакомый пример с глобальной переменной:

```
i = 0
def increment():
    global i
    i += 1
```

Здесь функция увеличивает на 1 значение глобальной переменной i. Это способ изменять глобальную переменную, определенную вне функции. Без него функция не будет знать, что такое переменная i. Ключевое слово global можно вводить в любом месте, но переменную разрешается использовать только после ее объявления.

За редкими исключениями глобальные переменные лучше вообще не использовать.

8.11 Присвоение функции переменной

С существующей функцией func синтаксис максимально простой:

```
variable = func
```

Переменным также можно присваивать встроенные функции. Таким образом позже есть возможность вызывать функцию другим именем. Такой подход называется непрямым вызовом функции.

Менять название переменной также разрешается:

```
def func(x): return x
a1 = func
a1(10)
10
a2 = a1
a2()
10
```

В этом примере a1, a2 и func имеют один и тот же id. Они ссылаются на один объект.

Практический пример — рефакторинг существующего кода. Например, есть функция sq, которая вычисляет квадрат значения:

```
def sq(x): return x*x
```

Позже ее можно переименовать, используя более осмысленное имя. Первый вариант — просто сменить имя. Проблема в том, что если в другом месте кода используется sq, то этот участок не будет работать. Лучше просто добавить следующее выражение:

```
square = sq
```

Последний пример. Предположим, встроенная функция была переназначена:

```
dir = 3
```

Теперь к ней нельзя получить доступ, а это может стать проблемой. Чтобы вернуть ее обратно, нужно просто удалить переменную:

```
del dir
dir()
```

8.12 Анонимная функция: лямбда

Лямбда-функция — это короткая однострочная функция, которой даже не нужно имя давать. Такие выражения содержат лишь одну инструкцию, поэтому, например, if, for и while использовать нельзя. Их также можно присваивать переменным:

```
product = lambda x,y: x*y
```

В отличие от функций, здесь не используется ключевое слово return. Результат работы и так возвращается.

С помощью type() можно проверить тип:

```
>>> type(product)
function
```

На практике эти функции редко используются. Это всего лишь элегантный способ записи, когда она содержит одну инструкцию.

```
power = lambda x=1, y=2: x**y
square = power
square(5.)
25
power = lambda x,y,pow=2: x**pow + y
[power(x,2, 3) for x in [0,1,2]]
[2, 3, 10]
```

8.13 Изменяемые аргументы по умолчанию

```
>>> def foo(x=[]):
...     x.append(1)
...     print x
...
>>> foo()
[1]
```

```
>>> foo()
[1, 1]
>>> foo()
[1, 1, 1]
```

Вместо этого нужно использовать значение «не указано» и заменить на изменяемый объект по умолчанию:

```
>>> def foo(x=None):
...     if x is None:
...         x = []
...     x.append(1)
...     print x
>>> foo()
[1]
>>> foo()
[1]
```