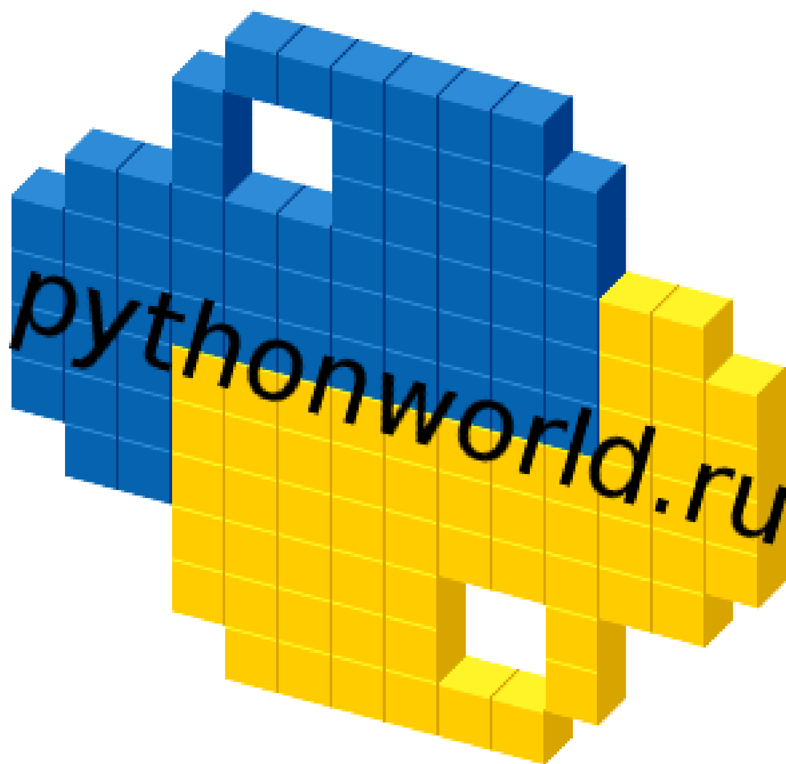


Самоучитель Python

Выпуск 0.2

Дмитрий Мусин

мая 07, 2017

Оглавление

1 Возможности языка python **1**

2 Скачать Python **2**
2.1 Установка Python на Windows 2
2.2 Установка Python на linux системы (ubuntu, linux mint и другие) 8

3 Первая программа. Среда разработки IDLE **10**

4 Синтаксис языка Python **13**
4.1 Синтаксис 13
4.2 Несколько специальных случаев 13

5 Программа не работает. Что делать? **15**

6 Инструкция if-elif-else, проверка истинности, трехместное выражение if/else **20**
6.1 Синтаксис инструкции if 20
6.2 Проверка истинности в Python 21
6.3 Трехместное выражение if/else 21

7 Циклы for и while, операторы break и continue, волшебное слово else **23**
7.1 Цикл while 23
7.2 Цикл for 23
7.3 Оператор continue 24
7.4 Оператор break 24
7.5 Волшебное слово else 24

8 Ключевые слова, модуль keyword **25**
8.1 Ключевые слова 25
8.2 Модуль keyword 26

9 Встроенные функции **27**
9.1 Встроенные функции, выполняющие преобразование типов 27
9.2 Другие встроенные функции 28

10 Числа: целые, вещественные, комплексные 31
10.1 Целые числа (int) . . . 31
10.2 Вещественные числа (float) . . . 34
10.3 Комплексные числа (complex) . . . 35

11 Работа со строками в Python: литералы 37
11.1 Литералы строк . . . 37

12 Строки. Функции и методы строк 40
12.1 Базовые операции . . . 40
12.2 Другие функции и методы строк . . . 41
12.3 Таблица “Функции и методы строк” . . . 42

13 Форматирование строк. Метод format 45
13.1 Форматирование строк с помощью метода format . . . 45

14 Списки (list). Функции и методы списков 48
14.1 Что такое списки? . . . 48
14.2 Функции и методы списков . . . 49
14.3 Таблица “методы списков” . . . 49

15 Индексы и срезы 51
15.1 Взятие элемента по индексу . . . 51
15.2 Срезы . . . 52

16 Кортежи (tuple) 54
16.1 Зачем нужны кортежи, если есть списки? . . . 54
16.2 Как работать с кортежами? . . . 55
16.3 Операции с кортежами . . . 56

17 Словари (dict) и работа с ними. Методы словарей 57
17.1 Методы словарей . . . 58

18 Множества (set и frozenset) 60
18.1 Что такое множество? . . . 60
18.2 frozenset . . . 62

19 Функции и их аргументы 63
19.1 Именные функции, инструкция def . . . 63
19.2 Аргументы функции . . . 64
19.3 Анонимные функции, инструкция lambda . . . 65

20 Исключения в python. Конструкция try - except для обработки исключений 66

21 Байты (bytes и bytearray) 71
21.1 Bytearray . . . 72

22 None (null), или немного о типе NoneType 73
22.1 Эквивалент null в Python: None . . . 73
22.2 Проверка на None . . . 74

23 Файлы. Работа с файлами. **76**
23.1 Чтение из файла . . . 77
23.2 Запись в файл . . . 77

24 With ... as - менеджеры контекста **79**

25 PEP 8 - руководство по написанию кода на Python **81**
25.1 Содержание . . . 81
25.2 Внешний вид кода . . . 83
25.3 Пробелы в выражениях и инструкциях . . . 87
25.4 Комментарии . . . 90
25.5 Контроль версий . . . 92
25.6 Соглашения по именованию . . . 92
25.7 Общие рекомендации . . . 97

26 Документирование кода в Python. PEP 257 **102**
26.1 Что такое строки документации? . . . 102
26.2 Однострочные строки документации . . . 103
26.3 Многострочные строки документации . . . 103

27 Работа с модулями: создание, подключение инструкциями import и from **105**
27.1 Подключение модуля из стандартной библиотеки . . . 105
27.2 Использование псевдонимов . . . 106
27.3 Инструкция from . . . 106
27.4 Создание своего модуля на Python . . . 107

28 Объектно-ориентированное программирование. Общее представление **110**

29 Инкапсуляция, наследование, полиморфизм **112**
29.1 Инкапсуляция . . . 112
29.2 Наследование . . . 113
29.3 Полиморфизм . . . 113

30 Перегрузка операторов **115**
30.1 Перегрузка арифметических операторов . . . 117

31 Декораторы **121**
31.1 Передача декоратором аргументов в функцию . . . 123
31.2 Декорирование методов . . . 124
31.3 Декораторы с аргументами . . . 126
31.4 Некоторые особенности работы с декораторами . . . 128
31.5 Примеры использования декораторов . . . 129

32 Устанавливаем python-пакеты с помощью pip **131**
32.1 Установка pip . . . 131
32.2 Начало работы . . . 132
32.3 Что ещё умеет делать pip . . . 132

33 Часто задаваемые вопросы **133**

33.1 Почему я получаю исключение UnboundLocalError, хотя переменная имеет значение? . . . 133
33.2 Каковы правила для глобальных и локальных переменных в Python? . . . 134
33.3 Почему анонимные функции (lambda), определенные в цикле с разными значениями, возвращают один и тот же результат? . . . 135
33.4 Как организовать совместный доступ к глобальным переменным для нескольких модулей? . . . 136
33.5 Как правильнее использовать импортирование? . . . 136
33.6 Почему значения по умолчанию разделяются между объектами? . . . 137
33.7 Как передать опциональные или именованные параметры из одной функции в другую? . . . 138
33.8 Почему изменение списка 'y' изменяет также список 'x'? . . . 139
33.9 Как создавать функции более высокого порядка? . . . 140
33.10 Как скопировать объект в Python? . . . 141
33.11 Как узнать доступные методы и атрибуты объекта? . . . 141
33.12 Как можно узнать имя объекта? . . . 141
33.13 Какой приоритет у оператора "запятая"? . . . 142
33.14 Есть ли в Python эквивалент тернарного оператора "?:" в C? . . . 142
33.15 Можно ли писать обфусцированные однострочники? . . . 142
33.16 Почему -22 // 10 равно -3? . . . 143
33.17 Как можно изменить строку? . . . 143
33.18 Как использовать строки для вызова функций/методов? . . . 144
33.19 Как удалить все символы новой строки в конце строки? . . . 145
33.20 Как удалить повторяющиеся элементы в списке? . . . 145
33.21 Как создать многомерный список? . . . 145
33.22 Почему a_tuple[i] += ['item'] не работает, а добавление работает? . . . 146

34 Задачи по Python **148**
34.1 Простейшие арифметические операции (1) . . . 148
34.2 Високосный год (2) . . . 148
34.3 Квадрат (3) . . . 149
34.4 Времена года (4) . . . 149
34.5 Банковский вклад (5) . . . 149
34.6 Простые числа (6) . . . 149
34.7 Правильная дата (7) . . . 149
34.8 XOR-шифрование (8) . . . 149

Глава 1

Возможности языка python

Так как мне часто стали задавать вопросы о том, чем может быть полезен Python, я решил написать небольшую обзорную статью на эту тему.

Вот лишь некоторые вещи, которые умеет делать python:

- Работа с xml/html файлами
- Работа с http запросами
- GUI (графический интерфейс)
- Создание веб-сценариев
- Работа с FTP
- Работа с изображениями, аудио и видео файлами
- Робототехника
- Программирование математических и научных вычислений

И многое, многое другое…

Таким образом, python подходит для решения львиной доли повседневных задач, будь то резервное копирование, чтение электронной почты, либо же какая-нибудь игрушка. Язык программирования Python практически ничем не ограничен, поэтому также может использоваться в крупных проектах. К примеру, python интенсивно применяется IT-гигантами, такими как, например, Google и Yandex. К тому же простота и универсальность python делают его одним из лучших языков программирования.

Глава 2

Скачать Python

Сегодня мы поговорим о том, как скачать и установить python 3 на свой компьютер. Бесплатно, без регистрации и SMS :)

Установка Python на Windows

Скачивать python будем с официального сайта. Кстати, не рекомендую скачивать интерпретатор python с других сайтов или через торрент, в них могут быть вирусы. Программа бесплатная. Заходим на https://python.org/downloads/windows/, выбираем "latest python release" и **python 3**.

На python 2 могут не работать некоторые мои примеры программ.

На момент написания статьи это python 3.4.1.

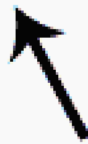
Появляется страница с описанием данной версии Python (на английском). Если интересно - можете почитать. Затем крутим в самый низ страницы, а затем открываем "download page".

More resources

- Change log for this release.
- Online Documentation
- What's new in 3.4?
- 3.4 Release Schedule
- Report bugs at http://bugs.python.org.
- Help fund Python and its community.

Download

Please proceed to the download page for the download.



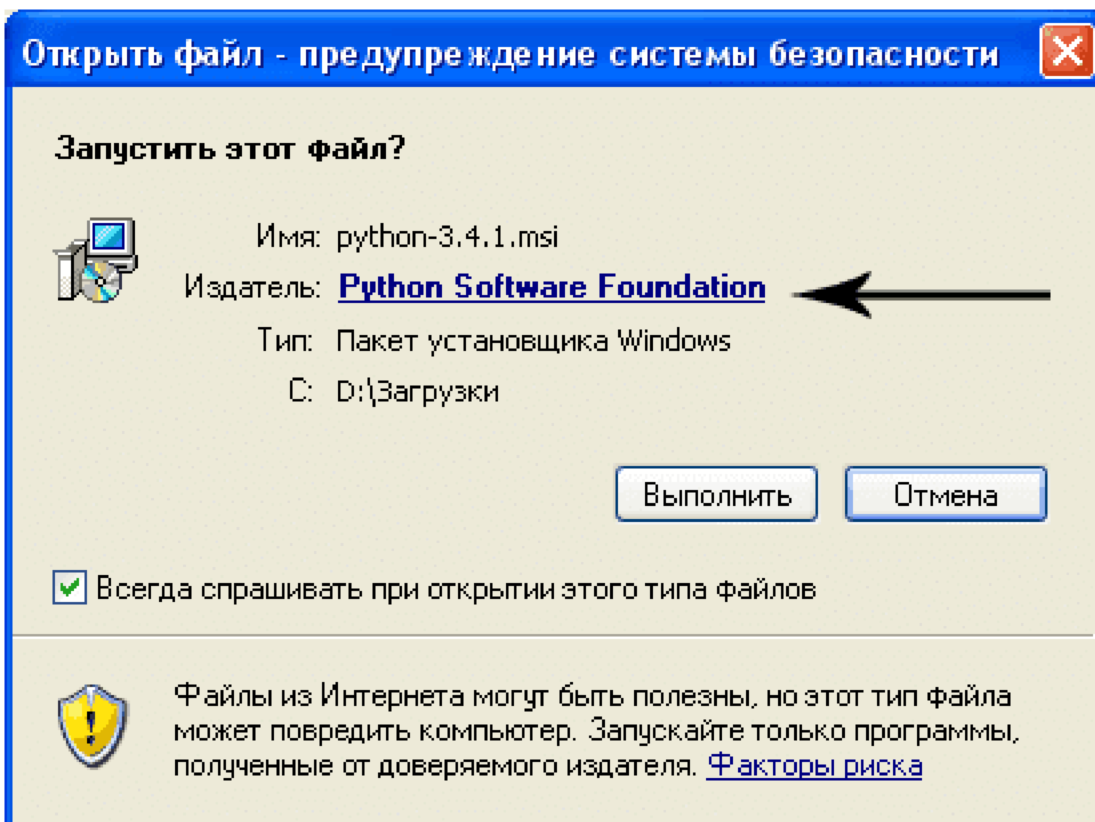
Notes on this release:

- The binaries for AMD64 will also work on processors that implement the Intel 64 architecture. (Also known as the "x64" architecture, and formerly known as both "EM64T" and "x86-64".) They will not work on Intel Itanium Processors (formerly "IA-64").
- There is important information about IDLE, Tkinter, and Tcl/Tk on Mac OS X here.

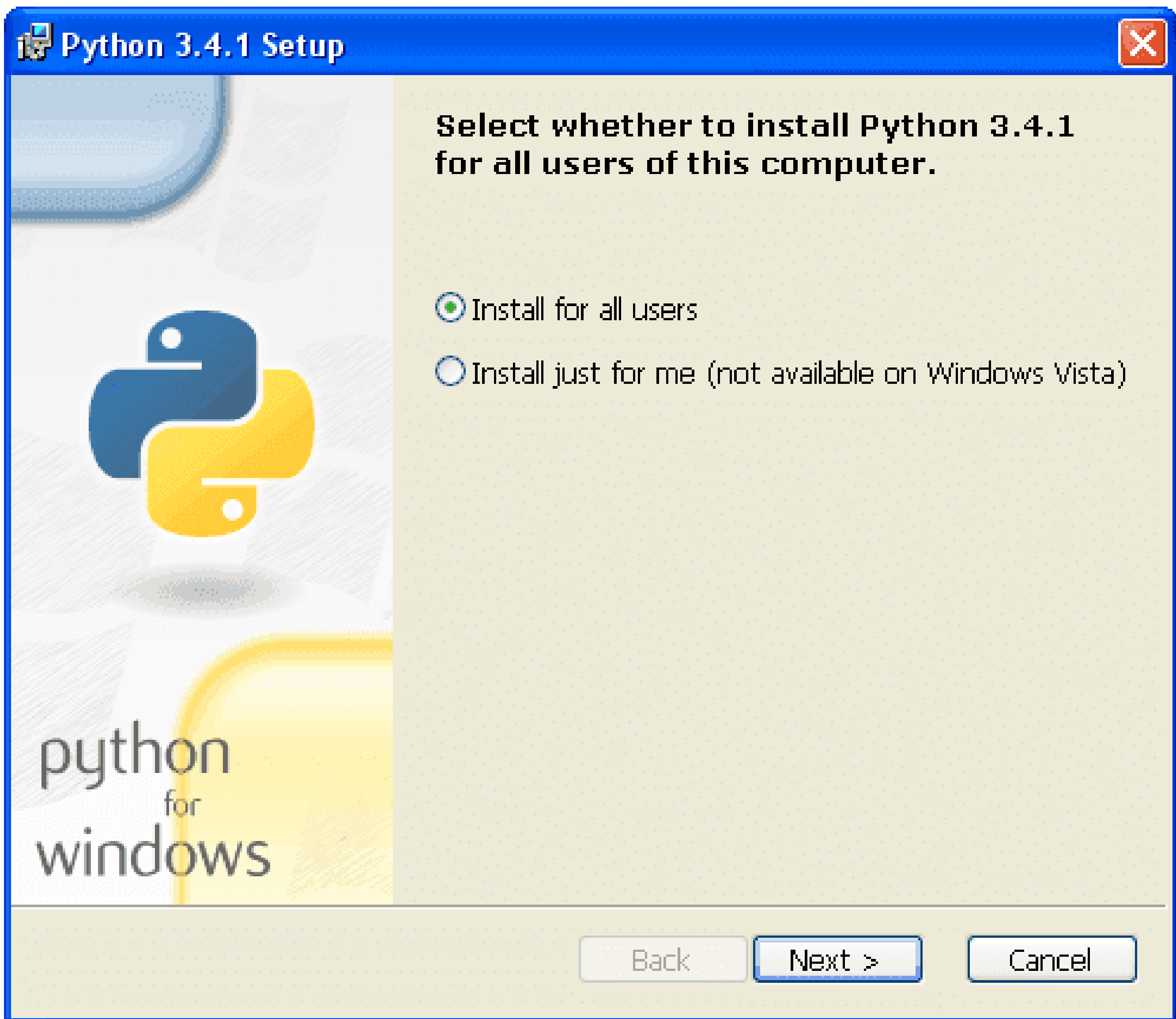
Вы увидите список файлов, которые можно загрузить. Нам нужен Windows x86 MSI installer (если система 32-х битная), или Windows x86-64 MSI installer (если система 64-х битная). Больше из файлов нам ничего не нужно.

Version	Operating System	Description	Date	MD5 Sum	File Size
Mac OS X 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later		316a2f83edff73bbbcb2c84390bee2db	22776248
Mac OS X 32-bit i386/PPC installer	Mac OS X	for Mac OS X 10.5 and later		534f8ec2f5ad5539f9165b3125b5e959	22692757
XZ compressed source tarball	Source release			6cafc183b4106476dd73d5738d7f616a	14125788
Gzipped source tarball	Source release			26695450087f8587b26d0b6a63844af5	19113124
Windows debug information files	Windows			9ce29e8356cf13f88e41f7595c2d7399	36744364
Windows x86 MSI installer	Windows			4940c3fad01ffa2ca7f9cc43a005b89a	24408064
Windows debug information files for 64-bit binaries	Windows			44a2d4d3c62a147f5a9f733b030490d1	24129218
Windows help file	Windows			6ff47ff938b15d2900f3c7311ab629e5	7297786
Windows x86-64 MSI installer	Windows	for AMD64/EM64T/x64, not Itanium processors		25440653f27ee1597fd6b3e15eee155f	25104384

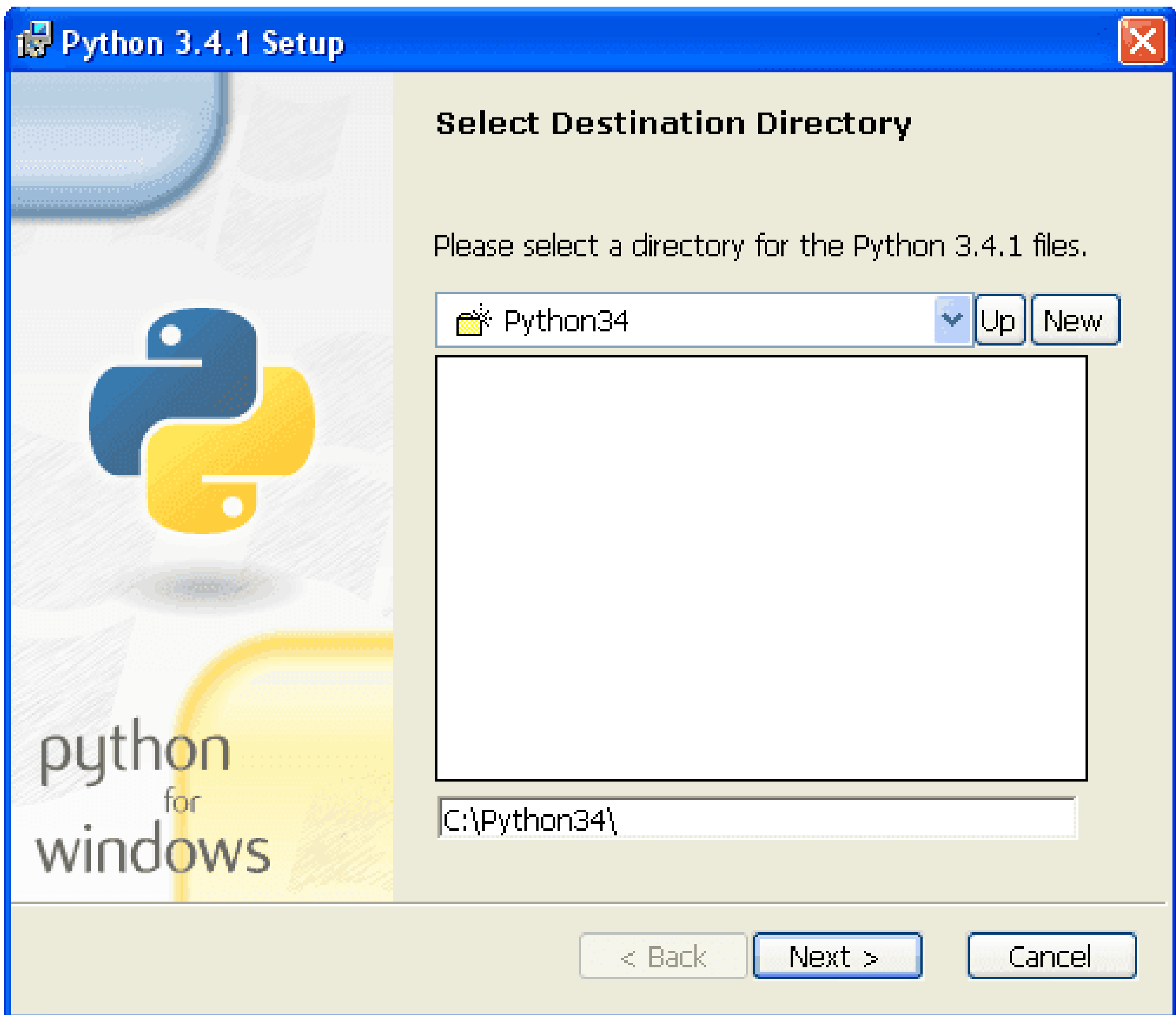
Ждём, пока python загрузится. Затем открываем загрузившийся файл. Файл подписан *Python Software Foundation*, значит, все в порядке. Пользуясь случаем, напоминаю, что не стоит открывать незнакомые exe файлы.



Устанавливаем для всех пользователей или только для одного (на ваше усмотрение).



Выбираем папку для установки. Я оставляю папку по умолчанию. Вы можете выбрать любую папку на своем диске.



Выбираем компоненты, которые будут установлены. Оставьте компоненты по умолчанию, если не уверены.

Ждем установки python…

Finish. Поздравляю, вы установили Python! Также в установщик python для windows встроена среда разработки IDLE. Прямо сейчас вы можете написать свою первую программу!

Установка Python на linux системы (ubuntu, linux mint и другие)

Откройте консоль (обычно ctrl+alt+t). Введите в консоли:

```
python3
```

Скорее всего, вас любезно поприветствует python 3:

```
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Если это так, то можно вас поздравить: у вас уже стоит python 3. В противном случае нужно установить пакет ***python3***:

```
sudo apt-get install python3
```

Либо через mintinstaller / synaptic / центр приложений ubuntu / что вам больше нравится.

В python для linux нет предустановленной среды IDLE. Если хотите, её можно установить отдельно. Пакет называется ***idle3*** (в более ранних версиях он может называться *python3-idle*).

Однако, её установка не является обязательной. Вы можете писать в своём любимом текстовом редакторе (gedit, vim, emacs…) и запускать программы через консоль:

```
python3 path_to_file.py
```

Теперь вы можете написать первую программу (хотите, пишите в IDLE, хотите - в своём любимом текстовом редакторе).

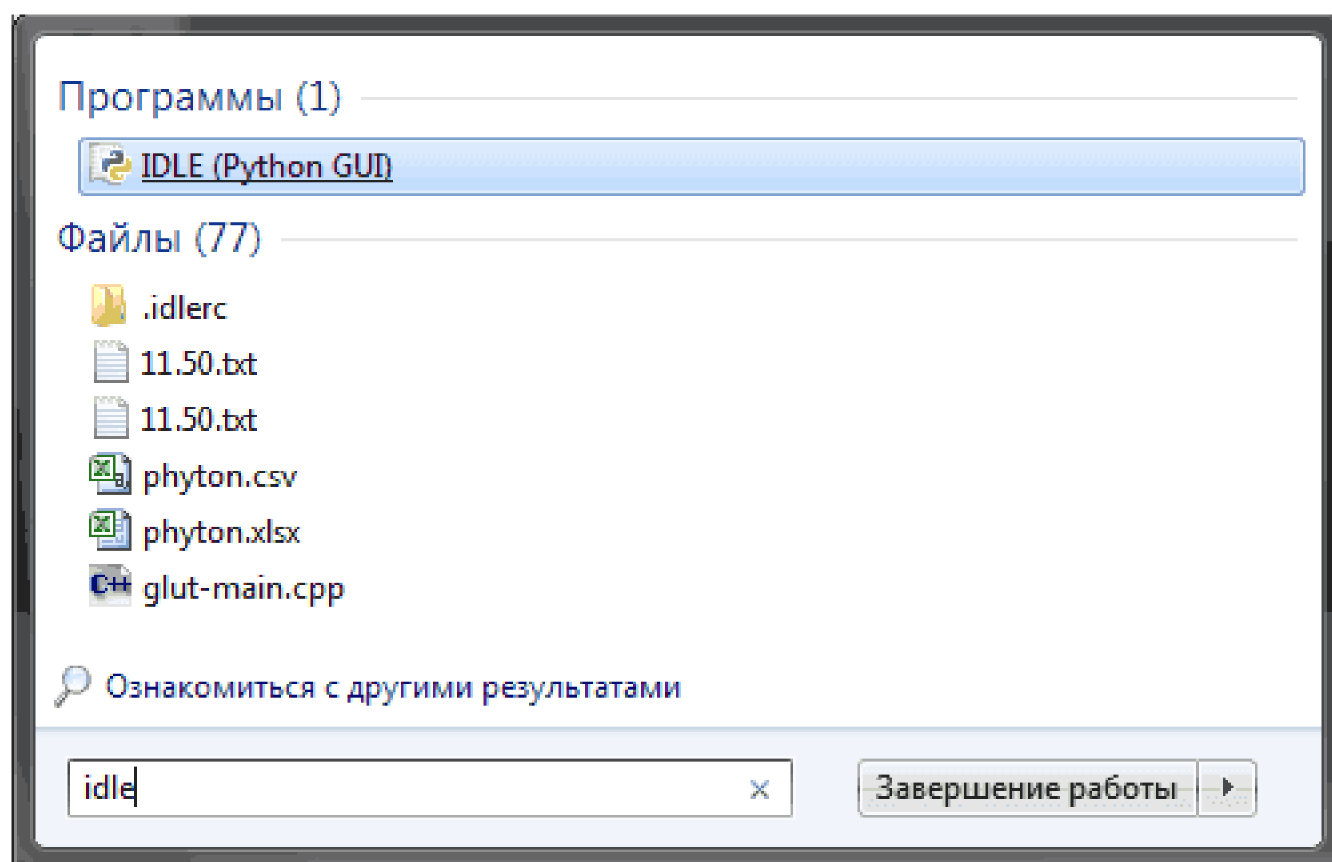
Глава 3

Первая программа. Среда разработки IDLE

Сегодня мы напишем свою первую программу в среде разработки IDLE.

После загрузки и установки python открываем IDLE (среда разработки на языке Python, поставляемая вместе с дистрибутивом).

Здесь и далее буду приводить примеры под ОС Windows, так как именно она у меня сейчас под рукой.

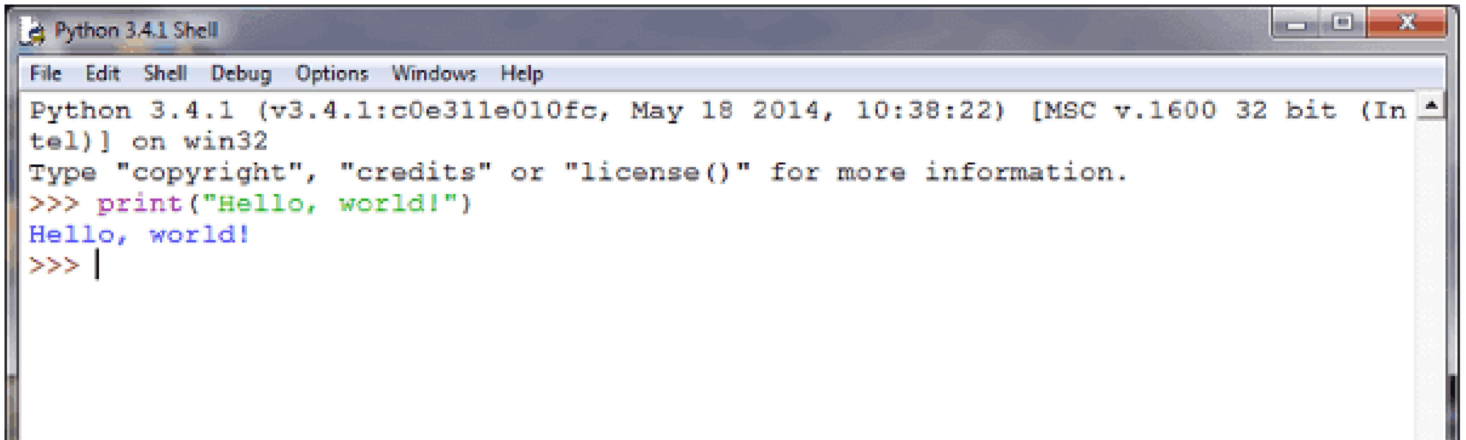


Запускаем IDLE (изначально запускается в интерактивном режиме), после чего уже можно начинать писать первую программу. Традиционно, первой программой у нас будет "hello world".

Чтобы написать "hello world" на python, достаточно всего одной строки:

```
print("Hello world!")
```

Вводим этот код в IDLE и нажимаем Enter. Результат виден на картинке:



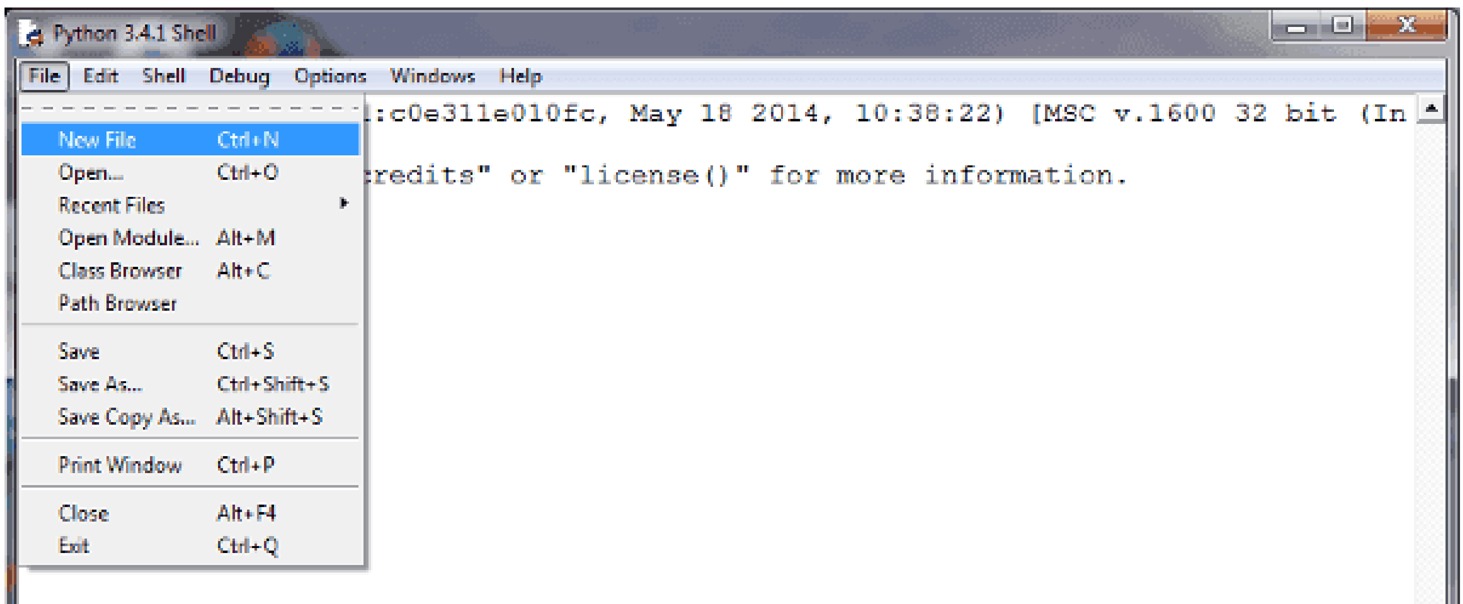
Поздравляю! Вы написали свою **первую программу на python**! (если что-то не работает).

С интерактивным режимом мы немного познакомились, можете с ним ещё поиграться, например, написать

```
print(3 + 4)
print(3 * 5)
print(3 ** 2)
```

Но, всё-таки, интерактивный режим не будет являться основным. В основном, вы будете сохранять программный код в файл и запускать уже файл.

Для того, чтобы создать новое окно, в интерактивном режиме IDLE выберите File → New File (или нажмите Ctrl + N).



В открывшемся окне введите следующий код:

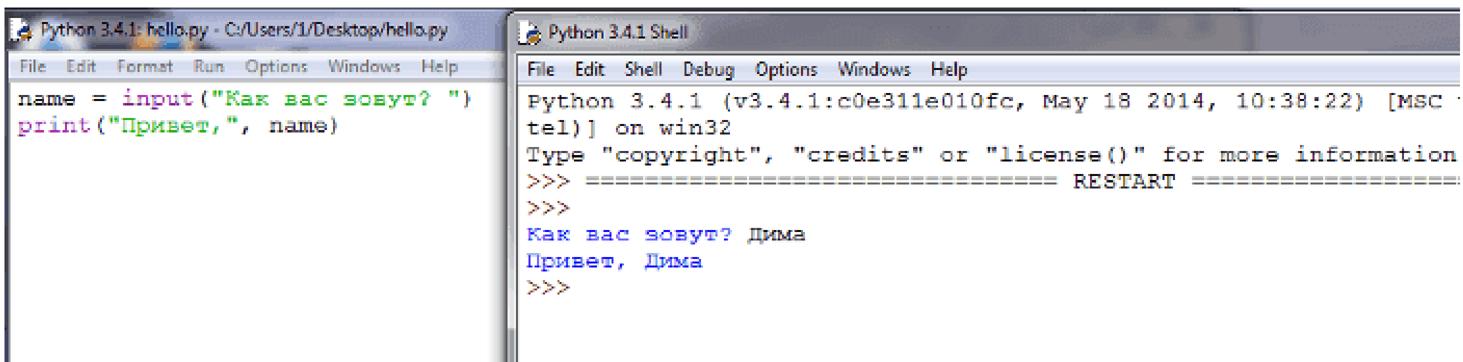
```
name = input("Как Вас зовут? ")
print("Привет,", name)
```

Первая строка печатает вопрос ("Как Вас зовут? "), ожидает, пока вы не напечатаете что-нибудь и не нажмёте Enter и сохраняет введённое значение в переменной name.

Во второй строке мы используем функцию print для вывода текста на экран, в данном случае для вывода "Привет, " и того, что хранится в переменной "name".

Теперь нажмём F5 (или выберем в меню IDLE Run → Run Module) и убедимся, что то, что мы написали, работает. Перед запуском IDLE предложит нам сохранить файл. Сохраним туда, куда вам будет удобно, после чего программа запустится.

Вы должны увидеть что-то наподобие этого (на скриншоте слева - файл с написанной вами программой, справа - результат её работы):



Поздравляю! Вы научились писать простейшие программы, а также познакомились со средой разработки IDLE. Теперь можно немного отдохнуть, а потом начать изучать python дальше. Можете посмотреть синтаксис python, циклы или условия. Желаю удачи!

Глава 4

Синтаксис языка Python

Синтаксис языка Python, как и сам язык, очень прост.

Синтаксис

- Конец строки является концом инструкции (точка с запятой не требуется).
- Вложенные инструкции объединяются в блоки по величине отступов. Отступ может быть любым, главное, чтобы в пределах одного вложенного блока отступ был одинаков. И про читаемость кода не забывайте. Отступ в 1 пробел, к примеру, не лучшее решение. Используйте 4 пробела (или знак табуляции, на худой конец).
- Вложенные инструкции в Python записываются в соответствии с одним и тем же шаблоном, когда основная инструкция завершается двоеточием, вслед за которым располагается вложенный блок кода, обычно с отступом под строкой основной инструкции.

```
Основная инструкция:
    Вложенный блок инструкций
```

Несколько специальных случаев

- Иногда возможно записать несколько инструкций в одной строке, разделяя их точкой с запятой:

```
a = 1; b = 2; print(a, b)
```

Но не делайте это слишком часто! Помните об удобочитаемости. А лучше вообще так не делайте.

- Допустимо записывать одну инструкцию в нескольких строках. Достаточно ее заключить в пару круглых, квадратных или фигурных скобок:

```
if (a == 1 and b == 2 and
    c == 3 and d == 4):  # Не забываем про двоеточие
      print('spam' * 3)
```

- Тело составной инструкции может располагаться в той же строке, что и тело основной, если тело составной инструкции не содержит составных инструкций. Ну я думаю, вы поняли :). Давайте лучше пример приведу:

```
if x > y: print(x)
```

Полное понимание синтаксиса, конечно, приходит с опытом, поэтому я советую вам заглянуть в рубрику “Примеры программ”.

Также советую прочитать PEP 8 — руководство по написанию кода на Python и Документирование кода в Python. PEP 257.

Глава 5

Программа не работает. Что делать?

Моя программа не работает! Что делать? В данной статье я постараюсь собрать наиболее частые ошибки начинающих программировать на python 3, а также расскажу, как их исправлять.

Проблема: Моя программа не запускается. На доли секунды появляется чёрное окошко, а затем исчезает.

Причина: после окончания выполнения программы (после выполнения всего кода или при возникновении исключения программа закрывается. И если вы её вызвали двойным кликом по иконке (а вы, скорее всего, вызвали её именно так), то она закроется вместе с окошком, в котором находится вывод программы.

Решение: запускать программу через IDLE или через консоль.

Проблема: Не работает функция input. Пишет SyntaxError.

Пример кода:

```
>>> a = input()
hello world
  File "<string>", line 1
    hello world
              ^
SyntaxError: unexpected EOF while parsing
```

Причина: Вы запустили Python 2.

Решение: Установить Python 3.

Проблема: Где-то увидел простую программу, а она не работает.

Пример кода:

```
name = raw_input()
print name
```

Ошибка:

```
  File "a.py", line 3
    print name
             ^
SyntaxError: invalid syntax
```

Причина: Вам подсунули программу на Python 2.

Решение: Прочитать об отличиях Python 2 от Python 3. Переписать её на Python 3. Например, данная программа на Python 3 будет выглядеть так:

```
name = input()
print(name)
```

Проблема: TypeError: Can’t convert ‘int’ object to str implicitly.

Пример кода:

```
>>> a = input() + 5
8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Причина: Нельзя складывать строку с числом.

Решение: Привести строку к числу с помощью функции int(). Кстати, заметьте, что функция input() всегда возвращает строку!

```
>>> a = int(input()) + 5
8
>>> a
13
```

Проблема: SyntaxError: invalid syntax.

Пример кода:

```
a = 5
if a == 5
    print('Ура!')
```

Ошибка:

```
  File "a.py", line 3
    if a == 5
            ^
SyntaxError: invalid syntax
```

Причина: Забыто двоеточие.

Решение:

```
a = 5
if a == 5:
    print('Ура!')
```

Проблема: SyntaxError: invalid syntax.

Пример кода:

```
a = 5
if a = 5:
    print('Ура!')
```

Ошибка:

```
  File "a.py", line 3
    if a = 5
         ^
SyntaxError: invalid syntax
```

Причина: Забыто равно.

Решение:

```
a = 5
if a == 5:
    print('Ура!')
```

Проблема: NameError: name 'a' is not defined.

Пример кода:

```
print(a)
```

Причина: Переменная "a" не существует. Возможно, вы опечатались в названии или забыли инициализировать её.

Решение: Исправить опечатку.

```
a = 10
print(a)
```

Проблема: IndentationError: expected an indented block.

Пример кода:

```
a = 10
if a > 0:
print(a)
```

Причина: Нужен отступ.

Решение:

```
a = 10
if a > 0:
    print(a)
```

Проблема: TabError: inconsistent use of tabs and spaces in indentation.

Пример кода:

```
a = 10
if a > 0:
    print(a)
    print('Ура!')
```

Ошибка:

```
  File "a.py", line 5
    print('Ура!')
                ^
TabError: inconsistent use of tabs and spaces in indentation
```

Причина: Смешение пробелов и табуляции в отступах.

Решение: Исправить отступы.

```
a = 10
if a > 0:
    print(a)
    print('Ура!')
```

Проблема: UnboundLocalError: local variable 'a' referenced before assignment.

Пример кода:

```
def f():
    a += 1
    print(a)

a = 10
f()
```

Ошибка:

```
Traceback (most recent call last):
  File "a.py", line 7, in <module>
    f()
  File "a.py", line 3, in f
    a += 1
UnboundLocalError: local variable 'a' referenced before assignment
```

Причина: Попытка обратиться к локальной переменной, которая ещё не создана.

Решение:

```
def f():
    global a
    a += 1
    print(a)

a = 10
f()
```

Проблема: Программа выполнилась, но в файл ничего не записалось / записалось не всё.

Пример кода:

```
>>> f = open('output.txt', 'w', encoding='utf-8')
>>> f.write('bla')
3
>>>
```

Причина: Не закрыт файл, часть данных могла остаться в буфере.

Решение:

```
>>> f = open('output.txt', 'w', encoding='utf-8')
>>> f.write('bla')
3
>>> f.close()
>>>
```

Проблема: Здесь может быть ваша проблема. Комментарии чуть ниже :)

Также вам может быть полезно это описание:

Глава 6

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.

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.

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.

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'
```

Глава 7

Циклы for и while, операторы break и continue, волшебное слово else

В этой статье я расскажу о **циклах for и while**, операторах **break** и **continue**, а также о слове **else**, которое, будучи употребленное с циклом, может сделать программный код несколько более понятным.

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
```

Цикл for

Цикл for уже чуточку сложнее, чуть менее универсальный, но выполняется гораздо быстрее цикла while. Этот цикл проходится по любому итерируемому объекту (например стро-

ке или списку), и во время каждого прохода выполняет тело цикла.

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

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
```

Оператор break

Оператор break досрочно прерывает цикл.

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

Волшебное слово 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
```

Глава 8

Ключевые слова, модуль keyword

Сегодня я по-быстрому пробегусь по всем ключевым словам в Python, а заодно и по модулю keyword.

Ключевые слова

False - ложь.

True - правда.

None - “пустой” объект.

and - логическое И.

with / as - менеджер контекста.

assert условие - возбуждает исключение, если условие ложно.

break - выход из цикла.

class - пользовательский тип, состоящий из методов и атрибутов.

continue - переход на следующую итерацию цикла.

def - определение функции.

del - удаление объекта.

elif - в противном случае, если.

else - см. for/else или if/else.

except - перехватить исключение.

finally - вкупе с инструкцией try, выполняет инструкции независимо от того, было ли исключение или нет.

for - цикл for.

from - импорт нескольких функций из модуля.

global - позволяет сделать значение переменной, присвоенное ей внутри функции, доступным и за пределами этой функции.

if - если.

import - импорт модуля.

in - проверка на вхождение.

is - ссылаются ли 2 объекта на одно и то же место в памяти.

lambda - определение анонимной функции.

nonlocal - позволяет сделать значение переменной, присвоенное ей внутри функции, доступным в объемлющей инструкции.

not - логическое НЕ.

or - логическое ИЛИ.

pass - ничего не делающая конструкция.

raise - возбудить исключение.

return - вернуть результат.

try - выполнить инструкции, перехватывая исключения.

while - цикл while.

yield - определение функции-генератора.

Модуль keyword

В общем-то, keyword - не такой уж и модуль, но все же.

keyword.kwlist - список всех доступных ключевых слов.

keyword.iskeyword(строка) - является ли строка ключевым словом.

Глава 9

Встроенные функции

Краткий обзор встроенных функций в Python 3.

Встроенные функции, выполняющие преобразование типов

bool(x) - преобразование к типу bool, использующая стандартную процедуру проверки истинности. Если x является ложным или опущен, возвращает значение False, в противном случае она возвращает True.

bytearray([источник [, кодировка [ошибки]]]) - преобразование к bytearray. Bytearray - изменяемая последовательность целых чисел в диапазоне 0<=X<256. Вызванная без аргументов, возвращает пустой массив байт.

bytes([источник [, кодировка [ошибки]]]) - возвращает объект типа bytes, который является неизменяемой последовательностью целых чисел в диапазоне 0<=X<256. Аргументы конструктора интерпретируются как для bytearray().

complex([real[, imag]]) - преобразование к комплексному числу.

dict([object]) - преобразование к словарю.

float([X]) - преобразование к числу с плавающей точкой. Если аргумент не указан, возвращается 0.0.

frozenset([последовательность]) - возвращает неизменяемое множество.

int([object], [основание системы счисления]) - преобразование к целому числу.

list([object]) - создает список.

memoryview([object]) - создает объект memoryview.

object() - возвращает безликий объект, являющийся базовым для всех объектов.

range([start=0], stop, [step=1]) - арифметическая прогрессия от start до stop с шагом step.

set([object]) - создает множество.

slice([start=0], stop, [step=1]) - объект среза от start до stop с шагом step.

str([object], [кодировка], [ошибки]) - строковое представление объекта. Использует метод __str__.

tuple(obj) - преобразование к кортежу.

Другие встроенные функции

abs(x) - Возвращает абсолютную величину (модуль числа).

all(последовательность) - Возвращает True, если все элементы истинные (или, если последовательность пуста).

any(последовательность) - Возвращает True, если хотя бы один элемент - истина. Для пустой последовательности возвращает False.

ascii(object) - Как repr(), возвращает строку, содержащую представление объекта, но заменяет не-ASCII символы на экранированные последовательности.

bin(x) - Преобразование целого числа в двоичную строку.

callable(x) - Возвращает True для объекта, поддерживающего вызов (как функции).

chr(x) - Возвращает односимвольную строку, код символа которой равен x.

classmethod(x) - Представляет указанную функцию методом класса.

compile(source, filename, mode, flags=0, dont_inherit=False) - Компиляция в программный код, который впоследствии может выполниться функцией eval или exec. Строка не должна содержать символов возврата каретки или нулевые байты.

delattr(object, name) - Удаляет атрибут с именем 'name'.

dir([object]) - Список имен объекта, а если объект не указан, список имен в текущей локальной области видимости.

divmod(a, b) - Возвращает частное и остаток от деления a на b.

enumerate(iterable, start=0) - Возвращает итератор, при каждом проходе предоставляющем кортеж из номера и соответствующего члена последовательности.

eval(expression, globals=None, locals=None) - Выполняет строку программного кода.

exec(object[, globals[, locals]]) - Выполняет программный код на Python.

filter(function, iterable) - Возвращает итератор из тех элементов, для которых function возвращает истину.

format(value[,format_spec]) - Форматирование (обычно форматирование строки).

getattr(object, name ,[default]) - извлекает атрибут объекта или default.

globals() - Словарь глобальных имен.

hasattr(object, name) - Имеет ли объект атрибут с именем 'name'.

hash(x) - Возвращает хеш указанного объекта.

help([object]) - Вызов встроенной справочной системы.

hex(x) - Преобразование целого числа в шестнадцатеричную строку.

id(object) - Возвращает "адрес" объекта. Это целое число, которое гарантированно будет уникальным и постоянным для данного объекта в течение срока его существования.

input([prompt]) - Возвращает введенную пользователем строку. Prompt - подсказка пользователю.

isinstance(object, ClassInfo) - Истина, если объект является экземпляром ClassInfo или его подклассом. Если объект не является объектом данного типа, функция всегда возвращает ложь.

issubclass(класс, ClassInfo) - Истина, если класс является подклассом ClassInfo. Класс считается подклассом себя.

iter(x) - Возвращает объект итератора.

len(x) - Возвращает число элементов в указанном объекте.

locals() - Словарь локальных имен.

map(function, iterator) - Итератор, получившийся после применения к каждому элементу последовательности функции function.

max(iter, [args …] * [, key]) - Максимальный элемент последовательности.

min(iter, [args …] * [, key]) - Минимальный элемент последовательности.

next(x) - Возвращает следующий элемент итератора.

oct(x) - Преобразование целого числа в восьмеричную строку.

open(file, mode='r', buffering=None, encoding=None, errors=None, newline=None, closefd=True) - Открывает файл и возвращает соответствующий поток.

ord(c) - Код символа.

pow(x, y[, r]) - (x ** y) % r.

reversed(object) - Итератор из развернутого объекта.

repr(obj) - Представление объекта.

print([object, …], *, sep=" ", end='\n', file=sys.stdout) - Печать.

property(fget=None, fset=None, fdel=None, doc=None)

round(X [, N]) - Округление до N знаков после запятой.

setattr(объект, имя, значение) - Устанавливает атрибут объекта.

sorted(iterable[, key][, reverse]) - Отсортированный список.

staticmethod(function) - Статический метод для функции.

sum(iter, start=0) - Сумма членов последовательности.

super([тип [, объект или тип]]) - Доступ к родительскому классу.

type(object) - Возвращает тип объекта.

type(name, bases, dict) - Возвращает новый экземпляр класса name.

vars([object]) - Словарь из атрибутов объекта. По умолчанию - словарь локальных имен.

zip(*iters) - Итератор, возвращающий кортежи, состоящие из соответствующих элементов аргументов-последовательностей.

Глава 10

Числа: целые, вещественные, комплексные

Числа в Python 3: целые, вещественные, комплексные. Работа с числами и операции над ними.

Целые числа (int)

Числа в Python 3 ничем не отличаются от обычных чисел. Они поддерживают набор самых обычных математических операций:

x + y	Сложение
x - y	Вычитание
x * y	Умножение
x / y	Деление
x // y	Получение целой части от деления
x % y	Остаток от деления
-x	Смена знака числа
abs(x)	Модуль числа
divmod(x, y)	Пара (x // y, x % y)
x ** y	Возведение в степень
pow(x, y[, z])	x^y по модулю (если модуль задан)

Также нужно отметить, что целые числа в python 3, в отличие от многих других языков, поддерживают длинную арифметику (однако, это требует больше памяти).

```
>>> 255 + 34
289
>>> 5 * 2
10
>>> 20 / 3
```

```
6.666666666666667
>>> 20 // 3
6
>>> 20 % 3
2
>>> 3 ** 4
81
>>> pow(3, 4)
81
>>> pow(3, 4, 27)
0
>>> 3 ** 150
369988485035126972924700782451696644186473100389722973815184405301748249
```

Битовые операции

Над целыми числами также можно производить битовые операции

x | y	Побитовое *или*
x ^ y	Побитовое *исключающее или*
x & y	Побитовое *и*
x << n	Битовый сдвиг влево
x >> y	Битовый сдвиг вправо
~x	Инверсия битов

Дополнительные методы

int.bit_length() - количество бит, необходимых для представления числа в двоичном виде, без учёта знака и лидирующих нулей.

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

int.to_bytes(length, byteorder, *, signed=False) - возвращает строку байтов, представляющих это число.

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() // 8) + 1, byteorder='little')
b'\xe8\x03'
```

classmethod **int.from_bytes**(bytes, byteorder, *, signed=False) - возвращает число из данной строки байтов.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

Системы счисления

Те, у кого в школе была информатика, знают, что числа могут быть представлены не только в десятичной системе счисления. К примеру, в компьютере используется двоичный код, и, к примеру, число 19 в двоичной системе счисления будет выглядеть как 10011. Также иногда нужно переводить числа из одной системы счисления в другую. Python для этого предоставляет несколько функций:

- **int**([object], [основание системы счисления]) - преобразование к целому числу в десятичной системе счисления. По умолчанию система счисления десятичная, но можно задать любое основание от 2 до 36 включительно.
- **bin**(x) - преобразование целого числа в двоичную строку.
- **hex**(x) - преобразование целого числа в шестнадцатеричную строку.
- **oct**(x) - преобразование целого числа в восьмеричную строку.

Примеры:

```
>>> a = int('19') # Переводим строку в число
>>> b = int('19.5')  # Строка не является целым числом
Traceback (most recent call last):
  File "", line 1, in
ValueError: invalid literal for int() with base 10: '19.5'
>>> c = int(19.5)  # Применённая к числу с плавающей точкой, отсекает дробную часть
>>> print(a, c)
19 19
>>> bin(19)
'0b10011'
>>> oct(19)
'0o23'
>>> hex(19)
'0x13'
>>> 0b10011  # Так тоже можно записывать числовые константы
19
>>> int('10011', 2)
```

```
19
>>> int('0b10011', 2)
19
```

Вещественные числа (float)

Вещественные числа поддерживают те же операции, что и целые. Однако (из-за представления чисел в компьютере) вещественные числа неточны, и это может привести к ошибкам:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
0.9999999999999999
```

Для высокой точности используют другие объекты (например Decimal и Fraction)).

Также вещественные числа не поддерживают длинную арифметику:

```
>>> a = 3 ** 1000
>>> a + 0.1
Traceback (most recent call last):
  File "", line 1, in
OverflowError: int too large to convert to float
```

Простенькие примеры работы с числами:

```
>>> c = 150
>>> d = 12.9
>>> c + d
162.9
>>> p = abs(d - c)  # Модуль числа
>>> print(p)
137.1
>>> round(p)  # Округление
137
```

Дополнительные методы

float.as_integer_ratio() - пара целых чисел, чьё отношение равно этому числу.

float.is_integer() - является ли значение целым числом.

float.hex() - переводит float в hex (шестнадцатеричную систему счисления).

classmethod **float.fromhex**(s) - float из шестнадцатеричной строки.

```
>>> (10.5).hex()
'0x1.5000000000000p+3'
```

```
>>> float.fromhex('0x1.5000000000000p+3')
10.5
```

Помимо стандартных выражений для работы с числами (а в Python их не так уж и много), в составе Python есть несколько полезных модулей.

Модуль math предоставляет более сложные математические функции.

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(85)
9.219544457292887
```

Модуль random реализует генератор случайных чисел и функции случайного выбора.

```
>>> import random
>>> random.random()
0.15651968855132303
```

Комплексные числа (complex)

В Python встроены также и комплексные числа:

```
>>> x = complex(1, 2)
>>> print(x)
(1+2j)
>>> y = complex(3, 4)
>>> print(y)
(3+4j)
>>> z = x + y
>>> print(x)
(1+2j)
>>> print(z)
(4+6j)
>>> z = x * y
>>> print(z)
(-5+10j)
>>> z = x / y
>>> print(z)
(0.44+0.08j)
>>> print(x.conjugate())  # Сопряжённое число
(1-2j)
>>> print(x.imag)  # Мнимая часть
2.0
>>> print(x.real)  # Действительная часть
1.0
>>> print(x > y)  # Комплексные числа нельзя сравнить
Traceback (most recent call last):
```

```
  File "", line 1, in
TypeError: unorderable types: complex() > complex()
>>> print(x == y)  # Но можно проверить на равенство
False
>>> abs(3 + 4j)  # Модуль комплексного числа
5.0
>>> pow(3 + 4j, 2)  # Возведение в степень
(-7+24j)
```

Также для работы с комплексными числами используется также модуль cmath.

Глава 11

Работа со строками в Python: литералы

Строки в Python - упорядоченные последовательности символов, используемые для хранения и представления текстовой информации, поэтому с помощью строк можно работать со всем, что может быть представлено в текстовой форме.

Это первая часть о работе со строками, а именно о литералах строк.

Литералы строк

Работа со строками в Python очень удобна. Существует несколько литералов строк, которые мы сейчас и рассмотрим.

Строки в апострофах и в кавычках

```
S = 'spam"s'
S = "spam's"
```

Строки в апострофах и в кавычках - одно и то же. Причина наличия двух вариантов в том, чтобы позволить вставлять в литералы строк символы кавычек или апострофов, не используя экранирование.

Экранированные последовательности - служебные символы

Экранированные последовательности позволяют вставить символы, которые сложно ввести с клавиатуры.

Экранированная последовательность	Назначение
\n	Перевод строки
\a	Звонок
\b	Забой
\f	Перевод страницы
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\N{id}	Идентификатор ID базы данных Юникода
\uhhhh	16-битовый символ Юникода в 16-ричном представлении
\Uhhhh…	32-битовый символ Юникода в 32-ричном представлении
\xhh	16-ричное значение символа
\ooo	8-ричное значение символа
\0	Символ Null (не является признаком конца строки)

"Сырые" строки - подавляют экранирование

Если перед открывающей кавычкой стоит символ 'r' (в любом регистре), то механизм экранирования отключается.

```
S = r'C:\newt.txt'
```

Но, несмотря на назначение, "сырая" строка не может заканчиваться символом обратного слэша. Пути решения:

```
S = r'\n\n\\'[:-1]
S = r'\n\n' + '\\'
S = '\\n\\n'
```

Строки в тройных апострофах или кавычках

Главное достоинство строк в тройных кавычках в том, что их можно использовать для записи многострочных блоков текста. Внутри такой строки возможно присутствие кавычек и апострофов, главное, чтобы не было трех кавычек подряд.

```
>>> c = '''это очень большая
... строка, многострочный
... блок текста'''
>>> c
'это очень большая\nстрока, многострочный\nблок текста'
>>> print(c)
это очень большая
строка, многострочный
блок текста
```

Это все о литералах строк и работе с ними. О функциях и методах строк я расскажу в следующей статье.

Глава 12

Строки. Функции и методы строк

Итак, о работе со строками мы немного поговорили, теперь поговорим о **функциях и методах строк**.

Я постарался собрать здесь все строковые методы и функции, но если я что-то забыл - поправляйте.

Базовые операции

- Конкатенация (сложение)

```
>>> S1 = 'spam'
>>> S2 = 'eggs'
>>> print(S1 + S2)
'spameggs'
```

- Дублирование строки

```
>>> print('spam' * 3)
spamspamspam
```

- Длина строки (функция len)

```
>>> len('spam')
4
```

- Доступ по индексу

```
>>> S = 'spam'
>>> S[0]
```

```
's'
>>> S[2]
'a'
>>> S[-2]
'a'
```

Как видно из примера, в Python возможен и доступ по отрицательному индексу, при этом отсчет идет от конца строки.

- Извлечение среза

  Оператор извлечения среза: [X:Y]. X – начало среза, а Y – окончание;

  символ с номером Y в срез не входит. По умолчанию первый индекс равен 0, а второй - длине строки.

```
>>> s = 'spameggs'
>>> s[3:5]
'me'
>>> s[2:-2]
'ameg'
>>> s[:6]
'spameg'
>>> s[1:]
'pameggs'
>>> s[:]
'spameggs'
```

Кроме того, можно задать шаг, с которым нужно извлекать срез.

```
>>> s[::-1]
'sggemaps'
>>> s[3:5:-1]
''
>>> s[2::2]
'aeg'
```

Другие функции и методы строк

При вызове методов необходимо помнить, что строки в Python относятся к категории неизменяемых последовательностей, то есть все функции и методы могут лишь создавать новую строку.

```
>>> s = 'spam'
>>> s[1] = 'b'
Traceback (most recent call last):
  File "", line 1, in
    s[1] = 'b'
TypeError: 'str' object does not support item assignment
```

```
>>> s = s[0] + 'b' + s[2:]
>>> s
'sbam'
```

Поэтому все строковые методы возвращают новую строку, которую потом следует присвоить переменной.

Таблица "Функции и методы строк"

Функция или метод	Назначение
S = 'str'; S = "str"; S = ''str''; S = """str"""	Литералы строк
S = "s\np\ta\nbbb"	Экранированные последовательности
S = r"C:\temp\new"	Неформатированные строки (подавляют экранирование)
S = b"byte"	Строка байтов
S1 + S2	Конкатенация (сложение строк)
S1 * 3	Повторение строки
S[i]	Обращение по индексу
S[i:j:step]	Извлечение среза
len(S)	Длина строки
S.find(str, [start],[end])	Поиск подстроки в строке. Возвращает номер первого вхождения или -1
S.rfind(str, [start],[end])	Поиск подстроки в строке. Возвращает номер последнего вхождения или -1
S.index(str, [start],[end])	Поиск подстроки в строке. Возвращает номер первого вхождения или вызывает ValueError
S.rindex(str, [start],[end])	Поиск подстроки в строке. Возвращает номер последнего вхождения или вызывает ValueError
S.replace(шаблон, замена)	Замена шаблона
S.split(символ)	Разбиение строки по разделителю
S.isdigit()	Состоит ли строка из цифр
S.isalpha()	Состоит ли строка из букв
S.isalnum()	Состоит ли строка из цифр или букв
S.islower()	Состоит ли строка из символов в нижнем регистре
S.isupper()	Состоит ли строка из символов в верхнем регистре

Продолжается на следующей странице

Таблица 12.1 – продолжение с предыдущей страницы

Функция или метод	Назначение
S.isspace()	Состоит ли строка из неотображаемых символов (пробел, символ перевода страницы ('\f'), "новая строка" ('\n'), "перевод каретки" ('\r'), "горизонтальная табуляция" ('\t') и "вертикальная табуляция" ('\v'))
S.istitle()	Начинаются ли слова в строке с заглавной буквы
S.upper()	Преобразование строки к верхнему регистру
S.lower()	Преобразование строки к нижнему регистру
S.startswith(str)	Начинается ли строка S с шаблона str
S.endswith(str)	Заканчивается ли строка S шаблоном str
S.join(список)	Сборка строки из списка с разделителем S
ord(символ)	Символ в его код ASCII
chr(число)	Код ASCII в символ
S.capitalize()	Переводит первый символ строки в верхний регистр, а все остальные в нижний
S.center(width, [fill])	Возвращает отцентрованную строку, по краям которой стоит символ fill (пробел по умолчанию)
S.count(str, [start],[end])	Возвращает количество непересекающихся вхождений подстроки в диапазоне [начало, конец] (0 и длина строки по умолчанию)
S.expandtabs([tabsize])	Возвращает копию строки, в которой все символы табуляции заменяются одним или несколькими пробелами, в зависимости от текущего столбца. Если TabSize не указан, размер табуляции полагается равным 8 пробелам
S.lstrip([chars])	Удаление пробельных символов в начале строки
S.rstrip([chars])	Удаление пробельных символов в конце строки
S.strip([chars])	Удаление пробельных символов в начале и в конце строки
S.partition(шаблон)	Возвращает кортеж, содержащий часть перед первым шаблоном, сам шаблон, и часть после шаблона. Если шаблон не найден, возвращается кортеж, содержащий саму строку, а затем две пустых строки

Продолжается на следующей странице

Таблица 12.1 – продолжение с предыдущей страницы

Функция или метод	Назначение
S.rpartition(sep)	Возвращает кортеж, содержащий часть перед последним шаблоном, сам шаблон, и часть после шаблона. Если шаблон не найден, возвращается кортеж, содержащий две пустых строки, а затем саму строку
S.swapcase()	Переводит символы нижнего регистра в верхний, а верхнего – в нижний
S.title()	Первую букву каждого слова переводит в верхний регистр, а все остальные в нижний
S.zfill(width)	Делает длину строки не меньшей width, по необходимости заполняя первые символы нулями
S.ljust(width, fillchar=" ")	Делает длину строки не меньшей width, по необходимости заполняя последние символы символом fillchar
S.rjust(width, fillchar=" ")	Делает длину строки не меньшей width, по необходимости заполняя первые символы символом fillchar
S.format(*args, **kwargs)	Форматирование строки

Глава 13

Форматирование строк. Метод format

Иногда (а точнее, довольно часто) возникают ситуации, когда нужно сделать строку, подставив в неё некоторые данные, полученные в процессе выполнения программы (пользовательский ввод, данные из файлов и т. д.). Подстановку данных можно сделать с помощью форматирования строк. Форматирование можно сделать с помощью оператора %, либо с помощью метода format.

Форматирование строк с помощью метода format

Если для подстановки требуется только один аргумент, то значение - сам аргумент:

```
>>> 'Hello, {}!'.format('Vasya')
'Hello, Vasya!'
```

А если несколько, то значениями будут являться все аргументы со строками подстановки (обычных или именованных):

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')
'abracadabra'
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-
↪115.81W')
```

```
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

Однако метод format умеет большее. Вот его синтаксис:

```
поле замены     ::=  "{" [имя поля] ["!" преобразование] [":" спецификация] "}"
имя поля        ::=  arg_name ("." имя атрибута | "[" индекс "]")*
преобразование  ::=  "r" (внутреннее представление) | "s" (человеческое␣
↪представление)
спецификация    ::=  см. ниже
```

Например:

```
>>> "Units destroyed: {players[0]}".format(players = [1, 2, 3])
'Units destroyed: 1'
>>> "Units destroyed: {players[0]!r}".format(players = ['1', '2', '3'])
"Units destroyed: '1'"
```

Теперь спецификация формата:

```
спецификация ::=  [[fill]align][sign][#][0][width][,][.precision][type]
заполнитель  ::=  символ кроме '{' или '}'
выравнивание ::=  "<" | ">" | "=" | "^"
знак         ::=  "+" | "-" | " "
ширина       ::=  integer
точность     ::=  integer
тип          ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" |
                  "n" | "o" | "s" | "x" | "X" | "%"
```

Выравнивание производится при помощи символа-заполнителя. Доступны следующие варианты выравнивания:

Флаг	Значение
'<'	Символы-заполнители будут справа (выравнивание объекта по левому краю) (по умолчанию).
'>'	выравнивание объекта по правому краю.
'='	Заполнитель будет после знака, но перед цифрами. Работает только с числовыми типами.
'^'	Выравнивание по центру.

Опция "знак" используется только для чисел и может принимать следующие значения:

Флаг	Значение
'+'	Знак должен быть использован для всех чисел.
'-'	'-' для отрицательных, ничего для положительных.
'Пробел'	'-' для отрицательных, пробел для положительных.

Поле "тип" может принимать следующие значения:

Тип	Значение
'd', 'i', 'u'	Десятичное число.
'o'	Число в восьмеричной системе счисления.
'x'	Число в шестнадцатеричной системе счисления (буквы в нижнем регистре).
'X'	Число в шестнадцатеричной системе счисления (буквы в верхнем регистре).
'e'	Число с плавающей точкой с экспонентой (экспонента в нижнем регистре).
'E'	Число с плавающей точкой с экспонентой (экспонента в верхнем регистре).
'f', 'F'	Число с плавающей точкой (обычный формат).
'g'	Число с плавающей точкой. с экспонентой (экспонента в нижнем регистре), если она меньше, чем -4 или точности, иначе обычный формат.
'G'	Число с плавающей точкой. с экспонентой (экспонента в верхнем регистре), если она меньше, чем -4 или точности, иначе обычный формат.
'c'	Символ (строка из одного символа или число - код символа).
's'	Строка.
'%'	Число умножается на 100, отображается число с плавающей точкой, а за ним знак %.

И напоследок, несколько примеров:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f};
↪{:f}'
'3.140000; -3.140000'
>>> # format also supports binary numbers
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
>>> points = 19.5
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 88.64%'
```

Глава 14

Списки (list). Функции и методы списков

Сегодня я расскажу о таком типе данных, как **списки**, операциях над ними и методах, о генераторах списков и о применении списков.

Что такое 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.

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).

Глава 15

Индексы и срезы

Сегодня мы поговорим об операциях взятия индекса и среза.

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
```

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]
```

Глава 16

Кортежи (tuple)

Сегодня я расскажу о таком типе данных, как **кортежи** (**tuple**) и о том, где они применяются.

Кортеж, по сути - неизменяемый список.

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'
```

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', '!')
```

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
```

Глава 17

Словари (dict) и работа с ними. Методы словарей

Сегодня я расскажу о таком типе данных, как **словари**, о работе со словарями, операциях над ними, методах, о генераторах словарей.

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. Создать его 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.

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 с 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.

Глава 18

Множества (set и frozenset)

Доброго времени суток! Сегодня я расскажу о работе с множествами в python, операциях над ними и покажу примеры их применения.

Что такое 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.

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'
```

Глава 19

Функции и их аргументы

В этой статье я планирую рассказать о функциях, именных и анонимных, инструкциях def, return и lambda, обязательных и необязательных аргументах функции, функциях с произвольным числом аргументов.

Именные функции, инструкция def

Функция в python - объект, принимающий аргументы и возвращающий значение. Обычно функция определяется с помощью инструкции **def**.

Определим простейшую функцию:

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

Инструкция **return** говорит, что нужно вернуть значение. В нашем случае функция возвращает сумму x и y.

Теперь мы ее можем вызвать:

```
>>> add(1, 10)
11
>>> add('abc', 'def')
'abcdef'
```

Функция может быть любой сложности и возвращать любые объекты (списки, кортежи, и даже функции!):

```
>>> def newfunc(n):
...     def myfunc(x):
```

```
...         return x + n
...     return myfunc
...
>>> new = newfunc(100)  # new - это функция
>>> new(200)
300
```

Функция может и не заканчиваться инструкцией return, при этом функция вернет значение None:

```
>>> def func():
...     pass
...
>>> print(func())
None
```

Аргументы функции

Функция может принимать произвольное количество аргументов или не принимать их вовсе. Также распространены функции с произвольным числом аргументов, функции с позиционными и именованными аргументами, обязательными и необязательными.

```
>>> def func(a, b, c=2): # c - необязательный аргумент
...     return a + b + c
...
>>> func(1, 2)  # a = 1, b = 2, c = 2 (по умолчанию)
5
>>> func(1, 2, 3)  # a = 1, b = 2, c = 3
6
>>> func(a=1, b=3)  # a = 1, b = 3, c = 2
6
>>> func(a=3, c=6)  # a = 3, c = 6, b не определен
Traceback (most recent call last):
  File "", line 1, in
    func(a=3, c=6)
TypeError: func() takes at least 2 arguments (2 given)
```

Функция также может принимать переменное количество позиционных аргументов, тогда перед именем ставится *:

```
>>> def func(*args):
...     return args
...
>>> func(1, 2, 3, 'abc')
(1, 2, 3, 'abc')
>>> func()
()
>>> func(1)
```

```
(1,)
```

Как видно из примера, args - это кортеж из всех переданных аргументов функции, и с переменной можно работать также, как и с кортежем.

Функция может принимать и произвольное число именованных аргументов, тогда перед именем ставится **:

```
>>> def func(**kwargs):
...     return kwargs
...
>>> func(a=1, b=2, c=3)
{'a': 1, 'c': 3, 'b': 2}
>>> 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)
```

Глава 20

Исключения в python. Конструкция try - except для обработки исключений

Исключения (exceptions) - ещё один тип данных в python. Исключения необходимы для того, чтобы сообщать программисту об ошибках.

Самый простейший пример исключения - деление на ноль:

```
>>> 100 / 0
Traceback (most recent call last):
  File "", line 1, in
    100 / 0
ZeroDivisionError: division by zero
```

В данном случае интерпретатор сообщил нам об исключении ZeroDivisionError, то есть делении на ноль. Также возможны и другие исключения:

```
>>> 2 + '1'
Traceback (most recent call last):
  File "", line 1, in
    2 + '1'
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> int('qwerty')
Traceback (most recent call last):
  File "", line 1, in
    int('qwerty')
ValueError: invalid literal for int() with base 10: 'qwerty'
```

В этих двух примерах генерируются исключения TypeError и ValueError соответственно. Подсказки дают нам полную информацию о том, где порождено исключение, и с чем оно связано.

Рассмотрим иерархию встроенных в python исключений, хотя иногда вам могут встретиться и другие, так как программисты могут создавать собственные исключения. Данный список актуален для python 3.3, в более ранних версиях есть незначительные изменения.

- **BaseException** - базовое исключение, от которого берут начало все остальные.
  - **SystemExit** - исключение, порождаемое функцией sys.exit при выходе из программы.
  - **KeyboardInterrupt** - порождается при прерывании программы пользователем (обычно сочетанием клавиш Ctrl+C).
  - **GeneratorExit** - порождается при вызове метода close объекта generator.
  - **Exception** - а вот тут уже заканчиваются полностью системные исключения (которые лучше не трогать) и начинаются обыкновенные, с которыми можно работать.
    - **StopIteration** - порождается встроенной функцией next, если в итераторе больше нет элементов.
    - **ArithmeticError** - арифметическая ошибка.
      - **FloatingPointError** - порождается при неудачном выполнении операции с плавающей запятой. На практике встречается нечасто.
      - **OverflowError** - возникает, когда результат арифметической операции слишком велик для представления. Не появляется при обычной работе с целыми числами (так как python поддерживает длинные числа), но может возникать в некоторых других случаях.
      - **ZeroDivisionError** - деление на ноль.
    - **AssertionError** - выражение в функции assert ложно.
    - **AttributeError** - объект не имеет данного атрибута (значения или метода).
    - **BufferError** - операция, связанная с буфером, не может быть выполнена.
    - **EOFError** - функция наткнулась на конец файла и не смогла прочитать то, что хотела.
    - **ImportError** - не удалось импортирование модуля или его атрибута.
    - **LookupError** - некорректный индекс или ключ.
      - **IndexError** - индекс не входит в диапазон элементов.
      - **KeyError - несуществующий ключ (в** словаре, множестве или другом объекте).
    - **MemoryError** - недостаточно памяти.
    - **NameError** - не найдено переменной с таким именем.
      - **UnboundLocalError - сделана ссылка на локальную переменную в** функции, но переменная не определена ранее.

* **OSError** - ошибка, связанная с системой.
  - **BlockingIOError**
  - **ChildProcessError** - неудача при операции с дочерним процессом.
  - **ConnectionError** - базовый класс для исключений, связанных с подключениями.
  - **BrokenPipeError**
  - **ConnectionAbortedError**
  - **ConnectionRefusedError**
  - **ConnectionResetError**
  - **FileExistsError** - попытка создания файла или директории, которая уже существует.
  - **FileNotFoundError** - файл или директория не существует.
  - **InterruptedError** - системный вызов прерван входящим сигналом.
  - **IsADirectoryError** - ожидался файл, но это директория.
  - **NotADirectoryError** - ожидалась директория, но это файл.
  - **PermissionError** - не хватает прав доступа.
  - **ProcessLookupError** - указанного процесса не существует.
  - **TimeoutError** - закончилось время ожидания.
* **ReferenceError** - попытка доступа к атрибуту со слабой ссылкой.
* **RuntimeError** - возникает, когда исключение не попадает ни под одну из других категорий.
* **NotImplementedError** - возникает, когда абстрактные методы класса требуют переопределения в дочерних классах.
* **SyntaxError** - синтаксическая ошибка.
  - **IndentationError** - неправильные отступы.
  - **TabError** - смешивание в отступах табуляции и пробелов.
* **SystemError** - внутренняя ошибка.
* **TypeError** - операция применена к объекту несоответствующего типа.
* **ValueError** - функция получает аргумент правильного типа, но некорректного значения.
* **UnicodeError** - ошибка, связанная с кодированием / раскодированием unicode в строках.
  - **UnicodeEncodeError** - исключение, связанное с кодированием unicode.
  - **UnicodeDecodeError** - исключение, связанное с декодированием unicode.

- **UnicodeTranslateError** - исключение, связанное с переводом unicode.

* **Warning** - предупреждение.

Теперь, зная, когда и при каких обстоятельствах могут возникнуть исключения, мы можем их обрабатывать. Для обработки исключений используется конструкция **try - except**.

Первый пример применения этой конструкции:

```
>>> try:
...     k = 1 / 0
... except ZeroDivisionError:
...     k = 0
...
>>> print(k)
0
```

В блоке try мы выполняем инструкцию, которая может породить исключение, а в блоке except мы перехватываем их. При этом перехватываются как само исключение, так и его потомки. Например, перехватывая ArithmeticError, мы также перехватываем FloatingPointError, OverflowError и ZeroDivisionError.

```
>>> try:
...     k = 1 / 0
... except ArithmeticError:
...     k = 0
...
>>> print(k)
0
```

Также возможна инструкция except без аргументов, которая перехватывает вообще всё (и прерывание с клавиатуры, и системный выход и т. д.). Поэтому в такой форме инструкция except практически не используется, а используется except Exception. Однако чаще всего перехватывают исключения по одному, для упрощения отладки (вдруг вы ещё другую ошибку сделаете, а except её перехватит).

Ещё две инструкции, относящиеся к нашей проблеме, это **finally** и else. Finally выполняет блок инструкций в любом случае, было ли исключение, или нет (применима, когда нужно непременно что-то сделать, к примеру, закрыть файл). Инструкция else выполняется в том случае, если исключения не было.

```
>>> f = open('1.txt')
>>> ints = []
>>> try:
...     for line in f:
...         ints.append(int(line))
... except ValueError:
...     print('Это не число. Выходим.')
... except Exception:
...     print('Это что ещё такое?')
... else:
...     print('Всё хорошо.')
... finally:
```

```
...     f.close()
...     print('Я закрыл файл.')
...     # Именно в таком порядке: try, группа except, затем else, и только потом␣
↪finally.
...
Это не число. Выходим.
Я закрыл файл.
```

Глава 21

Байты (bytes и bytearray)

Байтовые строки в Python - что это такое и с чем это едят? Байтовые строки очень похожи на обычные строки, но с некоторыми отличиями. Попробуем выяснить, с какими.

Что такое байты? Байт - минимальная единица хранения и обработки цифровой информации. Последовательность байт представляет собой какую-либо информацию (текст, картинку, мелодию...).

Создаём байтовую строку:

```
>>> b'bytes'
b'bytes'
>>> 'Байты'.encode('utf-8')
b'\xd0\x91\xd0\xb0\xd0\xb9\xd1\x82\xd1\x8b'
>>> bytes('bytes', encoding = 'utf-8')
b'bytes'
>>> bytes([50, 100, 76, 72, 41])
b'2dLH)'
```

Если первые три способа нам уже известны (тут, тут и тут), то последний нужно пояснить. Функция bytes принимает список чисел от 0 до 255 и возвращает байты, получающиеся применением функции chr.

```
>>> chr(50)
'2'
>>> chr(100)
'd'
>>> chr(76)
'L'
```

Что делать с байтами? Хотя байтовые строки поддерживают практически все строковые методы, с ними мало что нужно делать. Обычно их надо записать в файл / прочесть из

файла и преобразовать во что-либо другое (конечно, если очень хочется, то можно и распечатать). Для преобразования в строку используется метод decode:

```
>>> b'\xd0\x91\xd0\xb0\xd0\xb9\xd1\x82\xd1\x8b'.decode('utf-8')
'Байты'
```

Bytearray

Bytearray в python - массив байт. От типа **bytes** отличается только тем, что является изменяемым. Про него, в общем-то, больше рассказать нечего.

```
>>> b = bytearray(b'hello world!')
>>> b
bytearray(b'hello world!')
>>> b[0]
104
>>> b[0] = b'h'
Traceback (most recent call last):
  File "", line 1, in
    b[0] = b'h'
TypeError: an integer is required
>>> b[0] = 105
>>> b
bytearray(b'iello world!')
>>> for i in range(len(b)):
...     b[i] += i
...
>>> b
bytearray(b'ifnos%}vzun,')
```

Глава 22

None (null), или немного о типе NoneType

Ключевое слово null обычно используется во многих языках программирования, таких как Java, C++, C# и JavaScript. Это значение, которое присваивается переменной.

Концепция ключевого слова null в том, что она дает переменной нейтральное или "нулевое" поведение.

А что же в Python?

Эквивалент null в Python: None

Он был разработан таким образом, по двум причинам:

Многие утверждают, что слово *null* несколько эзотерично. Это не наиболее дружелюбное слово для новичков. Кроме того, **None** относится именно к требуемой функциональности - это ничего, и не имеет поведения.

Присвоить переменной значение None очень просто:

```
my_none_variable = None
```

Существует много случаев, когда следует использовать None.

Часто вы хотите выполнить действие, которое может работать либо завершиться неудачно. Используя None, вы можете проверить успех действия. Вот пример:

```
# Мы хотели бы подключиться к базе данных. Мы не знаем, верны ли логин и пароль
# Если соединение с базой будет неуспешно, то
# Он бросит исключение. Обратите внимание, что MyDatabase и DatabaseException
# НЕ являются реальными классами, мы просто используем их в качестве примеров.
```

```
try:
    database = MyDatabase(db_host, db_user, db_password, db_database)
    database_connection = database.connect()
except DatabaseException:
    pass

if database_connection is None:
    print('The database could not connect')
else:
    print('The database could connect')
```

Python является объектно-ориентированным, и поэтому None - тоже объект, и имеет свой тип.

```
>>> type(None)
<class 'NoneType'>
```

Проверка на None

Есть (формально) два способа проверить, на равенство None.

Один из способов - с помощью ключевого слова **is**.

Второй - с помощью == (но никогда не пользуйтесь вторым способом, и я попробую объяснить, почему).

Посмотрим на примеры:

```
null_variable = None
not_null_variable = 'Hello There!'

# The is keyword
if null_variable is None:
    print('null_variable is None')
else:
    print('null_variable is not None')

if not_null_variable is None:
    print('not_null_variable is None')
else:
    print('not_null_variable is not None')


# The == operator
if null_variable == None:
    print('null_variable is None')
else:
    print('null_variable is not None')
```

```
if not_null_variable == None:
    print('not_null_variable is None')
else:
    print('not_null_variable is not None')
```

Данный код выведет:

```
null_variable is None
not_null_variable is not None
null_variable is None
not_null_variable is not None
```

Отлично, так они делают одно и то же! Однако, не совсем. Для встроенных типов - да. Но с пользовательскими классами вы должны быть осторожны. Python предоставляет возможность переопределения операторов сравнения в пользовательских классах. Таким образом, вы можете сравнить классы, например, MyObject == MyOtherObject.

```
class MyClass:
    def __eq__(self, my_object):
        # Просто вернем True

        return True

my_class = MyClass()

if my_class is None:
    print('my_class is None, using the is keyword')
else:
    print('my_class is not None, using the is keyword')

if my_class == None:
    print('my_class is None, using the == syntax')
else:
    print('my_class is not None, using the == syntax')
```

И получаем немного неожиданный результат:

```
my_class is not None, using the is keyword
my_class is None, using the == syntax
```

Интересно, не правда ли? Вот поэтому нужно проверять на None с помощью ключевого слова is.

А ещё (для некоторых классов) вызов метода __eq__ может занимать много времени, и is будет просто-напросто быстрее.

Глава 23

Файлы. Работа с файлами.

В данной статье мы рассмотрим встроенные средства python для работы с файлами: открытие / закрытие, чтение и запись.

Итак, начнем. Прежде, чем работать с файлом, его надо открыть. С этим замечательно справится встроенная функция open:

```
f = open('text.txt', 'r')
```

У функции open много параметров, они указаны в статье "Встроенные функции", нам пока важны 3 аргумента: первый, это имя файла. Путь к файлу может быть относительным или абсолютным. Второй аргумент, это режим, в котором мы будем открывать файл.

Режим	Обозначение
'r'	открытие на чтение (является значением по умолчанию).
'w'	открытие на запись, содержимое файла удаляется, если файла не существует, создается новый.
'x'	открытие на запись, если файла не существует, иначе исключение.
'a'	открытие на дозапись, информация добавляется в конец файла.
'b'	открытие в двоичном режиме.
't'	открытие в текстовом режиме (является значением по умолчанию).
'+'	открытие на чтение и запись

Режимы могут быть объединены, то есть, к примеру, 'rb' - чтение в двоичном режиме. По умолчанию режим равен 'rt'.

И последний аргумент, encoding, нужен только в текстовом режиме чтения файла. Этот аргумент задает кодировку.

Чтение из файла

Открыли мы файл, а теперь мы хотим прочитать из него информацию. Для этого есть несколько способов, но большого интереса заслуживают лишь два из них.

Первый - метод read, читающий весь файл целиком, если был вызван без аргументов, и n символов, если был вызван с аргументом (целым числом n).

```
>>> f = open('text.txt')
>>> f.read(1)
'H'
>>> f.read()
'ello world!\nThe end.\n\n'
```

Ещё один способ сделать это - прочитать файл построчно, воспользовавшись циклом for:

```
>>> f = open('text.txt')
>>> for line in f:
...     line
...
'Hello world!\n'
'\n'
'The end.\n'
'\n'
```

Запись в файл

Теперь рассмотрим запись в файл. Попробуем записать в файл вот такой вот список:

```
>>> l = [str(i)+str(i-1) for i in range(20)]
>>> l
['0-1', '10', '21', '32', '43', '54', '65', '76', '87', '98', '109', '1110', '1211
↪', '1312', '1413', '1514', '1615', '1716', '1817', '1918']
```

Откроем файл на запись:

```
>>> f = open('text.txt', 'w')
```

Запись в файл осуществляется с помощью метода write:

```
>>> for index in l:
...     f.write(index + '\n')
...
4
3
3
3
3
```

Для тех, кто не понял, что это за цифры, поясню: метод write возвращает число записанных символов.

После окончания работы с файлом его **обязательно нужно закрыть** с помощью метода close:

```
>>> f.close()
```

Теперь попробуем воссоздать этот список из получившегося файла. Откроем файл на чтение (надеюсь, вы поняли, как это сделать?), и прочитаем строки.

```
>>> f = open('text.txt', 'r')
>>> l = [line.strip() for line in f]
>>> l
['0-1', '10', '21', '32', '43', '54', '65', '76', '87', '98', '109', '1110', '1211
↪', '1312', '1413', '1514', '1615', '1716', '1817', '1918']
>>> f.close()
```

Мы получили тот же список, что и был. В более сложных случаях (словарях, вложенных кортежей и т. д.) алгоритм записи придумать сложнее. Но это и не нужно. В python уже давно придумали средства, такие как pickle или json, позволяющие сохранять в файле сложные структуры.

Глава 24

With … as - менеджеры контекста

Конструкция with … as используется для оборачивания выполнения блока инструкций менеджером контекста. Иногда это более удобная конструкция, чем try…except…finally.

Синтаксис конструкции with … as:

```
"with" expression ["as" target] ("," expression ["as" target])* ":"
    suite
```

Теперь по порядку о том, что происходит при выполнении данного блока:

1. Выполняется выражение в конструкции with … as.
2. Загружается специальный метод __exit__ для дальнейшего использования.
3. Выполняется метод __enter__. Если конструкция with включает в себя слово as, то возвращаемое методом __enter__ значение записывается в переменную.
4. Выполняется suite.
5. Вызывается метод __exit__, причём неважно, выполнилось ли suite или произошло исключение. В этот метод передаются параметры исключения, если оно произошло, или во всех аргументах значение None, если исключения не было.

Если в конструкции with - as было несколько выражений, то это эквивалентно нескольким вложенным конструкциям:

```
with A() as a, B() as b:
    suite
```

эквивалентно

```
with A() as a:
    with B() as b:
        suite
```

Для чего применяется конструкция with … as? Для гарантии того, что критические функции выполнятся в любом случае. Самый распространённый пример использования этой конструкции - открытие файлов. Я уже рассказывал об открытии файлов с помощью функции open, однако конструкция with … as, как правило, является более удобной и гарантирует закрытие файла в любом случае.

Например:

```
with open('newfile.txt', 'w', encoding='utf-8') as g:
    d = int(input())
    print('1 / {} = {}'.format(d, 1 / d), file=g)
```

И вы можете быть уверены, что файл будет закрыт вне зависимости от того, что введёт пользователь.

Глава 25

PEP 8 - руководство по написанию кода на Python

Этот документ описывает соглашение о том, как писать код для языка python, включая стандартную библиотеку, входящую в состав python.

PEP 8 создан на основе рекомендаций Гуидо ван Россума с добавлениями от Барри. Если где-то возникал конфликт, мы выбирали стиль Гуидо. И, конечно, этот PEP может быть неполным (фактически, он, наверное, никогда не будет закончен).

Ключевая идея Гуидо такова: код читается намного больше раз, чем пишется. Собственно, рекомендации о стиле написания кода направлены на то, чтобы улучшить читаемость кода и сделать его согласованным между большим числом проектов. В идеале, весь код будет написан в едином стиле, и любой сможет легко его прочесть.

Это руководство о согласованности и единстве. Согласованность с этим руководством очень важна. Согласованность внутри одного проекта еще важнее. А согласованность внутри модуля или функции — самое важное. Но важно помнить, что иногда это руководство неприменимо, и понимать, когда можно отойти от рекомендаций. Когда вы сомневаетесь, просто посмотрите на другие примеры и решите, какой выглядит лучше.

Две причины для того, чтобы нарушить данные правила:

1. Когда применение правила сделает код менее читаемым даже для того, кто привык читать код, который следует правилам.
2. Чтобы писать в едином стиле с кодом, который уже есть в проекте и который нарушает правила (возможно, в силу исторических причин) — впрочем, это возможность переписать чужой код.

Содержание

- *Внешний вид кода*

  - *Отступы*
  - *Табуляция или пробелы?*
  - *Максимальная длина строки*
  - *Пустые строки*
  - *Кодировка исходного файла*
  - *Импорты*
- *Пробелы в выражениях и инструкциях*
  - *Избегайте использования пробелов в следующих ситуациях:*
  - *Другие рекомендации*
- *Комментарии*
  - *Блоки комментариев*
  - *“Встрочные” комментарии*
  - *Строки документации*
- *Контроль версий*
- *Соглашения по именованию*
  - *Главный принцип*
  - *Описание: Стили имен*
  - *Предписания: соглашения по именованию*
    * *Имена, которых следует избегать*
    * *Имена модулей и пакетов*
    * *Имена классов*
    * *Имена исключений*
    * *Имена глобальных переменных*
    * *Имена функций*
    * *Аргументы функций и методов*
    * *Имена методов и переменных экземпляров классов*
    * *Константы*
    * *Проектирование наследования*
- *Общие рекомендации*

Внешний вид кода

Отступы

Используйте 4 пробела на каждый уровень отступа.

Продолжительные строки должны выравнивать обернутые элементы либо вертикально, используя неявную линию в скобках (круглых, квадратных или фигурных), либо с использованием висячего отступа. При использовании висячего отступа следует применять следующие соображения: на первой линии не должно быть аргументов, а остальные строки должны четко восприниматься как продолжение линии.

Правильно:

```
# Выровнено по открывающему разделителю
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Больше отступов включено для отличения его от остальных
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

Неправильно:

```
# Аргументы на первой линии запрещены, если не используется вертикальное␣
↪выравнивание
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# Больше отступов требуется, для отличения его от остальных
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Опционально:

```
# Нет необходимости в большем количестве отступов.
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

Закрывающие круглые/квадратные/фигурные скобки в многострочных конструкциях могут находиться под первым непробельным символом последней строки списка, например:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
```

```
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

либо быть под первым символом строки, начинающей многострочную конструкцию:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

Табуляция или пробелы?

Пробелы - самый предпочтительный метод отступов.

Табуляция должна использоваться только для поддержки кода, написанного с отступами с помощью табуляции.

Python 3 запрещает смешивание табуляции и пробелов в отступах.

Python 2 пытается преобразовать табуляцию в пробелы.

Когда вы вызываете интерпретатор Python 2 в командной строке с параметром -t, он выдает предупреждения (warnings) при использовании смешанного стиля в отступах, а запустив интерпретатор с параметром -tt, вы получите в этих местах ошибки (errors). Эти параметры очень рекомендуются!

Максимальная длина строки

Ограничьте длину строки максимум 79 символами.

Для более длинных блоков текста с меньшими структурными ограничениями (строки документации или комментарии), длину строки следует ограничить 72 символами.

Ограничение необходимой ширины окна редактора позволяет иметь несколько открытых файлов бок о бок, и хорошо работает при использовании инструментов анализа кода, которые предоставляют две версии в соседних столбцах.

Некоторые команды предпочитают большую длину строки. Для кода, поддерживающегося исключительно или преимущественно этой группой, в которой могут прийти к согласию по этому вопросу, нормально увеличение длины строки с 80 до 100 символов (фактически увеличивая максимальную длину до 99 символов), при условии, что комментарии и строки документации все еще будут 72 символа.

Стандартная библиотека Python консервативна и требует ограничения длины строки в 79 символов (а строк документации/комментариев в 72).

Предпочтительный способ переноса длинных строк является использование подразумеваемых продолжений строк Python внутри круглых, квадратных и фигурных скобок. Длинные строки могут быть разбиты на несколько строк, обернутые в скобки. Это предпочтительнее использования обратной косой черты для продолжения строки.

Обратная косая черта все еще может быть использована время от времени. Например, длинная конструкция with не может использовать неявные продолжения, так что обратная косая черта является приемлемой:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
        open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

Ещё один случай - assert.

Сделайте правильные отступы для перенесённой строки. Предпочтительнее вставить перенос строки после логического оператора, но не перед ним. Например:

```
class Rectangle(Blob):

    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if (width == 0 and height == 0 and
                color == 'red' and emphasis == 'strong' or
                highlight > 100):
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                             (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

Пустые строки

Отделяйте функции верхнего уровня и определения классов двумя пустыми строками.

Определения методов внутри класса разделяются одной пустой строкой.

Дополнительные пустые строки возможно использовать для разделения различных групп похожих функций. Пустые строки могут быть опущены между несколькими связанными однострочниками (например, набор фиктивных реализаций).

Используйте пустые строки в функциях, чтобы указать логические разделы.

Python расценивает символ control+L как незначащий (whitespace), и вы можете использовать его, потому что многие редакторы обрабатывают его как разрыв страницы — таким образом логические части в файле будут на разных страницах. Однако, не все редакторы распознают control+L и могут на его месте отображать другой символ.

Кодировка исходного файла

Кодировка Python должна быть UTF-8 (ASCII в Python 2).

Файлы в ASCII (Python 2) или UTF-8 (Python 3) не должны иметь объявления кодировки.

В стандартной библиотеке, нестандартные кодировки должны использоваться только для целей тестирования, либо когда комментарий или строка документации требует упомянуть имя автора, содержащего не ASCII символы; в остальных случаях использование \x, \u, \U или \N - наиболее предпочтительный способ включить не ASCII символы в строковых литералах.

Начиная с версии python 3.0 в стандартной библиотеке действует следующее соглашение: все идентификаторы обязаны содержать только ASCII символы, и означать английские слова везде, где это возможно (во многих случаях используются сокращения или неанглийские технические термины). Кроме того, строки и комментарии тоже должны содержать лишь ASCII символы. Исключения составляют: (а) test case, тестирующий не-ASCII особенности программы, и (б) имена авторов. Авторы, чьи имена основаны не на латинском алфавите, должны транслитерировать свои имена в латиницу.

Проектам с открытым кодом для широкой аудитории также рекомендуется использовать это соглашение.

Импорты

- Каждый импорт, как правило, должен быть на отдельной строке.

  Правильно:

  ```
  import os
  import sys
  ```

  Неправильно:

  ```
  import sys, os
  ```

  В то же время, можно писать так:

  ```
  from subprocess import Popen, PIPE
  ```

- Импорты всегда помещаются в начале файла, сразу после комментариев к модулю и строк документации, и перед объявлением констант.

  Импорты должны быть сгруппированы в следующем порядке:

  1. импорты из стандартной библиотеки
  2. импорты сторонних библиотек
  3. импорты модулей текущего проекта

  Вставляйте пустую строку между каждой группой импортов.

  Указывайте спецификации __all__ после импортов.

- Рекомендуется абсолютное импортирование, так как оно обычно более читаемо и ведет себя лучше (или, по крайней мере, даёт понятные сообщения об ошибках) если импортируемая система настроена неправильно (например, когда каталог внутри пакета заканчивается на sys.path):

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

  Тем не менее, явный относительный импорт является приемлемой альтернативой абсолютному импорту, особенно при работе со сложными пакетами, где использование абсолютного импорта было бы излишне подробным:

```
from . import sibling
from .sibling import example
```

  В стандартной библиотеке следует избегать сложной структуры пакетов и всегда использовать абсолютные импорты.

  Неявные относительно импорты никогда не должны быть использованы, и были удалены в Python 3.

- Когда вы импортируете класс из модуля, вполне можно писать вот так:

```
from myclass import MyClass
from foo.bar.yourclass import YourClass
```

  Если такое написание вызывает конфликт имен, тогда пишите:

```
import myclass
import foo.bar.yourclass
```

  И используйте "myclass.MyClass" и "foo.bar.yourclass.YourClass".

- Шаблоны импортов (from import *) следует избегать, так как они делают неясным то, какие имена присутствуют в глобальном пространстве имён, что вводит в заблуждение как читателей, так и многие автоматизированные средства. Существует один оправданный пример использования шаблона импорта, который заключается в опубликовании внутреннего интерфейса как часть общественного API (например, переписав реализацию на чистом Python в модуле акселератора (и не будет заранее известно, какие именно функции будут перезаписаны).

Пробелы в выражениях и инструкциях

Избегайте использования пробелов в следующих ситуациях:

- Непосредственно внутри круглых, квадратных или фигурных скобок.

  Правильно:

```
spam(ham[1], {eggs: 2})
```

Неправильно:

```
spam( ham[ 1 ], { eggs: 2 } )
```

- Непосредственно перед запятой, точкой с запятой или двоеточием:

  Правильно:

```
if x == 4: print(x, y); x, y = y, x
```

Неправильно:

```
if x == 4 : print(x , y) ; x , y = y , x
```

- Сразу перед открывающей скобкой, после которой начинается список аргументов при вызове функции:

  Правильно:

```
spam(1)
```

Неправильно:

```
spam (1)
```

- Сразу перед открывающей скобкой, после которой следует индекс или срез:

  Правильно:

```
dict['key'] = list[index]
```

Неправильно:

```
dict ['key'] = list [index]
```

- Использование более одного пробела вокруг оператора присваивания (или любого другого) для того, чтобы выровнять его с другим:

  Правильно:

```
x = 1
y = 2
long_variable = 3
```

Неправильно:

```
x             = 1
y             = 2
long_variable = 3
```

Другие рекомендации

- Всегда окружайте эти бинарные операторы одним пробелом с каждой стороны: присваивания (=, +=, -= и другие), сравнения (==, <, >, !=, <>, <=, >=, in, not in, is, is not), логические (and, or, not).
- Если используются операторы с разными приоритетами, попробуйте добавить пробелы вокруг операторов с самым низким приоритетом. Используйте свои собственные суждения, однако, никогда не используйте более одного пробела, и всегда используйте одинаковое количество пробелов по обе стороны бинарного оператора.

  Правильно:

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

  Неправильно:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

- Не используйте пробелы вокруг знака =, если он используется для обозначения именованного аргумента или значения параметров по умолчанию.

  Правильно:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

  Неправильно:

```
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

- Не используйте составные инструкции (несколько команд в одной строке).

  Правильно:

```
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

  Неправильно:

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

- Иногда можно писать тело циклов while, for или ветку if в той же строке, если команда короткая, но если команд несколько, никогда так не пишите. А также избегайте длинных строк!

  Точно неправильно:

```
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

  Вероятно, неправильно:

```
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                             list, like, this)

if foo == 'blah': one(); two(); three()
```

Комментарии

Комментарии, противоречащие коду, хуже, чем отсутствие комментариев. Всегда исправляйте комментарии, если меняете код!

Комментарии должны являться законченными предложениями. Если комментарий — фраза или предложение, первое слово должно быть написано с большой буквы, если только это не имя переменной, которая начинается с маленькой буквы (никогда не изменяйте регистр переменной!).

Если комментарий короткий, можно опустить точку в конце предложения. Блок комментариев обычно состоит из одного или более абзацев, составленных из полноценных предложений, поэтому каждое предложение должно оканчиваться точкой.

Ставьте два пробела после точки в конце предложения.

Программисты, которые не говорят на английском языке, пожалуйста, пишите комментарии на английском, если только вы не уверены на 120%, что ваш код никогда не будут читать люди, не знающие вашего родного языка.

Блоки комментариев

Блок комментариев обычно объясняет код (весь, или только некоторую часть), идущий после блока, и должен иметь тот же отступ, что и сам код. Каждая строчка такого блока должна начинаться с символа # и одного пробела после него (если только сам текст комментария не имеет отступа).

Абзацы внутри блока комментариев разделяются строкой, состоящей из одного символа #.

“Встрочные” комментарии

Старайтесь реже использовать подобные комментарии.

Такой комментарий находится в той же строке, что и инструкция. “Встрочные” комментарии должны отделяться по крайней мере двумя пробелами от инструкции. Они должны начинаться с символа # и одного пробела.

Комментарии в строке с кодом не нужны и только отвлекают от чтения, если они объясняют очевидное. Не пишите вот так:

```
x = x + 1                 # Increment x
```

Впрочем, такие комментарии иногда полезны:

```
x = x + 1                 # Компенсация границы
```

Строки документации

- Пишите документацию для всех публичных модулей, функций, классов, методов. Строки документации необязательны для приватных методов, но лучше написать, что делает метод. Комментарий нужно писать после строки с def.
- PEP 257 объясняет, как правильно и хорошо документировать. Заметьте, очень важно, чтобы закрывающие кавычки стояли на отдельной строке. А еще лучше, если перед ними будет ещё и пустая строка, например:

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.

"""
```

- Для однострочной документации можно оставить закрывающие кавычки на той же строке.

Контроль версий

Если вам нужно использовать Subversion, CVS или RCS в ваших исходных кодах, делайте вот так:

```
__version__ = "$Revision: 1a40d4eaa00b $"
# $Source$
```

Вставляйте эти строки после документации модуля перед любым другим кодом и отделяйте их пустыми строками по одной до и после.

Соглашения по именованию

Соглашения по именованию переменных в python немного туманны, поэтому их список никогда не будет полным — тем не менее, ниже мы приводим список рекомендаций, действующих на данный момент. Новые модули и пакеты должны быть написаны согласно этим стандартам, но если в какой-либо уже существующей библиотеке эти правила нарушаются, предпочтительнее писать в едином с ней стиле.

Главный принцип

Имена, которые видны пользователю как часть общественного API должны следовать конвенциям, которые отражают использование, а не реализацию.

Описание: Стили имен

Существует много разных стилей. Поможем вам распознать, какой стиль именования используется, независимо от того, для чего он используется.

Обычно различают следующие стили:

- b (одиночная маленькая буква)
- B (одиночная заглавная буква)
- lowercase (слово в нижнем регистре)
- lower_case_with_underscores (слова из маленьких букв с подчеркиваниями)
- UPPERCASE (заглавные буквы)
- UPPERCASE_WITH_UNDERSCORES (слова из заглавных букв с подчеркиваниями)
- CapitalizedWords (слова с заглавными буквами, или CapWords, или CamelCase). Замечание: когда вы используете аббревиатуры в таком стиле, пишите все буквы аббревиатуры заглавными — HTTPServerError лучше, чем HttpServerError.
- mixedCase (отличается от CapitalizedWords тем, что первое слово начинается с маленькой буквы)

- Capitalized_Words_With_Underscores (слова с заглавными буквами и подчеркиваниями — уродливо!)

Ещё существует стиль, в котором имена, принадлежащие одной логической группе, имеют один короткий префикс. Этот стиль редко используется в python, но мы упоминаем его для полноты. Например, функция os.stat() возвращает кортеж, имена в котором традиционно имеют вид st_mode, st_size, st_mtime и так далее. (Так сделано, чтобы подчеркнуть соответствие этих полей структуре системных вызовов POSIX, что помогает знакомым с ней программистам).

В библиотеке X11 используется префикс X для всех public-функций. В python этот стиль считается излишним, потому что перед полями и именами методов стоит имя объекта, а перед именами функций стоит имя модуля.

В дополнение к этому, используются следующие специальные формы записи имен с добавлением символа подчеркивания в начало или конец имени:

- _single_leading_underscore: слабый индикатор того, что имя используется для внутренних нужд. Например, from M import * не будет импортировать объекты, чьи имена начинаются с символа подчеркивания.
- single_trailing_underscore_: используется по соглашению для избежания конфликтов с ключевыми словами языка python, например:

```
Tkinter.Toplevel(master, class_='ClassName')
```

- __double_leading_underscore: изменяет имя атрибута класса, то есть в классе FooBar поле __boo становится _FooBar__boo.
- __double_leading_and_trailing_underscore__ (двойное подчеркивание в начале и в конце имени): магические методы или атрибуты, которые находятся в пространствах имен, управляемых пользователем. Например, __init__, __import__ или __file__. Не изобретайте такие имена, используйте их только так, как написано в документации.

Предписания: соглашения по именованию

Имена, которых следует избегать

Никогда не используйте символы l (маленькая латинская буква «эль»), O (заглавная латинская буква «о») или I (заглавная латинская буква «ай») как однобуквенные идентификаторы.

В некоторых шрифтах эти символы неотличимы от цифры один и нуля. Если очень нужно l, пишите вместо неё заглавную L.

Имена модулей и пакетов

Модули должны иметь короткие имена, состоящие из маленьких букв. Можно использовать символы подчеркивания, если это улучшает читабельность. То же самое относится

и к именам пакетов, однако в именах пакетов не рекомендуется использовать символ подчёркивания.

Так как имена модулей отображаются в имена файлов, а некоторые файловые системы являются нечувствительными к регистру символов и обрезают длинные имена, очень важно использовать достаточно короткие имена модулей — это не проблема в Unix, но, возможно, код окажется непереносимым в старые версии Windows, Mac, или DOS.

Когда модуль расширения, написанный на С или C++, имеет сопутствующий python-модуль (содержащий интерфейс высокого уровня), C/C++ модуль начинается с символа подчеркивания, например, _socket.

Имена классов

Имена классов должны обычно следовать соглашению CapWords.

Вместо этого могут использоваться соглашения для именования функций, если интерфейс документирован и используется в основном как функции.

Обратите внимание, что существуют отдельные соглашения о встроенных именах: большинство встроенных имен - одно слово (либо два слитно написанных слова), а соглашение CapWords используется только для именования исключений и встроенных констант.

Имена исключений

Так как исключения являются классами, к исключениями применяется стиль именования классов. Однако вы можете добавить Error в конце имени (если, конечно, исключение действительно является ошибкой).

Имена глобальных переменных

Будем надеяться, что глобальные переменные используются только внутри одного модуля. Руководствуйтесь теми же соглашениями, что и для имен функций.

Добавляйте в модули, которые написаны так, чтобы их использовали с помощью from M import *, механизм __all__, чтобы предотвратить экспортирование глобальных переменных. Или же, используйте старое соглашение, добавляя перед именами таких глобальных переменных один символ подчеркивания (которым вы можете обозначить те глобальные переменные, которые используются только внутри модуля).

Имена функций

Имена функций должны состоять из маленьких букв, а слова разделяться символами подчеркивания — это необходимо, чтобы увеличить читабельность.

Стиль mixedCase допускается в тех местах, где уже преобладает такой стиль, для сохранения обратной совместимости.

Аргументы функций и методов

Всегда используйте self в качестве первого аргумента метода экземпляра объекта.

Всегда используйте cls в качестве первого аргумента метода класса.

Если имя аргумента конфликтует с зарезервированным ключевым словом python, обычно лучше добавить в конец имени символ подчеркивания, чем исказить написание слова или использовать аббревиатуру. Таким образом, class_ лучше, чем clss. (Возможно, хорошим вариантом будет подобрать синоним).

Имена методов и переменных экземпляров классов

Используйте тот же стиль, что и для имен функций: имена должны состоять из маленьких букв, а слова разделяться символами подчеркивания.

Используйте один символ подчёркивания перед именем для непубличных методов и атрибутов.

Чтобы избежать конфликтов имен с подклассами, используйте два ведущих подчеркивания.

Python искажает эти имена: если класс Foo имеет атрибут с именем __a, он не может быть доступен как Foo.__a. (Настойчивый пользователь все еще может получить доступ, вызвав Foo._Foo__a.) Вообще, два ведущих подчеркивания должны использоваться только для того, чтобы избежать конфликтов имен с атрибутами классов, предназначенных для наследования.

Примечание: есть некоторые разногласия по поводу использования __ имена (см. ниже).

Константы

Константы обычно объявляются на уровне модуля и записываются только заглавными буквами, а слова разделяются символами подчеркивания. Например: MAX_OVERFLOW, TOTAL.

Проектирование наследования

Обязательно решите, каким должен быть метод класса или экземпляра класса (далее - атрибут) — публичный или непубличный. Если вы сомневаетесь, выберите непубличный атрибут. Потом будет проще сделать его публичным, чем наоборот.

Публичные атрибуты — это те, которые будут использовать другие программисты, и вы должны быть уверены в отсутствии обратной несовместимости. Непубличные атрибуты, в свою очередь, не предназначены для использования третьими лицами, поэтому вы можете не гарантировать, что не измените или не удалите их.

Мы не используем термин "приватный атрибут", потому что на самом деле в python таких не бывает.

Другой тип атрибутов классов принадлежит так называемому API подклассов (в других языках они часто называются protected). Некоторые классы проектируются так, чтобы от них наследовали другие классы, которые расширяют или модифицируют поведение базового класса. Когда вы проектируете такой класс, решите и явно укажите, какие атрибуты являются публичными, какие принадлежат API подклассов, а какие используются только базовым классом.

Теперь сформулируем рекомендации:

- Открытые атрибуты не должны иметь в начале имени символа подчеркивания.
- Если имя открытого атрибута конфликтует с ключевым словом языка, добавьте в конец имени один символ подчеркивания. Это более предпочтительно, чем аббревиатура или искажение написания (однако, у этого правила есть исключение — аргумента, который означает класс, и особенно первый аргумент метода класса (class method) должен иметь имя cls).
- Назовите простые публичные атрибуты понятными именами и не пишите сложные методы доступа и изменения (accessor/mutator, get/set, — прим. перев.) Помните, что в python очень легко добавить их потом, если потребуется. В этом случае используйте свойства (properties), чтобы скрыть функциональную реализацию за синтаксисом доступа к атрибутам.

  Примечание 1: Свойства (properties) работают только в классах нового стиля (в Python 3 все классы являются таковыми).

  Примечание 2: Постарайтесь избавиться от побочных эффектов, связанным с функциональным поведением; впрочем, такие вещи, как кэширование, вполне допустимы.

  Примечание 3: Избегайте использования вычислительно затратных операций, потому что из-за записи с помощью атрибутов создается впечатление, что доступ происходит (относительно) быстро.
- Если вы планируете класс таким образом, чтобы от него наследовались другие классы, но не хотите, чтобы подклассы унаследовали некоторые атрибуты, добавьте в имена два символа подчеркивания в начало, и ни одного — в конец. Механизм изменения имен в python сработает так, что имя класса добавится к имени такого атрибута, что позволит избежать конфликта имен с атрибутами подклассов.

  Примечание 1: Будьте внимательны: если подкласс будет иметь то же имя класса и имя атрибута, то вновь возникнет конфликт имен.

  Примечание 2: Механизм изменения имен может затруднить отладку или работу с __getattr__(), однако он хорошо документирован и легко реализуется вручную.

  Примечание 3: Не всем нравится этот механизм, поэтому старайтесь достичь компромисса между необходимостью избежать конфликта имен и возможностью доступа к этим атрибутам.

Общие рекомендации

- Код должен быть написан так, чтобы не зависеть от разных реализаций языка (PyPy, Jython, IronPython, Pyrex, Psyco и пр.).

  Например, не полагайтесь на эффективную реализацию в CPython конкатенации строк в выражениях типа a+=b или a=a+b. Такие инструкции выполняются значительно медленнее в Jython. В критичных к времени выполнения частях программы используйте ''.join() — таким образом склеивание строк будет выполнено за линейное время независимо от реализации python.

- Сравнения с None должны обязательно выполняться с использованием операторов is или is not, а не с помощью операторов сравнения. Кроме того, не пишите if x, если имеете в виду if x is not None — если, к примеру, при тестировании такая переменная может принять значение другого типа, отличного от None, но при приведении типов может получиться False!

- При реализации методов сравнения, лучше всего реализовать все 6 операций сравнения (__eq__, __ne__, __lt__, __le__, __gt__, __ge__), чем полагаться на то, что другие программисты будут использовать только конкретный вид сравнения.

  Для минимизации усилий можно воспользоваться декоратором functools.total_ordering() для реализации недостающих методов.

  PEP 207 указывает, что интерпретатор может поменять y > x на x < y, y >= x на x <= y, и может поменять местами аргументы x == y и x != y. Гарантируется, что операции sort() и min() используют оператор <, а max() использует оператор >. Однако, лучше всего осуществить все шесть операций, чтобы не возникало путаницы в других местах.

- Всегда используйте выражение def, а не присваивание лямбда-выражения к имени.

  Правильно:

  ```
  def f(x): return 2*x
  ```

  Неправильно:

  ```
  f = lambda x: 2*x
  ```

- Наследуйте свой класс исключения от Exception, а не от BaseException. Прямое наследование от BaseException зарезервировано для исключений, которые не следует перехватывать.

- Используйте цепочки исключений соответствующим образом. В Python 3, “raise X from Y” следует использовать для указания явной замены без потери отладочной информации.

  Когда намеренно заменяется исключение (использование “raise X” в Python 2 или “raise X from None” в Python 3.3+), проследите, чтобы соответствующая информация передалась в новое исключение (такие, как сохранение имени атрибута при преобразовании KeyError в AttributeError или вложение текста исходного исключения в новом).

- Когда вы генерируете исключение, пишите raise ValueError('message') вместо старого синтаксиса raise ValueError, message.

  Старая форма записи запрещена в python 3.

  Такое использование предпочтительнее, потому что из-за скобок не нужно использовать символы для продолжения перенесенных строк, если эти строки длинные или если используется форматирование.

- Когда код перехватывает исключения, перехватывайте конкретные ошибки вместо простого выражения except:.

  К примеру, пишите вот так:

```
try:
    import platform_specific_module
except ImportError:
    platform_specific_module = None
```

  Простое написание "except:" также перехватит и SystemExit, и KeyboardInterrupt, что породит проблемы, например, сложнее будет завершить программу нажатием control+C. Если вы действительно собираетесь перехватить все исключения, пишите "except Exception:".

  Хорошим правилом является ограничение использования "except:", кроме двух случаев:

  1. Если обработчик выводит пользователю всё о случившейся ошибке; по крайней мере, пользователь будет знать, что произошла ошибка.
  2. Если нужно выполнить некоторый код после перехвата исключения, а потом вновь "бросить" его для обработки где-то в другом месте. Обычно же лучше пользоваться конструкцией "try…finally".

- При связывании перехваченных исключений с именем, предпочитайте явный синтаксис привязки, добавленный в Python 2.6:

```
try:
    process_data()
except Exception as exc:
    raise DataProcessingFailedError(str(exc))
```

  Это единственный синтаксис, поддерживающийся в Python 3, который позволяет избежать проблем неоднозначности, связанных с более старым синтаксисом на основе запятой.

- При перехвате ошибок операционной системы, предпочитайте использовать явную иерархию исключений, введенную в Python 3.3, вместо анализа значений errno.

- Постарайтесь заключать в каждую конструкцию try…except минимум кода, чтобы легче отлавливать ошибки. Опять же, это позволяет избежать замаскированных ошибок.

  Правильно:

```
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
else:
    return handle_value(value)
```

Неправильно:

```
try:
    # Здесь много действий!
    return handle_value(collection[key])
except KeyError:
    # Здесь также перехватится KeyError, который может быть сгенерирован␣
↪handle_value()
    return key_not_found(key)
```

- Когда ресурс является локальным на участке кода, используйте выражение with для того, чтобы после выполнения он был очищен оперативно и надёжно.
- Менеджеры контекста следует вызывать с помощью отдельной функции или метода, всякий раз, когда они делают что-то другое, чем получение и освобождение ресурсов. Например:

  Правильно:

```
with conn.begin_transaction():
    do_stuff_in_transaction(conn)
```

  Неправильно:

```
with conn:
    do_stuff_in_transaction(conn)
```

  Последний пример не дает никакой информации, указывающей на то, что __enter__ и __exit__ делают что-то кроме закрытия соединения после транзакции. Быть явным важно в данном случае.

- Используйте строковые методы вместо модуля string — они всегда быстрее и имеют тот же API для unicode-строк. Можно отказаться от этого правила, если необходима совместимость с версиями python младше 2.0.

  В Python 3 остались только строковые методы.

- Пользуйтесь ''.startswith() и ''.endswith() вместо обработки срезов строк для проверки суффиксов или префиксов.

  startswith() и endswith() выглядят чище и порождают меньше ошибок. Например:

  Правильно:

```
if foo.startswith('bar'):
```

Неправильно:

```
if foo[:3] == 'bar':
```

- Сравнение типов объектов нужно делать с помощью isinstance(), а не прямым сравнением типов:

  Правильно:

  ```
  if isinstance(obj, int):
  ```

  Неправильно:

  ```
  if type(obj) is type(1):
  ```

  Когда вы проверяете, является ли объект строкой, обратите внимание на то, что строка может быть unicode-строкой. В python 2 у str и unicode есть общий базовый класс, поэтому вы можете написать:

  ```
  if isinstance(obj, basestring):
  ```

  Отметим, что в Python 3, unicode и basestring больше не существуют (есть только str) и bytes больше не является своего рода строкой (это последовательность целых чисел).

- Для последовательностей (строк, списков, кортежей) используйте тот факт, что пустая последовательность есть false:

  Правильно:

  ```
  if not seq:
  if seq:
  ```

  Неправильно:

  ```
  if len(seq)
  if not len(seq)
  ```

- Не пользуйтесь строковыми константами, которые имеют важные пробелы в конце — они невидимы, а многие редакторы (а теперь и reindent.py) обрезают их.

- Не сравнивайте логические типы с True и False с помощью ==:

  Правильно:

  ```
  if greeting:
  ```

  Неправильно:

  ```
  if greeting == True:
  ```

  Совсем неправильно:

```
if greeting is True:
```

Глава 26

Документирование кода в Python. PEP 257

Документирование кода в python - достаточно важный аспект, ведь от нее порой зависит читаемость и быстрота понимания вашего кода, как другими людьми, так и вами через полгода.

PEP 257 описывает соглашения, связанные со строками документации python, рассказывает о том, как нужно документировать python код.

Цель этого PEP - стандартизировать структуру строк документации: что они должны в себя включать, и как это написать (не касаясь вопроса синтаксиса строк документации). Этот PEP описывает соглашения, а не правила или синтаксис.

При нарушении этих соглашений, самое худшее, чего можно ожидать - некоторых неодобрительных взглядов. Но некоторые программы (например, docutils), знают о соглашениях, поэтому следование им даст вам лучшие результаты.

Что такое строки документации?

Строки документации - строковые литералы, которые являются первым оператором в модуле, функции, классе или определении метода. Такая строка документации становится специальным атрибутом __doc__ этого объекта.

Все модули должны, как правило, иметь строки документации, и все функции и классы, экспортируемые модулем также должны иметь строки документации. Публичные методы (в том числе __init__) также должны иметь строки документации. Пакет модулей может быть документирован в __init__.py.

Для согласованности, всегда используйте “”“triple double quotes”“” для строк документации. Используйте r”“”raw triple double quotes”“”, если вы будете использовать обратную косую черту в строке документации.

Существует две формы строк документации: однострочная и многострочная.

Однострочные строки документации

Однострочники предназначены для действительно очевидных случаев. Они должны умещаться на одной строке. Например:

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
```

Используйте тройные кавычки, даже если документация умещается на одной строке. Потом будет проще её дополнить.

Закрывающие кавычки на той же строке. Это смотрится лучше.

Нет пустых строк перед или после документации.

Однострочная строка документации не должна быть "подписью" параметров функции / метода (которые могут быть получены с помощью интроспекции). Не делайте:

```
def function(a, b):
    """function(a, b) -> list"""
```

Этот тип строк документации подходит только для C функций (таких, как встроенные модули), где интроспекция не представляется возможной. Тем не менее, возвращаемое значение не может быть определено путем интроспекции. Предпочтительный вариант для такой строки документации будет что-то вроде:

```
def function(a, b):
    """Do X and return a list."""
```

(Конечно, "Do X" следует заменить полезным описанием!)

Многострочные строки документации

Многострочные строки документации состоят из однострочной строки документации с последующей пустой строкой, а затем более подробным описанием. Первая строка может быть использована автоматическими средствами индексации, поэтому важно, чтобы она находилась на одной строке и была отделена от остальной документации пустой строкой. Первая строка может быть на той же строке, где и открывающие кавычки, или на следующей строке. Вся документация должна иметь такой же отступ, как кавычки на первой строке (см. пример ниже).

Вставляйте пустую строку до и после всех строк документации (однострочных или многострочных), которые документируют класс - вообще говоря, методы класса разделены друг от друга одной пустой строкой, а строка документации должна быть смещена от первого

метода пустой строкой; для симметрии, поставьте пустую строку между заголовком класса и строкой документации. Строки документации функций и методов, как правило, не имеют этого требования.

Строки документации скрипта (самостоятельной программы) должны быть доступны в качестве "сообщения по использованию", напечатанной, когда программа вызывается с некорректными или отсутствующими аргументами (или, возможно, с опцией "-h", для помощи). Такая строка документации должна документировать функции программы и синтаксис командной строки, переменные окружения и файлы. Сообщение по использованию может быть довольно сложным (несколько экранов) и должно быть достаточным для нового пользователя для использования программы должным образом, а также полный справочник со всеми вариантами и аргументами для искушенного пользователя.

Строки документации модуля должны, как правило, перечислять классы, исключения, функции (и любые другие объекты), которые экспортируются модулем, с краткими пояснениями (в одну строчку) каждого из них. (Эти строки, как правило, дают меньше деталей, чем первая строка документации к объекту). Строки документации пакета модулей (т.е. строка документации в __init__.py) также должны включать модули и подпакеты.

Строки документации функции или метода должны обобщить его поведение и документировать свои аргументы, возвращаемые значения, побочные эффекты, исключения, дополнительные аргументы, именованные аргументы, и ограничения на вызов функции.

Строки документации класса обобщают его поведение и перечисляют открытые методы и переменные экземпляра. Если класс предназначен для подклассов, и имеет дополнительный интерфейс для подклассов, этот интерфейс должен быть указан отдельно (в строке документации). Конструктор класса должен быть задокументирован в документации метода __init__. Отдельные методы должны иметь свои строки документации.

Если класс - подкласс другого класса, и его поведение в основном унаследовано от этого класса, строки документации должны отмечать это и обобщить различия. Используйте глагол "override", чтобы указать, что метод подкласса заменяет метод суперкласса и не вызывает его; используйте глагол "extend", чтобы указать, что метод подкласса вызывает метод суперкласса (в дополнение к собственному поведению).

И, напоследок, пример:

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)

    """
    if imag == 0.0 and real == 0.0: return complex_zero
    ...
```

А ещё больше примеров можно посмотреть в стандартной библиотеке python (например, в папке Lib вашего интерпретатора python).

Глава 27

Работа с модулями: создание, подключение инструкциями import и from

Модулем в Python называется любой файл с программой (да-да, все те программы, которые вы писали, можно назвать модулями). В этой статье мы поговорим о том, как создать модуль, и как подключить модуль, из стандартной библиотеки или написанный вами.

Каждая программа может импортировать модуль и получить доступ к его классам, функциям и объектам. Нужно заметить, что модуль может быть написан не только на Python, а например, на C или C++.

Подключение модуля из стандартной библиотеки

Подключить модуль можно с помощью инструкции import. К примеру, подключим модуль os для получения текущей директории:

```
>>> import os
>>> os.getcwd()
'C:\\Python33'
```

После ключевого слова **import** указывается название модуля. Одной инструкцией можно подключить несколько модулей, хотя этого не рекомендуется делать, так как это снижает читаемость кода. Импортируем модули time и random.

```
>>> import time, random
>>> time.time()
1376047104.056417
>>> random.random()
0.9874550833306869
```

После импортирования модуля его название становится переменной, через которую можно получить доступ к атрибутам модуля. Например, можно обратиться к константе e, расположенной в модуле math:

```
>>> import math
>>> math.e
2.718281828459045
```

Стоит отметить, что если указанный атрибут модуля не будет найден, возбудится исключение AttributeError. А если не удастся найти модуль для импортирования, то ImportError.

```
>>> import notexist
Traceback (most recent call last):
  File "", line 1, in
    import notexist
ImportError: No module named 'notexist'
>>> import math
>>> math.Ё
Traceback (most recent call last):
  File "", line 1, in
    math.Ё
AttributeError: 'module' object has no attribute 'Ё'
```

Использование псевдонимов

Если название модуля слишком длинное, или оно вам не нравится по каким-то другим причинам, то для него можно создать псевдоним, с помощью ключевого слова as.

```
>>> import math as m
>>> m.e
2.718281828459045
```

Теперь доступ ко всем атрибутам модуля math осуществляется только с помощью переменной m, а переменной math в этой программе уже не будет (если, конечно, вы после этого не напишете import math, тогда модуль будет доступен как под именем m, так и под именем math).

Инструкция from

Подключить определенные атрибуты модуля можно с помощью инструкции from. Она имеет несколько форматов:

```
from <Название модуля> import <Атрибут 1> [ as <Псевдоним 1> ], [<Атрибут 2> [ as
↪<Псевдоним 2> ] ...]
from <Название модуля> import *
```

Первый формат позволяет подключить из модуля только указанные вами атрибуты. Для длинных имен также можно назначить псевдоним, указав его после ключевого слова as.

```
>>> from math import e, ceil as c
>>> e
2.718281828459045
>>> c(4.6)
5
```

Импортируемые атрибуты можно разместить на нескольких строках, если их много, для лучшей читаемости кода:

```
>>> from math import (sin, cos,
...                   tan, atan)
```

Второй формат инструкции from позволяет подключить все (точнее, почти все) переменные из модуля. Для примера импортируем все атрибуты из модуля sys:

```
>>> from sys import *
>>> version
'3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC v.1600 32 bit (Intel)]'
>>> version_info
sys.version_info(major=3, minor=3, micro=2, releaselevel='final', serial=0)
```

Следует заметить, что не все атрибуты будут импортированы. Если в модуле определена переменная __all__ (список атрибутов, которые могут быть подключены), то будут подключены только атрибуты из этого списка. Если переменная __all__ не определена, то будут подключены все атрибуты, не начинающиеся с нижнего подчёркивания. Кроме того, необходимо учитывать, что импортирование всех атрибутов из модуля может нарушить пространство имен главной программы, так как переменные, имеющие одинаковые имена, будут перезаписаны.

Создание своего модуля на Python

Теперь пришло время создать свой модуль. Создадим файл mymodule.py, в которой определим какие-нибудь функции:

```
def hello():
    print('Hello, world!')

def fib(n):
    a = b = 1
    for i in range(n - 2):
        a, b = b, a + b
    return b
```

Теперь в этой же папке создадим другой файл, например, main.py:

```
import mymodule

mymodule.hello()
print(mymodule.fib(10))
```

Выведет:

```
Hello, world!
55
```

Поздравляю! Вы **сделали свой модуль**! Напоследок отвечу ещё на пару вопросов, связанных с созданием модулей:

Как назвать модуль?

Помните, что вы (или другие люди) будут его импортировать и использовать в качестве переменной. Модуль нельзя именовать также, как и ключевое слово (их список можно посмотреть тут). Также имена модулей нельзя начинать с цифры. И не стоит называть модуль также, как какую-либо из встроенных функций. То есть, конечно, можно, но это создаст большие неудобства при его последующем использовании.

Куда поместить модуль?

Туда, где его потом можно будет найти. Пути поиска модулей указаны в переменной sys.path. В него включены текущая директория (то есть модуль можно оставить в папке с основной программой), а также директории, в которых установлен python. Кроме того, переменную sys.path можно изменять вручную, что позволяет положить модуль в любое удобное для вас место (главное, не забыть в главной программе модифицировать sys.path).

Можно ли использовать модуль как самостоятельную программу?

Можно. Однако надо помнить, что при импортировании модуля его код выполняется полностью, то есть, если программа что-то печатает, то при её импортировании это будет напечатано. Этого можно избежать, если проверять, запущен ли скрипт как программа, или импортирован. Это можно сделать с помощью переменной __name__, которая определена в любой программе, и равна "__main__", если скрипт запущен в качестве главной программы, и имя, если он импортирован. Например, mymodule.py может выглядеть вот так:

```
def hello():
    print('Hello, world!')

def fib(n):
    a = b = 1
    for i in range(n - 2):
```

```
        a, b = b, a + b
    return b

if __name__ == "__main__":
    hello()
    for i in range(10):
        print(fib(i))
```

Глава 28

Объектно-ориентированное программирование. Общее представление

Сегодня мы поговорим об объектно-ориентированном программировании и о его применении в python.

Объектно-ориентированное программирование (**ООП**) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

Класс — тип, описывающий устройство объектов. **Объект** — это экземпляр класса. Класс можно сравнить с чертежом, по которому создаются объекты.

Python соответствует принципам объектно-ориентированного программирования. В python всё является объектами - и строки, и списки, и словари, и всё остальное.

Но возможности ООП в python этим не ограничены. Программист может написать свой тип данных (класс), определить в нём свои методы.

Это не является обязательным - мы можем пользоваться только встроенными объектами. Однако ООП полезно при долгосрочной разработке программы несколькими людьми, так как упрощает понимание кода.

Приступим теперь собственно к написанию своих классов на python. Попробуем определить собственный класс:

```
>>> # Пример самого простейшего класса
... class A:
...     pass
```

Теперь мы можем создать несколько экземпляров этого класса:

```
>>> a = A()
>>> b = A()
>>> a.arg = 1  # у экземпляра a появился атрибут arg, равный 1
```

```
>>> b.arg = 2  # а у экземпляра b - атрибут arg, равный 2
>>> print(a.arg)
1
```

Классу возможно задать собственные методы:

```
>>> class A:
...     def g(self): # self - обязательный аргумент, содержащий в себе экземпляр
...                  # класса, передающийся при вызове метода,
...                  # поэтому этот аргумент должен присутствовать
...                  # во всех методах класса.
...         return 'hello world'
...
>>> a = A()
>>> a.g()
'hello world'
```

И напоследок еще один пример:

```
>>> class B:
...     arg = 'Python' # Все экземпляры этого класса будут иметь атрибут arg,
...                    # равный "Python"
...                    # Но впоследствии мы его можем изменить
...     def g(self):
...         return self.arg
...
>>> b = B()
>>> b.g()
'Python'
>>> B.g(b)
'Python'
>>> b.arg = 'spam'
>>> b.g()
'spam'
```

Глава 29

Инкапсуляция, наследование, полиморфизм

Недавно мы говорили об основах объектно-ориентированного программирования в python, теперь продолжим эту тему и поговорим о таких понятиях ООП, как **инкапсуляция**, **наследование** и **полиморфизм**.

Инкапсуляция

Инкапсуляция — ограничение доступа к составляющим объект компонентам (методам и переменным). Инкапсуляция делает некоторые из компонент доступными только внутри класса.

Инкапсуляция в Python работает лишь на уровне соглашения между программистами о том, какие атрибуты являются общедоступными, а какие — внутренними.

Одиночное подчеркивание в начале имени атрибута говорит о том, что переменная или метод не предназначен для использования вне методов класса, однако атрибут доступен по этому имени.

```
class A:
    def _private(self):
        print("Это приватный метод!")

>>> a = A()
>>> a._private()
Это приватный метод!
```

Двойное подчеркивание в начале имени атрибута даёт большую защиту: атрибут становится недоступным по этому имени.

```
>>> class B:
...     def __private(self):
...         print("Это приватный метод!")
...
>>> b = B()
>>> b.__private()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'B' object has no attribute '__private'
```

Однако полностью это не защищает, так как атрибут всё равно остаётся доступным под именем _ИмяКласса__ИмяАтрибута:

```
>>> b._B__private()
Это приватный метод!
```

Наследование

Наследование подразумевает то, что дочерний класс содержит все атрибуты родительского класса, при этом некоторые из них могут быть переопределены или добавлены в дочернем. Например, мы можем создать свой класс, похожий на словарь:

```
>>> class Mydict(dict):
...     def get(self, key, default = 0):
...         return dict.get(self, key, default)
...
>>> a = dict(a=1, b=2)
>>> b = Mydict(a=1, b=2)
```

Класс Mydict ведёт себя точно так же, как и словарь, за исключением того, что метод get по умолчанию возвращает не None, а 0.

```
>>> b['c'] = 4
>>> print(b)
{'a': 1, 'c': 4, 'b': 2}
>>> print(a.get('v'))
None
>>> print(b.get('v'))
0
```

Полиморфизм

Полиморфизм - разное поведение одного и того же метода в разных классах. Например, мы можем сложить два числа, и можем сложить две строки. При этом получим разный результат, так как числа и строки являются разными классами.

```
>>> 1 + 1
2
>>> "1" + "1"
'11'
```

Глава 30

Перегрузка операторов

Перегрузка операторов — один из способов реализации полиморфизма, когда мы можем задать свою реализацию какого-либо метода в своём классе.

Например, у нас есть два класса:

```
class A:
    def go(self):
        print('Go, A!')

class B(A):
    def go(self, name):
        print('Go, {}!'.format(name))
```

В данном примере класс B наследует класс A, но переопределяет метод go, поэтому он имеет мало общего с аналогичным методом класса A.

Однако в python имеются методы, которые, как правило, не вызываются напрямую, а вызываются встроенными функциями или операторами.

Например, метод **__init__** перегружает конструктор класса. Конструктор - создание экземпляра класса.

```
>>> class A:
...     def __init__(self, name):
...         self.name = name
...
>>> a = A('Vasya')
>>> print(a.name)
Vasya
```

Собственно, далее пойдёт список таких "магических" методов.

__new__(cls[, ...]) — управляет созданием экземпляра. В качестве обязательного аргумента принимает класс (не путать с экземпляром). Должен возвращать экземпляр класса для его последующей его передачи методу __init__.

__init__(self[, ...]) - как уже было сказано выше, конструктор.

__del__(self) - вызывается при удалении объекта сборщиком мусора.

__repr__(self) - вызывается встроенной функцией repr; возвращает "сырые" данные, использующиеся для внутреннего представления в python.

__str__(self) - вызывается функциями str, print и format. Возвращает строковое представление объекта.

__bytes__(self) - вызывается функцией bytes при преобразовании к байтам.

__format__(self, format_spec) - используется функцией format (а также методом format у строк).

__lt__(self, other) - x < y вызывает x.__lt__(y).

__le__(self, other) - x <= y вызывает x.__le__(y).

__eq__(self, other) - x == y вызывает x.__eq__(y).

__ne__(self, other) - x != y вызывает x.__ne__(y)

__gt__(self, other) - x > y вызывает x.__gt__(y).

__ge__(self, other) - x >= y вызывает x.__ge__(y).

__hash__(self) - получение хэш-суммы объекта, например, для добавления в словарь.

__bool__(self) - вызывается при проверке истинности. Если этот метод не определён, вызывается метод __len__ (объекты, имеющие ненулевую длину, считаются истинными).

__getattr__(self, name) - вызывается, когда атрибут экземпляра класса не найден в обычных местах (например, у экземпляра нет метода с таким названием).

__setattr__(self, name, value) - назначение атрибута.

__delattr__(self, name) - удаление атрибута (del obj.name).

__call__(self[, args...]) - вызов экземпляра класса как функции.

__len__(self) - длина объекта.

__getitem__(self, key) - доступ по индексу (или ключу).

__setitem__(self, key, value) - назначение элемента по индексу.

__delitem__(self, key) - удаление элемента по индексу.

__iter__(self) - возвращает итератор для контейнера.

__reversed__(self) - итератор из элементов, следующих в обратном порядке.

__contains__(self, item) - проверка на принадлежность элемента контейнеру (item in self).

Перегрузка арифметических операторов

__add__(self, other) - сложение. x + y вызывает x.__add__(y).

__sub__(self, other) - вычитание (x - y).

__mul__(self, other) - умножение (x * y).

__truediv__(self, other) - деление (x / y).

__floordiv__(self, other) - целочисленное деление (x // y).

__mod__(self, other) - остаток от деления (x % y).

__divmod__(self, other) - частное и остаток (divmod(x, y)).

__pow__(self, other[, modulo]) - возведение в степень (x ** y, pow(x, y[, modulo])).

__lshift__(self, other) - битовый сдвиг влево (x << y).

__rshift__(self, other) - битовый сдвиг вправо (x >> y).

__and__(self, other) - битовое И (x & y).

__xor__(self, other) - битовое ИСКЛЮЧАЮЩЕЕ ИЛИ (x ^ y).

__or__(self, other) - битовое ИЛИ (x | y).

Пойдём дальше.

__radd__(self, other),

__rsub__(self, other),

__rmul__(self, other),

__rtruediv__(self, other),

__rfloordiv__(self, other),

__rmod__(self, other),

__rdivmod__(self, other),

__rpow__(self, other),

__rlshift__(self, other),

__rrshift__(self, other),

__rand__(self, other),

__rxor__(self, other),

__ror__(self, other) - делают то же самое, что и арифметические операторы, перечисленные выше, но для аргументов, находящихся справа, и только в случае, если для левого операнда не определён соответствующий метод.

Например, операция x + y будет сначала пытаться вызвать x.__add__(y), и только в том случае, если это не получилось, будет пытаться вызвать y.__radd__(x). Аналогично для остальных методов.

Идём дальше.

__iadd__(self, other) - +=.

__isub__(self, other) - -=.

__imul__(self, other) - *=.

__itruediv__(self, other) - /=.

__ifloordiv__(self, other) - //=.

__imod__(self, other) - %=.

__ipow__(self, other[, modulo]) - **=.

__ilshift__(self, other) - <<=.

__irshift__(self, other) - >>=.

__iand__(self, other) - &=.

__ixor__(self, other) - ^=.

__ior__(self, other) - |=.

__neg__(self) - унарный -.

__pos__(self) - унарный +.

__abs__(self) - модуль (abs()).

__invert__(self) - инверсия (~).

__complex__(self) - приведение к complex.

__int__(self) - приведение к int.

__float__(self) - приведение к float.

__round__(self[, n]) - округление.

__enter__(self), **__exit__**(self, exc_type, exc_value, traceback) - реализация менеджеров контекста.

Рассмотрим некоторые из этих методов на примере двухмерного вектора, для которого переопределим некоторые методы:

```
import math

class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Vector2D({}, {})'.format(self.x, self.y)

    def __str__(self):
        return '({}, {})'.format(self.x, self.y)
```

```
    def __add__(self, other):
        return Vector2D(self.x + other.x, self.y + other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self

    def __sub__(self, other):
        return Vector2D(self.x - other.x, self.y - other.y)

    def __isub__(self, other):
        self.x -= other.x
        self.y -= other.y
        return self

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __bool__(self):
        return self.x != 0 or self.y != 0

    def __neg__(self):
        return Vector2D(-self.x, -self.y)

>>> x = Vector2D(3, 4)
>>> x
Vector2D(3, 4)
>>> print(x)
(3, 4)
>>> abs(x)
5.0
>>> y = Vector2D(5, 6)
>>> y
Vector2D(5, 6)
>>> x + y
Vector2D(8, 10)
>>> x - y
Vector2D(-2, -2)
>>> -x
Vector2D(-3, -4)
>>> x += y
>>> x
Vector2D(8, 10)
>>> bool(x)
True
>>> z = Vector2D(0, 0)
>>> bool(z)
False
>>> -z
```

```
Vector2D(0, 0)
```

В заключение хочу сказать, что перегрузка специальных методов - вещь хорошая, но не стоит ей слишком злоупотреблять. Перегружайте их только тогда, когда вы уверены в том, что это поможет пониманию программного кода.

Глава 31

Декораторы

Декораторы в Python и примеры их практического использования.

Итак, что же это такое? Для того, чтобы понять, как работают декораторы, в первую очередь следует вспомнить, что функции в python являются объектами, соответственно, их можно возвращать из другой функции или передавать в качестве аргумента. Также следует помнить, что функция в python может быть определена и внутри другой функции.

Вспомнив это, можно смело переходить к декораторам. **Декораторы** — это, по сути, “обёртки”, которые дают нам возможность изменить поведение функции, не изменяя её код.

Создадим свой декоратор “вручную”:

```
>>> def my_shiny_new_decorator(function_to_decorate):
...     # Внутри себя декоратор определяет функцию-"обёртку". Она будет обёрнута␣
↪вокруг декорируемой,
...     # получая возможность исполнять произвольный код до и после неё.
...     def the_wrapper_around_the_original_function():
...         print("Я - код, который отработает до вызова функции")
...         function_to_decorate() # Сама функция
...         print("А я - код, срабатывающий после")
...     # Вернём эту функцию
...     return the_wrapper_around_the_original_function
...
>>> # Представим теперь, что у нас есть функция, которую мы не планируем больше␣
↪трогать.
>>> def stand_alone_function():
...     print("Я простая одинокая функция, ты ведь не посмеешь меня изменять?")
...
>>> stand_alone_function()
Я простая одинокая функция, ты ведь не посмеешь меня изменять?
>>> # Однако, чтобы изменить её поведение, мы можем декорировать её, то есть␣
↪просто передать декоратору,
```

```
>>> # который обернет исходную функцию в любой код, который нам потребуется, и␣
↪вернёт новую,
>>> # готовую к использованию функцию:
>>> stand_alone_function_decorated = my_shiny_new_decorator(stand_alone_function)
>>> stand_alone_function_decorated()
Я - код, который отработает до вызова функции
Я простая одинокая функция, ты ведь не посмеешь меня изменять?
А я - код, срабатывающий после
```

Наверное, теперь мы бы хотели, чтобы каждый раз, во время вызова stand_alone_function, вместо неё вызывалась stand_alone_function_decorated. Для этого просто перезапишем stand_alone_function:

```
>>> stand_alone_function = my_shiny_new_decorator(stand_alone_function)
>>> stand_alone_function()
Я - код, который отработает до вызова функции
Я простая одинокая функция, ты ведь не посмеешь меня изменять?
А я - код, срабатывающий после
```

Собственно, это и есть декораторы. Вот так можно было записать предыдущий пример, используя синтаксис декораторов:

```
>>> @my_shiny_new_decorator
... def another_stand_alone_function():
...     print("Оставь меня в покое")
...
>>> another_stand_alone_function()
Я - код, который отработает до вызова функции
Оставь меня в покое
А я - код, срабатывающий после
```

То есть, декораторы в python — это просто синтаксический сахар для конструкций вида:

```
another_stand_alone_function = my_shiny_new_decorator(another_stand_alone_function)
```

При этом, естественно, можно использовать несколько декораторов для одной функции, например так:

```
>>> def bread(func):
...     def wrapper():
...         print()
...         func()
...         print("<\______/>")
...     return wrapper
...
>>> def ingredients(func):
...     def wrapper():
...         print("#помидоры#")
...         func()
...         print("~салат~")
...     return wrapper
```

```
...
>>> def sandwich(food="--ветчина--"):
...     print(food)
...
>>> sandwich()
--ветчина--
>>> sandwich = bread(ingredients(sandwich))
>>> sandwich()

#помидоры#
--ветчина--
~салат~
<\______/>
```

И используя синтаксис декораторов:

```
>>> @bread
... @ingredients
... def sandwich(food="--ветчина--"):
...     print(food)
...
>>> sandwich()

#помидоры#
--ветчина--
~салат~
<\______/>
```

Также нужно помнить о том, что важен порядок декорирования. Сравните с предыдущим примером:

```
>>> @ingredients
... @bread
... def sandwich(food="--ветчина--"):
...     print(food)
...
>>> sandwich()
#помидоры#

--ветчина--
<\______/>
~салат~
```

Передача декоратором аргументов в функцию

Однако, все декораторы, которые мы рассматривали, не имели одного очень важного функционала — передачи аргументов декорируемой функции. Собственно, это тоже несложно сделать.

```
>>> def a_decorator_passing_arguments(function_to_decorate):
...     def a_wrapper_accepting_arguments(arg1, arg2):
...         print("Смотри, что я получил:", arg1, arg2)
...         function_to_decorate(arg1, arg2)
...     return a_wrapper_accepting_arguments
...
>>> # Теперь, когда мы вызываем функцию, которую возвращает декоратор, мы вызываем␣
↪её "обёртку",
>>> # передаём ей аргументы и уже в свою очередь она передаёт их декорируемой␣
↪функции
>>> @a_decorator_passing_arguments
... def print_full_name(first_name, last_name):
...     print("Меня зовут", first_name, last_name)
...
>>> print_full_name("Vasya", "Pupkin")
Смотри, что я получил: Vasya Pupkin
Меня зовут Vasya Pupkin
```

Декорирование методов

Один из важных фактов, которые следует понимать, заключается в том, что функции и методы в Python — это практически одно и то же, за исключением того, что методы всегда ожидают первым параметром ссылку на сам объект (self). Это значит, что мы можем создавать декораторы для методов точно так же, как и для функций, просто не забывая про self.

```
>>> def method_friendly_decorator(method_to_decorate):
...     def wrapper(self, lie):
...         lie -= 3
...         return method_to_decorate(self, lie)
...     return wrapper
...
>>> class Lucy:
...     def __init__(self):
...         self.age = 32
...     @method_friendly_decorator
...     def sayYourAge(self, lie):
...         print("Мне {} лет, а ты бы сколько дал?".format(self.age + lie))
...
>>> l = Lucy()
>>> l.sayYourAge(-3)
Мне 26 лет, а ты бы сколько дал?
```

Конечно, если мы создаём максимально общий декоратор и хотим, чтобы его можно было применить к любой функции или методу, то можно воспользоваться распаковкой аргументов:

```
>>> def a_decorator_passing_arbitrary_arguments(function_to_decorate):
...     # Данная "обёртка" принимает любые аргументы
...     def a_wrapper_accepting_arbitrary_arguments(*args, **kwargs):
...         print("Передали ли мне что-нибудь?:")
...         print(args)
...         print(kwargs)
...         function_to_decorate(*args, **kwargs)
...     return a_wrapper_accepting_arbitrary_arguments
...
>>> @a_decorator_passing_arbitrary_arguments
... def function_with_no_argument():
...     print("Python is cool, no argument here.")
...
>>> function_with_no_argument()
Передали ли мне что-нибудь?:
()
{}
Python is cool, no argument here.
>>> @a_decorator_passing_arbitrary_arguments
... def function_with_arguments(a, b, c):
...     print(a, b, c)
...
>>> function_with_arguments(1, 2, 3)
Передали ли мне что-нибудь?:
(1, 2, 3)
{}
1 2 3
>>> @a_decorator_passing_arbitrary_arguments
... def function_with_named_arguments(a, b, c, platypus="Почему нет?"):
...     print("Любят ли {}, {} и {} утконосов? {}".format(a, b, c, platypus))
...
>>> function_with_named_arguments("Билл", "Линус", "Стив", platypus="Определенно!")
Передали ли мне что-нибудь?:
('Билл', 'Линус', 'Стив')
{'platypus': 'Определенно!'}
Любят ли Билл, Линус и Стив утконосов? Определенно!
>>> class Mary(object):
...     def __init__(self):
...         self.age = 31
...     @a_decorator_passing_arbitrary_arguments
...     def sayYourAge(self, lie=-3): # Теперь мы можем указать значение по умолчанию
...         print("Мне {} лет, а ты бы сколько дал?".format(self.age + lie))
...
>>> m = Mary()
>>> m.sayYourAge()
Передали ли мне что-нибудь?:
(<__main__.Mary object at 0x7f6373017780>,)
{}
Мне 28 лет, а ты бы сколько дал?
```

Декораторы с аргументами

А теперь попробуем написать декоратор, принимающий аргументы:

```
>>> def decorator_maker():
...     print("Я создаю декораторы! Я буду вызван только раз: когда ты попросишь␣
↪меня создать декоратор.")
...     def my_decorator(func):
...         print("Я - декоратор! Я буду вызван только раз: в момент декорирования␣
↪функции.")
...         def wrapped():
...             print ("Я - обёртка вокруг декорируемой функции.\n"
...                    "Я буду вызвана каждый раз, когда ты вызываешь декорируемую␣
↪функцию.\n"
...                    "Я возвращаю результат работы декорируемой функции.")
...             return func()
...         print("Я возвращаю обёрнутую функцию.")
...         return wrapped
...     print("Я возвращаю декоратор.")
...     return my_decorator
...
>>> # Давайте теперь создадим декоратор. Это всего лишь ещё один вызов функции
>>> new_decorator = decorator_maker()
Я создаю декораторы! Я буду вызван только раз: когда ты попросишь меня создать␣
↪декоратор.
Я возвращаю декоратор.
>>>
>>> # Теперь декорируем функцию
>>> def decorated_function():
...     print("Я - декорируемая функция.")
...
>>> decorated_function = new_decorator(decorated_function)
Я - декоратор! Я буду вызван только раз: в момент декорирования функции.
Я возвращаю обёрнутую функцию.
>>> # Теперь наконец вызовем функцию:
>>> decorated_function()
Я - обёртка вокруг декорируемой функции.
Я буду вызвана каждый раз, когда ты вызываешь декорируемую функцию.
Я возвращаю результат работы декорируемой функции.
Я - декорируемая функция.
```

Теперь перепишем данный код с помощью декораторов:

```
>>> @decorator_maker()
... def decorated_function():
...     print("Я - декорируемая функция.")
...
Я создаю декораторы! Я буду вызван только раз: когда ты попросишь меня создать␣
↪декоратор.
Я возвращаю декоратор.
Я - декоратор! Я буду вызван только раз: в момент декорирования функции.
```

```
Я возвращаю обёрнутую функцию.
>>> decorated_function()
Я - обёртка вокруг декорируемой функции.
Я буду вызвана каждый раз когда ты вызываешь декорируемую функцию.
Я возвращаю результат работы декорируемой функции.
Я - декорируемая функция.
```

Вернёмся к аргументам декораторов, ведь, если мы используем функцию, чтобы создавать декораторы "на лету", мы можем передавать ей любые аргументы, верно?

```
>>> def decorator_maker_with_arguments(decorator_arg1, decorator_arg2):
...     print("Я создаю декораторы! И я получил следующие аргументы:",
...            decorator_arg1, decorator_arg2)
...     def my_decorator(func):
...         print("Я - декоратор. И ты всё же смог передать мне эти аргументы:",
...                decorator_arg1, decorator_arg2)
...         # Не перепутайте аргументы декораторов с аргументами функций!
...         def wrapped(function_arg1, function_arg2):
...             print ("Я - обёртка вокруг декорируемой функции.\n"
...                    "И я имею доступ ко всем аргументам\n"
...                    "\t- и декоратора: {0} {1}\n"
...                    "\t- и функции: {2} {3}\n"
...                    "Теперь я могу передать нужные аргументы дальше"
...                    .format(decorator_arg1, decorator_arg2,
...                            function_arg1, function_arg2))
...             return func(function_arg1, function_arg2)
...         return wrapped
...     return my_decorator
...
>>> @decorator_maker_with_arguments("Леонард", "Шелдон")
... def decorated_function_with_arguments(function_arg1, function_arg2):
...     print ("Я - декорируемая функция и я знаю только о своих аргументах: {0}"
...            " {1}".format(function_arg1, function_arg2))
...
Я создаю декораторы! И я получил следующие аргументы: Леонард Шелдон
Я - декоратор. И ты всё же смог передать мне эти аргументы: Леонард Шелдон
>>> decorated_function_with_arguments("Раджеш", "Говард")
Я - обёртка вокруг декорируемой функции.
И я имею доступ ко всем аргументам
    - и декоратора: Леонард Шелдон
    - и функции: Раджеш Говард
Теперь я могу передать нужные аргументы дальше
Я - декорируемая функция и я знаю только о своих аргументах: Раджеш Говард
```

Таким образом, мы можем передавать декоратору любые аргументы, как обычной функции. Мы можем использовать и распаковку через *args и **kwargs в случае необходимости.

Некоторые особенности работы с декораторами

- Декораторы несколько замедляют вызов функции, не забывайте об этом.
- Вы не можете "раздекорировать" функцию. Безусловно, существуют трюки, позволяющие создать декоратор, который можно отсоединить от функции, но это плохая практика. Правильнее будет запомнить, что если функция декорирована — это не отменить.
- Декораторы оборачивают функции, что может затруднить отладку.

Последняя проблема частично решена добавлением в модуле functools функции functools.wraps, копирующей всю информацию об оборачиваемой функции (её имя, из какого она модуля, её документацию и т.п.) в функцию-обёртку.

Забавным фактом является то, что functools.wraps тоже является декоратором.

```
>>> def foo():
...     print("foo")
...
>>> print(foo.__name__)
foo
>>> # Однако, декораторы мешают нормальному ходу дел:
... def bar(func):
...     def wrapper():
...         print("bar")
...         return func()
...     return wrapper
...
>>> @bar
... def foo():
...     print("foo")
...
>>> print(foo.__name__)
wrapper
>>> import functools  # "functools" может нам с этим помочь
>>> def bar(func):
...     # Объявляем "wrapper" оборачивающим "func"
...     # и запускаем магию:
...     @functools.wraps(func)
...     def wrapper():
...         print("bar")
...         return func()
...     return wrapper
...
>>> @bar
... def foo():
...     print("foo")
...
>>> print(foo.__name__)
foo
```

Примеры использования декораторов

Декораторы могут быть использованы для расширения возможностей функций из сторонних библиотек (код которых мы не можем изменять), или для упрощения отладки (мы не хотим изменять код, который ещё не устоялся).

Также полезно использовать декораторы для расширения различных функций одним и тем же кодом, без повторного его переписывания каждый раз, например:

```
>>> def benchmark(func):
...     """
...     Декоратор, выводящий время, которое заняло
...     выполнение декорируемой функции.
...     """
...     import time
...     def wrapper(*args, **kwargs):
...         t = time.clock()
...         res = func(*args, **kwargs)
...         print(func.__name__, time.clock() - t)
...         return res
...     return wrapper
...
>>> def logging(func):
...     """
...     Декоратор, логирующий работу кода.
...     (хорошо, он просто выводит вызовы, но тут могло быть и логирование!)
...     """
...     def wrapper(*args, **kwargs):
...         res = func(*args, **kwargs)
...         print(func.__name__, args, kwargs)
...         return res
...     return wrapper
...
>>> def counter(func):
...     """
...     Декоратор, считающий и выводящий количество вызовов
...     декорируемой функции.
...     """
...     def wrapper(*args, **kwargs):
...         wrapper.count += 1
...         res = func(*args, **kwargs)
...         print("{0} была вызвана: {1}x".format(func.__name__, wrapper.count))
...         return res
...     wrapper.count = 0
...     return wrapper
...
>>> @benchmark
... @logging
... @counter
... def reverse_string(string):
...     return ''.join(reversed(string))
```

```
...
>>> print(reverse_string("А роза упала на лапу Азора"))
reverse_string была вызвана: 1x
wrapper ('А роза упала на лапу Азора',) {}
wrapper 0.00011799999999997923
арозА упал ан алапу азор А
>>> print(reverse_string("A man, a plan, a canoe, pasta, heros, rajahs, a␣
↪coloratura,"
... "maps, snipe, percale, macaroni, a gag, a banana bag, a tan, a tag,"
... "a banana bag again (or a camel), a crepe, pins, Spam, a rut, a Rolo, cash,"
... "a jar, sore hats, a peon, a canal: Panama!"))
reverse_string была вызвана: 2x
wrapper ('A man, a plan, a canoe, pasta, heros, rajahs, a coloratura,maps, snipe, .
↪..',) {}
wrapper 0.00017800000000001148
!amanaP :lanac a ,noep a ,stah eros ,raj a,hsac ,oloR a ,tur a ,mapS ,snip ,eperc␣
↪a , ...
```

Глава 32

Устанавливаем python-пакеты с помощью pip

pip - это система управления пакетами, которая используется для установки и управления программными пакетами, написанными на Python.

Установка pip

Прежде чем с помощью pip устанавливать python-пакеты, нужно сначала установить сам pip.

Python 3.4+

Начиная с Python версии 3.4, pip поставляется вместе с интерпретатором python.

Python <3.4

Официальная инструкция (https://pip.pypa.io/en/latest/installing.html):

- Загрузить get-pip.py (обязательно сохранив с расширением .py).
- Запустить этот файл (могут потребоваться права администратора).

Есть ещё один способ (для Windows). Возможно, он является более предпочтительным:

- Установить setuptools http://www.lfd.uci.edu/~gohlke/pythonlibs/#setuptools
- Установить pip http://www.lfd.uci.edu/~gohlke/pythonlibs/#pip

Начало работы

Попробуем с помощью pip установить какой-нибудь пакет, например, numpy:

Linux:

```
sudo pip3 install numpy
```

На Windows:

```
pip3 install numpy
```

Может не сработать, написав: *"python" не является внутренней или внешней командой, исполняемой программой или пакетным файлом* (такого, скорее всего, не должно быть при установке pip вторым способом, но проверить не на чем).

Тогда прийдётся обращаться напрямую:

```
C:\Python34\Tools\Scripts\pip3.exe install numpy
```

Либо добавлять папку C:\Python34\Tools\Scripts\ в PATH вручную (самому проверить не на чем, можете посмотреть на stackoverflow. У кого получится - напишите в комментарии).

Что ещё умеет делать pip

Пробежимся по основным командам pip:

pip help - помощь по доступным командам.

pip install package_name - установка пакета(ов).

pip uninstall package_name - удаление пакета(ов).

pip list - список установленных пакетов.

pip show package_name - показывает информацию об установленном пакете.

pip search - поиск пакетов по имени.

pip –proxy user:passwd@proxy.server:port - использование с прокси.

pip install -U - обновление пакета(ов).

pip install –force-reinstall - при обновлении, переустановить пакет, даже если он последней версии.

Глава 33

Часто задаваемые вопросы

Некоторые не совсем очевидные вещи, с которыми сталкиваются начинающие программисты Python.

Почему я получаю исключение UnboundLocalError, хотя переменная имеет значение?

Может показаться неожиданным получить `UnboundLocalError` в ранее работающем коде, в который добавили операцию присваивания где-то внутри функции.

Этот код:

```
>>> x = 10
>>> def bar():
...     print(x)
>>> bar()
10
```

работает, но следующий код:

```
>>> x = 10
>>> def foo():
...     print(x)
...     x += 1
```

приводит к `UnboundLocalError`:

```
>>> foo()
Traceback (most recent call last):
```

```
  ...
UnboundLocalError: local variable 'x' referenced before assignment
```

Это происходит потому, что, когда вы делаете присваивание переменной в области видимости, она становится локальной в этой области и скрывает другие переменные с таким же именем во внешних областях.

Когда последняя инструкция в foo присваивает новое значение переменной x, компилятор решает, что это локальная переменная. Следовательно, когда более ранний print пытается напечатать неинициализированную переменную, возникает ошибка.

В примере выше можно получить доступ к переменной, объявив её глобальной:

```
>>> x = 10
>>> def foobar():
...     global x
...     print(x)
...     x += 1
>>> foobar()
10
```

Это явное объявление требуется для того, чтобы напомнить вам, что (в отличие от внешне аналогичной ситуации с переменными класса и экземпляра), вы на самом деле, изменяете значение переменной во внешней области видимости:

```
>>> print(x)
11
```

Вы можете сделать подобную вещь во вложенной области видимости использованием ключевого слова nonlocal:

```
>>> def foo():
...    x = 10
...    def bar():
...        nonlocal x
...        print(x)
...        x += 1
...    bar()
...    print(x)
>>> foo()
10
11
```

Каковы правила для глобальных и локальных переменных в Python?

В Python, переменные, на которые только ссылаются внутри функции, считаются глобальными. Если переменной присваивается новое значение где-либо в теле функции,

считается, что она локальная, и, если вам нужно, то нужно явно указывать её глобальной.

Хотя это немного удивительно на первый взгляд, это легко объяснимо. С одной стороны, требование global для присваиваемых переменных предотвращает непреднамеренные побочные эффекты в bar. С другой стороны, если global был обязательным для всех глобальных ссылок, вы бы использовали global все время. Вы должны были бы объявить как глобальную каждую ссылку на встроенную функцию или компонент импортируемого модуля.

Почему анонимные функции (lambda), определенные в цикле с разными значениями, возвращают один и тот же результат?

Например, вы написали следующий код:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

Это даёт вам список из 5 функций, считающих x**2. Можно ожидать, что, будучи вызванными, они вернут, соответственно, 0, 1, 4, 9, и 16. Однако, вы увидите, что все они возвращают 16:

```
>>> squares[2]()
16
>>> squares[4]()
16
```

Это случается, поскольку x не является локальной для lambda, а определена во внешней области видимости, и получается тогда, когда она вызывается - а не когда определяется.

В конце цикла, x=4, поэтому все функции возвращают 4**2, то есть 16. Это можно также проверить, изменив значение x и посмотрев на результат:

```
>>> x = 8
>>> squares[2]()
64
```

Чтобы избежать подобного, необходимо сохранять значения переменных локально:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

Здесь, n=x создаёт локальную для функции переменную n и вычисляется в момент определения функции:

```
>>> squares[2]()
4
>>> squares[4]()
16
```

Это применимо не только к анонимным, а также и к обычным функциям.

Как организовать совместный доступ к глобальным переменным для нескольких модулей?

Канонический способ организовать подобный доступ - это создать отдельный модуль (часто называемый config или cfg). Просто добавьте `import config` в каждый модуль приложения. При этом модуль становится доступен через глобальное имя. Поскольку существует только один экземпляр модуля, любые изменения, произведённые в модуле отражаются везде. Например:

config.py:

```
x = 0
```

mod.py:

```
import config
config.x = 1
```

main.py:

```
import config
import mod
print(config.x)
```

По тем же соображениям, модули можно использовать как основу для имплементации синглтона.

Как правильнее использовать импортирование?

В общих случаях не используйте `from modulename import *`. Это засоряет пространство имён того, кто импортирует. Некоторые люди избегают этой идиомы даже для тех немногих модулей, которые были спроектированны, чтобы так импортироваться. Это такие модули как Tkinter и threading.

Импортируйте модули в начале файла. Это отвечает на вопрос, какие модули требует Ваш код и находится ли имя модуля в области видимости. Запись по одному импорту на строку упрощает добавление и удаление операторов импорта, но множественный импорт будет занимать меньше места на экране.

Хорошая практика, если Вы импортируете модули в следующем порядке:

- стандартные библиотечные модули (например, sys, os, getopt, re)
- модули сторонних разработчиков (всё, что установлено в директории site-packages) – например, PIL, NumPy и т.д.
- локально созданные модули

Иногда бывает необходимо поместить импорт в функцию или класс, чтобы избежать проблем с циклическим импортом. Gordon McMillan советует:

Циклический импорт отлично работает, если оба модуля используют форму import <module>. Но они терпят неудачу, когда второй модуль хочет извлечь имя из первого (from module import name) и импорт находится на внешнем уровне. Это происходит из-за того, что имена первого модуля ещё недоступны, так как первый модуль занят импортом второго.

В этом случае, если второй модуль используется только в одной функции, то импорт можно легко поместить в эту функцию. К тому времени, как он будет вызван, первый модуль уже закончит инициализацию и второй модуль осуществит свой импорт.

Может оказаться необходимым переместить импорт из начала файла, если один из модулей платформно-зависимый. В этом случае импорт всех модулей в начале файла окажется невозможным. В этой ситуации хорошим решением будет импорт нужных модулей в соответствующем платформно-зависимом коде.

Переносите импорт во вложенные области видимости, такие как определения функций, только если Вы столкнулись с проблемой, например циклического импорта, или если Вы пытаетесь сократить время инициализации модуля.

Эта техника полезна, если многие из импортов не являются необходимыми, и зависят от того, как программа будет исполняться. Вы также можете поместить импорт в функцию, если конкретные модули используются только в этой функции. Обратите внимание, что загрузить модуль в первый раз может оказаться дорого из-за задержки на инициализацию модуля, однако повторные загрузки “бесплатны”, они стоят только пары поисков в словарях. Даже если имя модуля исчезло из области видимости, модуль скорее всего до сих пор находится в `sys.modules`.

Почему значения по умолчанию разделяются между объектами?

Этот тип ошибки часто встречается среди начинающих. Предположим, функция:

```
def foo(mydict={}):  # Опасность: разделяемая ссылка между вызовами
    ... compute something ...
    mydict[key] = value
    return mydict
```

В первый раз, когда вы вызываете функцию, `mydict` содержит одно значение. Второй раз, `mydict` содержит 2 элемента, поскольку, когда `foo()` начинает выполняться, `mydict` уже содержит элемент.

Часто ожидается, что вызов функции создаёт новые объекты для значений по умолчанию. Но это не так. Значения по умолчанию создаются лишь однажды, когда функция определяется. Если этот объект изменяется, как словарь в нашем примере, последующие вызовы функции будут использовать изменённый объект.

По определению, неизменяемые объекты (числа, строки, кортежи и None), безопасны при изменении. Изменение изменяемых объектов, таких как словари, списки, и экземпляры пользовательских классов может привести к неожиданным последствиям.

Поэтому, хорошей практикой является не использовать изменяемые объекты в качестве значений по умолчанию. Вместо этого, используйте None и внутри функции, проверяйте аргумент на None и создавайте новый список/словарь. Например, не пишите:

```
def foo(mydict={}):
    ...
```

Но пишите так:

```
def foo(mydict=None):
    if mydict is None:
        mydict = {}  # create a new dict for local namespace
```

Однако, эта особенность может быть полезна. Когда у вас есть функция, которая долго выполняется, часто применяемая техника - кэширование параметров и результата каждого вызова функции:

```
def expensive(arg1, arg2, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # Расчёт значения
    result = ... expensive computation ...
    _cache[(arg1, arg2)] = result     # Кладём результат в кэш
    return result
```

Как передать опциональные или именованные параметры из одной функции в другую?

Получить такие параметры можно с помощью спецификаторов * и ** в списке аргументов функции; они возвращают кортеж позиционных аргументов и словарь именованых параметров. После этого Вы можете передать их в другую функцию, используя в её вызове * и **:

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

Почему изменение списка 'y' изменяет также список 'x'?

Если вы написали код:

```
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

вы, возможно, будете удивлены тому, что добавление в y изменяет также и x.

Два факта приводят к такому результату:

- Переменные - это просто ссылки на объекты. `y = x` не создаёт копию списка - это просто создаёт переменную y, которая ссылается на **тот же** объект, что и x.
- Списки изменяемы.

После вызова append, содержимое объекта было изменено с `[]` на `[10]`. Поскольку x и y ссылаются на один и тот же объект, использование любого из них даёт нам `[10]`.

Если мы используем неизменяемые объекты:

```
>>> x = 5  # числа неизменяемы
>>> y = x
>>> x = x + 1  # 5 нельзя изменить. Мы создаём НОВЫЙ объект
>>> x
6
>>> y
5
```

мы можем видеть, что x и y больше не равны, поскольку числа неизменяемы, и `x = x + 1` не изменяет число 5 путем увеличения. Вместо этого, создаётся новый объект 6 и присваивается переменной x (то есть, изменяется то, на какой объект ссылается x). После этого у нас 2 объекта (6 и 5) и 2 переменные, которые на них ссылаются.

Некоторые операции (например `y.append(10)` и `y.sort()`) изменяют объект, в то время, как внешне похожие операции (например `y = y + [10]` и `sorted(y)`) создают новый объект. Вообще в Python (и во всех случаях в стандартной библиотеке), метод, который изменяет объект, возвращает None, чтобы помочь избежать ошибок. Поэтому, если вы написали

```
y = y.sort()
```

думая, что это даст вам отсортированную копию y, вы вместо этого получите None, что скорее всего приведёт к легко диагностируемой ошибке.

Однако, существует один класс операций, где одна и та же операция ведёт себя по-разному с различными типами: расширенные операторы присваивания. Например, +=

изменяет списки, но не кортежи или числа (`a_list += [1, 2, 3]` эквивалентно `a_list.extend([1, 2, 3])`) и изменяет список, в то время, как `some_tuple += (1, 2, 3)` и `some_int += 1` создают новый объект.

Если вы хотите знать, ссылаются ли 2 переменные на один объект или нет, вы можете использовать оператор `is`, или встроенную функцию `id`.

Как создавать функции более высокого порядка?

Есть два пути: использовать вложенные функции или вызываемые объекты. Например, с использованием вложенных функций:

```
def linear(a, b):
    def result(x):
        return a * x + b
    return result
```

Использование вызываемого объекта:

```
class linear:

    def __init__(self, a, b):
        self.a, self.b = a, b

    def __call__(self, x):
        return self.a * x + self.b
```

В обоих случаях,

```
taxes = linear(0.3, 2)
```

даёт функцию, что (к примеру) `taxes(10e6) == 0.3 * 10e6 + 2`.

Использование вызываемого объекта - немного медленнее, и в результате получается больше кода. Однако, заметьте, что несколько функций могут разделять свою сигнатуру с помощью наследования:

```
class exponential(linear):
    # __init__ наследуется
    def __call__(self, x):
        return self.a * (x ** self.b)
```

Объект может сохранять свое состояние для нескольких вызовов:

```
class counter:

    value = 0

    def set(self, x):
        self.value = x
```

```
    def up(self):
        self.value = self.value + 1

    def down(self):
        self.value = self.value - 1

count = counter()
inc, dec, reset = count.up, count.down, count.set
```

Здесь `inc`, `dec`, `reset` выступают в роли функций, которые разделяют одну и ту же переменную.

Как скопировать объект в Python?

В общем случае, с помощью модуля copy.

Некоторые объекты можно скопировать более просто. Словари имеют метод copy:

```
newdict = olddict.copy()
```

Последовательности могут быть скопированы путём срезов:

```
new_l = l[:]
```

Как узнать доступные методы и атрибуты объекта?

`dir(x)` возвращает список методов и атрибутов.

Как можно узнать имя объекта?

Вообще говоря, никак, поскольку объекты в действительности не имеют имён. Важно: присваивание всегда связывает имя с объектом. Это верно и для инструкций `def` и `class`.

```
>>> class A:
...     pass
...
>>> B = A
>>>
>>> a = B()
>>> b = a
>>> print(b)
<__main__.A object at 0x7fbcc3ee5160>
>>> print(a)
<__main__.A object at 0x7fbcc3ee5160>
```

Возможно, класс имеет имя: однако, хотя он связан с двумя именами и запрашивается через имя B, созданный экземпляр всё ещё считается экземпляром класса A. Однако, невозможно сказать, имя экземпляра a или b, поскольку оба они связаны с одним и тем же значением.

Какой приоритет у оператора "запятая"?

Запятая не является оператором в Python.

```
>>> "a" in "b", "a"
(False, 'a')
```

Поскольку запятая - не оператор, но разделитель между выражениями, пример выше исполняется как если бы было введено:

```
("a" in "b"), "a"
```

А не

```
"a" in ("b", "a")
```

То же самое верно и для операторов присваивания (=, += и другие). Они не являются операторами как таковыми, а лишь синтаксическими разделителями в операциях присваивания.

Есть ли в Python эквивалент тернарного оператора "?:" в C?

Да. Синтаксис:

```
[on_true] if [expression] else [on_false]
```

```
x, y = 50, 25
small = x if x < y else y
```

Можно ли писать обфусцированные однострочники?

Можно.

```
from functools import reduce

# Простые числа < 1000
print(list(filter(None,map(lambda y:y*reduce(lambda x,y:x*y!=0,
map(lambda x,y=y:y%x,range(2,int(pow(y,0.5)+1))),1),range(2,1000)))))
```

```
# Первые 10 чисел Фибоначчи
print(list(map(lambda x,f=lambda x,f:(f(x-1,f)+f(x-2,f)) if x>1 else 1:
f(x,f), range(10))))

# Множество Мандельброта
print((lambda Ru,Ro,Iu,Io,IM,Sx,Sy:reduce(lambda x,y:x+y,map(lambda y,
Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,Sy=Sy,L=lambda yc,Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,i=IM,
Sx=Sx,Sy=Sy:reduce(lambda x,y:x+y,map(lambda x,xc=Ru,yc=yc,Ru=Ru,Ro=Ro,
i=i,Sx=Sx,F=lambda xc,yc,x,y,k,f=lambda xc,yc,x,y,k,f:(k<=0)or (x*x+y*y
>=4.0) or 1+f(xc,yc,x*x-y*y+xc,2.0*x*y+yc,k-1,f):f(xc,yc,x,y,k,f):chr(
64+F(Ru+x*(Ro-Ru)/Sx,yc,0,0,i)),range(Sx))):L(Iu+y*(Io-Iu)/Sy),range(Sy
))))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24))
#    \___ ___/  \___ ___/  |   |   |__ lines on screen
#        V          V      |   |______ columns on screen
#        |          |      |__________ maximum of "iterations"
#        |          |_________________ range on y axis
#        |____________________________ range on x axis
```

Не пытайтесь это делать дома!

Почему -22 // 10 равно -3?

Поскольку `i % j` имеет тот же знак, что `j`. А ещё

```
i == (i // j) * j + (i % j)
```

Как можно изменить строку?

Никак, поскольку строки неизменяемы. В большинстве ситуаций, нужно просто сделать новую строку из различных частей. Однако, если так нужно, можно использовать `io.StringIO`, либо модуль array:

```
>>> import io
>>> s = "Hello, world"
>>> sio = io.StringIO(s)
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

>>> import array
```

```
>>> a = array.array('u', s)
>>> print(a)
array('u', 'Hello, world')
>>> a[0] = 'y'
>>> print(a)
array('u', 'yello, world')
>>> a.tounicode()
'yello, world'
```

Как использовать строки для вызова функций/методов?

Существует несколько приёмов.

- Лучший - использование словаря, ставящего соответствие строке функцию. Его главное достоинство - строки не обязаны совпадать с названиями функций.

```
def a():
    pass

def b():
    pass

dispatch = {'go': a, 'stop': b}  # Note lack of parens for funcs

dispatch[get_input()]()
```

- Использование встроенной функции getattr:

```
import foo
getattr(foo, 'bar')()
```

- Использование locals или eval (не рекомендуется)

```
def myFunc():
    print("hello")

fname = "myFunc"

f = locals()[fname]
f()

f = eval(fname)
f()
```

Как удалить все символы новой строки в конце строки?

Можно использовать `S.rstrip("\r\n")` для удаления символов новой строки, без удаления конечных пробелов:

```
>>> lines = ("line 1 \r\n"
...          "\r\n"
...          "\r\n")
>>> lines.rstrip("\r\n")
'line 1 '
```

Как удалить повторяющиеся элементы в списке?

Существует несколько путей: http://code.activestate.com/recipes/52560/

Как создать многомерный список?

Возможно, вы попробуете этот неудачный вариант:

```
>>> A = [[None] * 2] * 3
```

Это выглядит правильно, если напечатать:

```
>>> A
[[None, None], [None, None], [None, None]]
```

Но если вы присвоите значение, то оно появится в нескольких местах:

```
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

Причина в том, что оператор `*` не создаёт копию, а только ссылку на существующий объект. `*3` создаёт список из 3 ссылок на один и тот же список. Изменение в одной строке изменяют другие, что, вероятно, не то, что вы хотите.

Возможные пути решения:

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

```
w, h = 2, 3
A = [[None] * w for i in range(h)]
```

Или, можно использовать специальные модули, предоставляющие матрицы. Наиболее известным является NumPy.

Почему a_tuple[i] += ['item'] не работает, а добавление работает?

Это из-за того, что расширенный оператор присваивания - оператор *присваивания*, а также из-за разницы между изменяемыми и неизменяемыми объектами в Python.

Это обсуждение относится в общем, когда расширенные операторы присваивания применяются к элементам кортежа, которые указывают на изменяемые объекты, но мы будем использовать список и +=, как образец.

Если вы напишете:

```
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

Причина исключения должна быть понятна: 1 добавляется к объекту `a_tuple[0]`, но когда мы пытаемся присвоить результат, 2, к первому элементу в кортеже, мы получаем ошибку, поскольку мы не можем изменить элемент кортежа.

То есть, это выражение делает следующее:

```
>>> result = a_tuple[0] + 1
>>> a_tuple[0] = result
Traceback (most recent call last):
 ...
TypeError: 'tuple' object does not support item assignment
```

Когда мы пишем что-то вроде:

```
>>> a_tuple = (['foo'], 'bar')
>>> a_tuple[0] += ['item']
Traceback (most recent call last):
 ...
TypeError: 'tuple' object does not support item assignment
```

Исключение немного более неожиданное, но более удивителен тот факт, что, несмотря на ошибку, элемент добавился!

```
>>> a_tuple[0]
['foo', 'item']
```

Чтобы понять, что случилось, нужно знать, что:

- Если объект определяет метод `__iadd__`, он вызывается, когда выполняется +=, и возвращенное значение используется для присваивания
- Для списков, `__iadd__` эквивалентен вызову `extend` для списка

Таким образом,

```
>>> a_list = []
>>> a_list += [1]
>>> a_list
[1]
```

Эквивалентен:

```
>>> result = a_list.__iadd__([1])
>>> a_list = result
```

Таким образом, наш пример с кортежом эквивалентен:

```
>>> result = a_tuple[0].__iadd__(['item'])
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

__iadd__ завершился успешно, и список увеличился, но присваивание законилось ошибкой.

Глава 34

Задачи по Python

Каждому изучающему Python нужно писать код для закрепления. Вашему вниманию предлагаются несколько задач для реализации (не слишком простых (кроме первой) и не слишком сложных).

Для выполнения заданий крайне рекомендуется пройти самоучитель.

Также для этих задач есть репозиторий с тестами и моими решениями (чтобы проверить себя).

Для запуска тестов для вашей функции проще всего будет добавить код из папки с тестами в конец файла с функцией.

А теперь, собственно, задачи:

Простейшие арифметические операции (1)

Написать функцию `arithmetic`, принимающую 3 аргумента: первые 2 - числа, третий - операция, которая должна быть произведена над ними. Если третий аргумент `+`, сложить их; если `--`, то вычесть; `*` — умножить; `/` — разделить (первое на второе). В остальных случаях вернуть строку “`Неизвестная операция`”.

Високосный год (2)

Написать функцию `is_year_leap`, принимающую 1 аргумент — год, и возвращающую True, если год високосный, и False иначе.

Квадрат (3)

Написать функцию `square`, принимающую 1 аргумент — сторону квадрата, и возвращающую 3 значения (с помощью кортежа): периметр квадрата, площадь квадрата и диагональ квадрата.

Времена года (4)

Написать функцию `season`, принимающую 1 аргумент — номер месяца (от 1 до 12), и возвращающую время года, которому этот месяц принадлежит (`зима`, `весна`, `лето` или `осень`).

Банковский вклад (5)

Пользователь делает вклад в размере `a` рублей сроком на `years` лет под 10% годовых (каждый год размер его вклада увеличивается на 10%. Эти деньги прибавляются к сумме вклада, и на них в следующем году тоже будут проценты).

Написать функцию `bank`, принимающая аргументы `a` и `years`, и возвращающую сумму, которая будет на счету пользователя.

Простые числа (6)

Написать функцию `is_prime`, принимающую 1 аргумент — число от 0 до 1000, и возвращающую True, если оно простое, и False - иначе.

Правильная дата (7)

Написать функцию `date`, принимающую 3 аргумента — день, месяц и год. Вернуть True, если такая дата есть в нашем календаре, и False иначе.

XOR-шифрование (8)

Написать функцию `XOR_cipher`, принимающая 2 аргумента: строку, которую нужно зашифровать, и ключ шифрования, которая возвращает строку, зашифрованную путем применения функции XOR (^) над символами строки с ключом. Написать также функцию `XOR_uncipher`, которая по зашифрованной строке и ключу восстанавливает исходную строку.