Python – это язык программирования с открытым кодом, который был разработан в конце 1980-х Guido van Rossum и представлен в 1991. Впервые появился в ArcGIS версии 9.0 и с тех пор стал наиболее предпочтительным языком для пользователей, создающих рабочие процессы геообработки. Python поддерживается постоянно растущим сообществом пользователей и обеспечивает удобочитаемость, ясный синтаксис, динамический ввод и обладает широким набором стандартных библиотек и библиотек, созданных сторонними разработчиками.

Python 3: преимущества и недостатки языка

1. **Python - интерпретируемый язык программирования**. С одной стороны, это позволяет значительно упростить отладку программ, с другой - обуславливает сравнительно низкую скорость выполнения.
2. Динамическая типизация. В python не надо заранее объявлять тип переменной, что очень удобно при разработке.
3. Хорошая поддержка модульности. Вы можете легко написать свой модуль и использовать его в других программах.
4. Встроенная поддержка Unicode в строках. В Python необязательно писать всё на английском языке, в программах вполне может использоваться ваш родной язык.
5. Поддержка объектно-ориентированного программирования. При этом его реализация в python является одной из самых понятных.
6. Автоматическая сборка мусора, отсутствие утечек памяти.
7. Интеграция с C/C++, если возможностей python недостаточно.
8. Понятный и лаконичный синтаксис, способствующий ясному отображению кода. Удобная система функций позволяет при грамотном подходе создавать код, в котором будет легко разобраться другому человеку в случае необходимости. Также вы сможете научиться читать программы и модули, написанные другими людьми.
9. Огромное количество модулей, как входящих в стандартную поставку Python 3, так и сторонних. В некоторых случаях для написания программы достаточно лишь найти подходящие модули и правильно их скомбинировать. Таким образом, вы можете думать о составлении программы на более высоком уровне, работая с уже готовыми элементами, выполняющими различные действия.
10. Кроссплатформенность. Программа, написанная на Python, будет функционировать совершенно одинаково вне зависимости от того, в какой операционной системе она запущена. Отличия возникают лишь в редких случаях, и их легко заранее предусмотреть благодаря наличию подробной документации.

Синтаксис Python

Первое, что, как правило, бросается в глаза, если говорить о синтаксисе в Python, это то, что отступы имеют значение:

- они определяют, какой код попадает в блок;
- когда блок кода начинается и заканчивается.

Основные правила:

- 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.

Пример кода Python:

```
a = 10
b = 5

if a > b:
    print("A больше B")
    print(a - b)
else:
    print("B больше или равно A")
    print(b - a)

print("Конец")

def open_file(filename):
    print("Чтение файла", filename)
    with open(filename) as f:
        return f.read()
        print("Готово")
```

Комментарии

При написании кода часто нужно оставить комментарий, например, чтобы описать особенности работы кода.

Комментарии в Python могут быть однострочными:

```
# Очень важный комментарий
a = 10
b = 5 # Очень нужный комментарий
```

Однострочные комментарии начинаются со знака решётки. Обратите внимание, что комментарий может быть как в строке, где находится сам код, так и в отдельной строке.

При необходимости написать несколько строк с комментариями, чтобы не ставить перед каждой решётку, можно сделать многострочный комментарий:

```
"""
Очень важный
и длинный комментарий
"""
a = 10
b = 5
```

Переменные

Переменные в Python не требуют объявления типа переменной (так как Python – язык с динамической типизацией) и являются ссылками на область памяти. Правила именования переменных:

- имя переменной может состоять только из букв, цифр и знака подчёркивания;
- имя не может начинаться с цифры;
- имя не может содержать специальных символов @, $, %.

Пример создания переменных в Python:

```
>>> a = 3

>>> b = 'Hello'

>>> c, d = 9, 'Test'

>>> print(a,b,c,d)

3 Hello 9 Test
```

Обратите внимание, что в Python не нужно указывать, что «a» это число, а «b» это строка.

Переменные являются ссылками на область памяти. Это можно продемонстрировать с помощью id(), которая показывает идентификатор объекта:

```
>>> a = b = c = 33

>>> id(a)
31671480

>>> id(b)
31671480

>>> id(c)
31671480
```

В этом примере видно, что все три имени ссылаются на один и тот же идентификатор, то есть, это один и тот же объект, на который указывают три ссылки – «a», «b» и «c». С числами у Python есть одна особенность, которая может немного сбить с понимания: числа от -5 до 256 заранее созданы и хранятся в массиве (списке). Поэтому при создании числа из этого диапазона фактически создаётся ссылка на число в созданном массиве.

Имена переменных

Имена переменных не должны пересекаться с названиями операторов и модулей или же других зарезервированных слов. В Python есть рекомендации по именованию функций, классов и переменных:

- имена переменных обычно пишутся или полностью большими или полностью маленькими буквами (например DB_NAME, db_name);
- имена функций задаются маленькими буквами, с подчёркиваниями между словами (например, get_names);

- имена классов задаются словами с заглавными буквами без пробелов, это так называемый CamelCase (например, CiscoSwitch).

Типы данных в Python

В Python есть несколько стандартных типов данных:

- Numbers (числа)
- Strings (строки)
- Lists (списки)
- Dictionaries (словари)
- Tuples (кортежи)
- Sets (множества)
- Boolean (логический тип данных)

Эти типы данных можно, в свою очередь, классифицировать по нескольким признакам:

- изменяемые (списки, словари и множества)
- неизменяемые (числа, строки и кортежи)
- упорядоченные (списки, кортежи, строки и словари)
- неупорядоченные (множества)

Работа со строками в Python

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.

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.

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'
```

Списки (list). Функции и методы списков

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']
```

Словари (dict) и работа с ними

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

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

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

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

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

В-третьих, с помощью метода fromkeys:

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

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

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

Теперь попробуем добавить записей в словарь и извлечь значения ключей:

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

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:

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

Циклы for и 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

ВЫВОДИТ:
5
7
9
11
13
```

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

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

ВЫВОДИТ:
hheellllloo  wwoorrlldd
```

Функции

Функция - это блок кода, выполняющий определенные действия:

- у функции есть имя, с помощью которого можно запускать этот блок кода сколько угодно раз
  - запуск кода функции называется вызовом функции
- при создании функции, как правило, определяются параметры функции.
  - параметры функции определяют, какие аргументы функция может принимать
  - функциям можно передавать аргументы
  - соответственно, код функции будет выполняться с учетом указанных аргументов

Зачем нужны функции?

Как правило, задачи, которые решает код, очень похожи и часто имеют что-то общее.

Например, при работе с конфигурационными файлами каждый раз надо выполнять такие действия:

- открытие файла
- удаление (или пропуск) пустых строк
- удаление символов перевода строки в конце строк
- преобразование полученного результата в список

Дальше действия могут отличаться в зависимости от того, что нужно делать.

Часто получается, что есть кусок кода, который повторяется. Конечно, его можно копировать из одного скрипта в другой. Но это очень неудобно, так как при внесении изменений в код нужно будет обновить его во всех файлах, в которые он скопирован.

Гораздо проще и правильней вынести этот код в функцию (это может быть и несколько функций).

Создание функции:

- функции создаются с помощью зарезервированного слова **def**
- за **def** следуют имя функции и круглые скобки
- внутри скобок могут указываться параметры, которые функция принимает
- после круглых скобок идет двоеточие и с новой строки, с отступом, идет блок кода, который выполняет функция
- первой строкой, опционально, может быть комментарий, так называемая **docstring**
- в функциях может использоваться оператор **return**
  - он используется для прекращения работы функции и выхода из нее
  - чаще всего, оператор return возвращает какое-то значение

```
def open_file(filename):
    print("Чтение файла", filename)
    with open(filename) as f:
        return f.read()
        print("Готово")
```

Оператор return
Оператор **return** используется для возврата какого-то значения, и в то же время он завершает работу функции. Функция может возвращать любой объект Python. По умолчанию, функция всегда возвращает `None`

Пространства имен. Области видимости

У переменных в Python есть область видимости. В зависимости от места в коде, где переменная была определена, определяется и область видимости, то есть, где переменная будет доступна.

При использовании имен переменных в программе, Python каждый раз ищет, создает или изменяет эти имена в соответствующем пространстве имен. Пространство имен, которое доступно в каждый момент, зависит от области, в которой находится код.

У Python есть правило LEGB, которым он пользуется при поиске переменных.

Например, если внутри функции выполняется обращение к имени переменной, Python ищет переменную в таком порядке по областям видимости (до первого совпадения):

- L (local) - в локальной (внутри функции)
- E (enclosing) - в локальной области объемлющих функций (это те функции, внутри которых находится наша функция)
- G (global) - в глобальной (в скрипте)
- B (built-in) - во встроенной (зарезервированные значения Python)

Соответственно, есть локальные и глобальные переменные:

- локальные переменные:
    - переменные, которые определены внутри функции
    - эти переменные становятся недоступными после выхода из функции
- глобальные переменные:
    - переменные, которые определены вне функции
    - эти переменные „глобальны“ только в пределах модуля
    - например, чтобы они были доступны в другом модуле, их надо импортировать

Итераторы

Итератор (iterator) - это объект, который возвращает свои элементы по одному за раз.

Один из самых распространенных примеров итератора - файл.

Файл r1.txt:

```
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
```

```
service sequence-numbers
```

При работе с файлами, использование файла как итератора не просто позволяет перебирать файл построчно - в каждую итерацию загружена только одна строка. Это очень важно при работе с большими файлами на тысячи и сотни тысяч строк, например, с лог-файлами.

Основы ООП

- Класс (class) - элемент программы, который описывает какой-то тип данных. Класс описывает шаблон для создания объектов, как правило, указывает переменные этого объекта и действия, которые можно выполнять применимо к объекту.
- Экземпляр класса (instance) - объект, который является представителем класса.
- Метод (method) - функция, которая определена внутри класса и описывает какое-то действие, которое поддерживает класс
- Переменная экземпляра (instance variable, а иногда и instance attribute) - данные, которые относятся к объекту
- Переменная класса (class variable) - данные, которые относятся к классу и разделяются всеми экземплярами класса
- Атрибут экземпляра (instance attribute) - переменные и методы, которые относятся к объектам (экземплярам) созданным на основании класса. У каждого объекта есть своя копия атрибутов.

Пример из реальной жизни в стиле ООП:

- Проект дома - это класс
- Конкретный дом, который был построен по проекту - экземпляр класса
- Такие особенности как цвет дома, количество окон - переменные экземпляра, то есть конкретного дома
- Дом можно продать, перекрасить, отремонтировать - это методы