Пройдено Позже

Начало
или
Заложим фундамент

Идеи

Пройдено Позже

Интерпретатор. Режимы работы

Существует 2 режима работы:

- Интерактивный - в режиме режиме интерпретатора.

```
python
Python 3.5.0 (default, Sep 23 2015, 04:41:33)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- Запуск программы из файла.

```
python program.py
```

Идеи от разработчиков

Разработчики python предлагают и пропагандируют идею хорошей разработки:

```
>>> import this
```

Введите в интерпретаторе эту строку и увидите такие строки.

```
The Zen of Python, by Tim Peters
```

```
2
3  Beautiful is better than ugly.
4  Explicit is better than implicit.
5  Simple is better than complex.
6  Complex is better than complicated.
7  Flat is better than nested.
8  ...
9
```

О чем это?

Красивое лучше, чем уродливое.
Явное лучше, чем неявное.
Простое лучше, чем сложное.
Сложное лучше, чем запутанное.
Плоское лучше, чем вложенное.
Разреженное лучше, чем плотное.
Читаемость имеет значение.
Особые случаи не настолько особые, чтобы нарушать правила.
При этом практичность важнее безупречности.
Ошибки никогда не должны замалчиваться.

История

Разработка языка Python была начата в конце 1980-х годов[8] сотрудником голландского института CWI Гвидо ван Россумом. Для распределённой ОС Amoeba требовался расширяемый скриптовый язык, и Гвидо начал писать Python на досуге, позаимствовав некоторые наработки для языка ABC (Гвидо участвовал в разработке этого языка, ориентированного на обучение программированию). В феврале 1991 года Гвидо опубликовал исходный текст в ньюсгруппе alt.sources[9]. С самого начала Python проектировался как объектно-ориентированный язык.

Автор назвал язык в честь популярного британского комедийного телешоу 1970-х «Летающий цирк Монти Пайтона».

Наличие дружелюбного, отзывчивого сообщества пользователей считается наряду с

дизайнерской интуицией Гвидо одним из факторов успеха Python. Развитие языка происходит согласно чётко регламентированному процессу создания, обсуждения, отбора и реализации документов PEP (англ. Python Enhancement Proposal) — предложений по развитию Python[10].

3 декабря 2008 года[11], после длительного тестирования, вышла первая версия Python 3000 (или Python 3.0, также используется сокращение Py3k). В Python 3000 устранены многие недостатки архитектуры с максимально возможным (но не полным) сохранением совместимости со старыми версиями Python. На сегодня поддерживаются обе ветви развития (Python 3.x и 2.x).

Влияние других языков на Python

Появившись сравнительно поздно, Python создавался под влиянием множества языков программирования:

- ABC — отступы для группировки операторов, высокоуровневые структуры данных (map)[12][13] (Python фактически создавался как попытка исправить ошибки, допущенные при проектировании ABC);
- Modula-3 — пакеты, модули, использование `else` совместно с `try` и `except`, именованные аргументы функций (на это также повлиял Common Lisp);
- C, C++ — некоторые синтаксические конструкции (как пишет сам Гвидо ван Россум — он использовал наиболее непротиворечивые конструкции из C, чтобы не вызвать неприязнь у C-программистов к Python[12]);
- Smalltalk — объектно-ориентированное программирование;
- Lisp — отдельные черты функционального программирования (`lambda, map, reduce, filter` и другие);
- Fortran — срезы массивов, комплексная арифметика;
- Miranda — списочные выражения;
- Java — модули logging, unittest, threading (часть возможностей оригинального модуля не реализована), xml.sax стандартной библиотеки, совместное использование finally и except при обработке исключений, использование @ для декораторов;
- Icon — генераторы.

Синтаксис

Пройдено Позже

Основные правила

- Конец строки является концом инструкции (точка с запятой не требуется).

```
1  print('Hello World!')
2
```

```
1  print('Hello World!')
2  print('Hello Man!')
3
```

- Вложенные инструкции объединяются в блоки по величине отступов.

```
1  if 7 > 3:
2      print('Hello World!')
3
```

```
if 7 > 3:
    for x in range(1, 10):
        print('Hello World #', x)

```

- Вложенные инструкции отделяются от своего условия, задания цикла (и тп) двоеточием.

```
if 7 > 3:
    print('Hello World!')

```

```
if 7 > 3:
    for x in range(1, 10):
        print('Hello World #', x)

```

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 п
        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:

```
1  if x > y: print(x)
2
```

Например

```
1  # coding: utf-8
2  import sqlite3
3  con = sqlite3.connect('users.db')
4  cur = con.cursor()
5
6  #cur.execute('CREATE TABLE users (id integer primary key, firstName varchar(100),secondName varchar(30))
7  #cur.execute('insert into users(id, firstName, secondName)values(NULL, "User1", "1")')
8
9  cur.execute('insert into users(id, firstName, secondName)values(NULL, "User2", "2")')
10 cur.execute('insert into users(id, firstName, secondName)values(NULL, "User3", "3")')
11 con.commit()
12
13 cur.execute('SELECT * FROM users')
14 for row in cur:
15     print('-'*10)
16     print('ID:', row[0])
17     print('First name:', row[1])
18     print('Second name:', row[2])
19     print('-'*10)
20 con.close()
21
22
```

Еще пример

```
1
2  from apps.products.models import Product
3  from apps.importing.funcs import update_make_content, update_excel, NewPriceException
4
5  class UploadForm(forms.Form):
6      pricelist = forms.FileField(
7          label = u'Прайс-лист из БЕСТа',
8          widget=forms.FileInput,
9          #upload_to='files'
10     )
11     auto_id=True
12
13 @login_required()
14 def excel(request, *args, **kwargs):
15     template = 'admin/importing/excel.html'
16     content = {
17         'title': u'Импорт из БЕСТа - Обновление EXCEL',
18         'form': None,
19         'success': True,
20     }
21     update_excel()
22     context = RequestContext(request, {}, [context_processor_auth])
23     return render_to_response(template, content, context_instance=context)
24
25
```

Ветвления

Пройдено Позже

Условие

- Для проверки на условие используем конструкцию **if-else**.

```
1  if x > 12:
2      print("Выполнить")
3
4  if 1 <= y < 20:
5      print("Сегодня")
6  else:
7      print("Завтра")
8
```

- Объединяются условия с помощью литералов **and** и **or**.

```
1  if x == 7 and y != 100:
2      print("Wrong way!")
3
4  if x == 7 or y != 100:
5      print("Wrong way!")
6
```

- Для уточнения логики используйте скобки **(** и **)**.

```
1  if (x == 7 and y != 100) or (x == 7 and y == 17):
2      print("Wrong way!")
3
```

Знаки условий

```
1  x > y # - x больше, чем y
2  x < y # - x меньше, чем y
3  x == y # - x и y равны
4  x != y # - x не равен y
5
6  x >= y # - x больше или равен y
7  x <= y # - x меньше или равен y
8
9  # Допускается объединять в конструкции такого вида:
10 x <= y < z
11
12 # К условию для проверки на ошибочность добавляется not
13 if not uslovie:
14     print("No way!")
15
16
```

Цикл

- Для того, чтобы повторить действие многократно, используйте циклы.

```
for name in names: # проходит по всем элементам списка
    print(name)

k = 1
while k < 7: # проверяет на условие
    k += 1
```

- Более сложная логика достигается объединением конструкций.
- Для прерывания цикла используется литерал **break**.
- Для немедленного перехода к следующей итерации - **continue**.

```
k = 1
while k < 7: # проверяет на условие
    k += 1
    for name in names: # проходит по всем элементам списка
        if name == "end":
            break
        if len(name) == 0:
            continue
        print(1, name)

```

Например

```
if error == '':
    if not update:
        print("making: {}/{}".format(current, next))
    stat = next * 100 / total
    for row in range(current, next):
        tmp = {
            'sku_number': sheet.cell(rowx=row, colx=1).value.strip(),
            'num': row,
            'title': sheet.cell(rowx=row, colx=2).value.strip(),
            'unit': sheet.cell(rowx=row, colx=3).value.strip(),
            'price_rur': sheet.cell(rowx=row, colx=7).value,
            'category': sheet.cell(rowx=row, colx=0).value.strip()
        }
        # transform empty price into float
        if tmp['price_rur'] == '':
            tmp['price_rur'] = '0.0'
            price_no_count += 1


```

Еще пример

```python
class MessageBuffer(object):
    def __init__(self):
        self.waiters = set()
        self.cache = []
        self.cache_size = 200

    def wait_for_messages(self, cursor=None):
        print("MessageBuffer.wait_for_messages")
        # Construct a Future to return to our caller.  This allows
        # wait_for_messages to be yielded from a coroutine even though
        # it is not a coroutine itself.  We will set the result of the
        # Future when results are available.
        result_future = Future()
        if cursor:
            new_count = 0
            for msg in reversed(self.cache):
                if msg["id"] == cursor:
                    break
                new_count += 1
            if new_count:
                result_future.set_result(self.cache[-new_count:])
                return result_future
        self.waiters.add(result_future)
        return result_future
```

Содержание

Пробуем

Пройдено Позже

Для домашней работы

- В начале программы всегда указывайте кодировку файла.

```
# coding: utf-8
import os
print("Привет!") # Таким образом работа с русским текстом будет правильной
os.listdir('.') # Получаем список файлов в папке

```

- Работа с файлом

```
f = open("filename") # открывает файл на чтение
s = f.read()         # читает весь файл

f2 = open("filename_2", "w") # открывает файл на запись
f2.write("some text")        # записывает в файл данные
f2.close()                   # закрывает файл для сохранения

```

- Необходимо:

```
1. Найти в данной папке все файлы, в которых содержится слово python, и вывести на
    экран имена файлов.
2. Посчитать общее количество найденных слов и вывести на экран
3. Записать в файл "result.txt" список найденных файлов и число найденных слов python

```