* Сделать игру на Python и запустить на серваке nginx/apache.

Информация про Python

Питон входит в топ популярнейших языков программирования по нынешним оценкам рейтинга TIOBE. Именно с него многие программисты начинают карьеру в своих домах или университетах. Вам не составит особого труда самостоятельно разобраться в языке.

Где применяется Python?

Несколько основных функциональных достоинств Python:

* Умеет работать с расширениями xml/html;
* Поддерживает управление http-запросами;
* Обладает графическим интерфейсом;
* Помогает создавать веб-скрипты;
* Может работать с FTP;
* Способен оперировать картинками, видео и звуковыми файлами;
* Используется в разработке робототехники;
* Отвечает за создание научных, вычислительных площадок и ещё много всего другого.

Можно сказать, что Python отлично подходит для выполнения большей части ежедневных задач программистов. Участвует как в создании обычных бекапов или чтении писем с почты, так и в разработке крупных игр. Питон практически не имеет ограничений по сферам использования, что позволяет использовать его в проектах колоссальных масштабов.

Python часто применяется китами мира IT – Google и Яндекс. Если добавить к этому простоту создания программ, несложно понять, почему Питон заслуживает место в списке лучших языков.

Чтобы начать работать на Python, прежде стоит пройти на официальный веб-ресурс для скачивания дистрибутива языка. На сайте есть масса полезной информации в отношении правил написания кода и расширяемости языка.

Сфера применения языка выходит далеко за пределы разработки обычных приложений. Он используется в разработке игр, веб-программировании и многом другом. Для работы в подобных направлениях нужно только подобрать, скачать и установить соответствующий фреймворк. Библиотеки помогут расширить функции языка в конкретном направлении. С библиотеками проще создавать проекты, так как не требуется создавать «велосипед».

* python3 - вход в питон
* python3 --version и python --version - разное.
* python example.py - запуск файла с кодом питон (его интерпретация и вывод)
* python язык не компилируемый, а интерпретируемый. Тут нет ни какой компиляции.
* print("Hello") - вывод Hello

ПРИ ОШИБКЕ: *File "./y.sh", line 2  
SyntaxError: Non-ASCII character 'xd0' in file ./y.sh on line 2, but no encoding declared; see http://www.python.org/peps/pep-0263.html for details*

Что нужно добавить, чтобы сказать интерпретатору о наличии юникода?

\*\* Нужно добавить в начало специальный комментарий.

#! /usr/bin/env python

# -\*- coding: utf-8 -\*-

print "Привет"

\* Этак с начала времён (юниксовых, т.е. 1.01.1970) в начале скрипта рекомендовалось использовать shebang / hashbang - строку, указывающую на используемый интерпретатор, например:

#!/bin/bash

echo 'hello world'

#!/usr/bin/python

print 'hello world'

#!/usr/bin/python3

print('hello world')

Недавно наткнулся на рекомендацию использовать вместо этого такую форму:

#!/usr/bin/env bash

echo 'hello world'

#!/usr/bin/env python

print 'hello world'

#!/usr/bin/env python3

print('hello world')

Пожалуйста, объясните, как это работает и в чём преимущества такого подхода? Если есть ограничения и/или недостатки по сравнению с обычным способом - то и о них хотелось бы услышать.

\* Простейший пример из практики: в Linux bash лежит в /bin/bash, а во FreeBSD в /usr/local/bin/bash и первый вариант не запустится. В своё время напоролся на это и теперь везде использую только второй вариант) – [andreymal](https://ru.stackoverflow.com/users/178213/andreymal" \o "10,469 баллов репутации) [5 июл '16 в 9:18](https://ru.stackoverflow.com/questions/541589/%D0%92-%D1%87%D1%91%D0%BC-%D1%81%D0%BC%D1%8B%D1%81%D0%BB-%D0%B8-%D0%BF%D1%80%D0%B5%D0%B8%D0%BC%D1%83%D1%89%D0%B5%D1%81%D1%82%D0%B2%D0%B0-usr-bin-env#comment685903_541589)

\* Основная идея - улучшение переносимости. Не гарантируется, что на различных системах исполняемый файл будет лежать по пути, который указан в shebang.

Использование env позволяет снизить этот риск за счет запуска команды на основе данных из переменной среды PATH

Более того, если по каким-либо причинам вместо стандартного исполняемого файла пользователь хочет использовать свой, то ему достаточно добавить путь к этому файлу в PATH без необходимости исправления скриптов:

~ $ cp /bin/bash /home/soon/python

~ $ export PATH=/home/soon:$PATH

~ $ env python

[soon@archlinux ~]$ exit

В примере выше я скопировал bash к себе в домашнюю директорию (переименовав при этом файл в python), добавил путь в PATH и запустил *python* с помощью env, которая усужливо запустила bash, т.к. нашла его раньше.

Еще одним примером является использование виртуальных окружений при разработке на Python (virtualenv). Поскольку они также перебивают PATH, env позволяет использовать нужную версию исполняемого файла:

~ $ workon black-box-challenge-2016-py2

~ (venv:black-box-challenge-2016-py2) $ env python

Python 2.7.11 (default, Mar 31 2016, 06:18:34)

[GCC 5.3.0] on linux2

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

>>> import sys

>>> print sys.executable

/home/soon/.virtualenvs/black-box-challenge-2016-py2/bin/python

>>>

#! /usr/bin/env python3

# -\*- coding: utf-8 -\*-

print("Hello")

print(3 \*\* 2) - 3 в степени 2

print(3 + 2)

print(2.56 - 45)

print(25 // 7)

# сколько 7 в числе 25 - вывод:3

print(25 % 7) # остаток

\* Since you are writing for Python 3.x, you'll want to begin your script with:

#!/usr/bin/env python3

If you use:

#!/usr/bin/env python

It will default to Python 2.x. These go on the first line of your script, if there is nothing that starts with *#!* (aka the shebang).

If your scripts just start with:

#! python

Then you can change it to:

#! python3

Although this shorter formatting is only recognized by a few programs, such as the launcher, so it is not the best choice.

The first two examples are much more widely used and will help ensure your code will work on any machine that has Python installed.

\* You should use raw\_input because you are using python-2.7. When you use input() on a variable (for example: s = input('Name: ')), it will execute the command ON the Python environment without saving what you wrote on the variable (s) and create an error if what you wrote is not defined.

raw\_input() will save correctly what you wrote on the variable (for example: f = raw\_input('Name : ')), and it will not execute it in the Python environment without creating any possible error:

input\_variable = raw\_input('Enter Your Name : ')

print("Your Name Is : " + (input\_variable))

\* For anyone else that may run into this issue, turns out that even if you include #!/usr/bin/env python3 at the beginning of your script, the shebang is ignored if the file isn't executable.

To determine whether or not your file is executable:

* run ./filename.py from the command line
* if you get -bash: ./filename.py: Permission denied, run chmod a+x filename.py
* run ./filename.py again

If you've included import sys; print(sys.version) as Kevin suggested, you'll now see that the script is being interpreted by python3

import sys

print(sys.version\_info)

Put as first line import sys and as second line print(sys.version\_info) in order to ascertain which version you are using.

namge = input("Как Вас зовут? ")

print("Привет,", namge)

#Works in Python 2 and 3:

try: input = raw\_input

except NameError: pass

print("Hi, " + input("Your name: "))

Создание сайтов на Python

На сегодня самой популярной библиотекой является Django - это фреймворк для веб-разработки сайтов, в котором есть функции для быстрого добавления действий на веб-ресурс. Google активно применяет язык в ряде собственных проектов, а всё по причине удобной работы и широкого функционала, применимого в любых направлениях программирования. Если метите на место в Гугл, изучение Python – неплохой шанс начать работать в компании.

Типы данных и переменные в Python

Типы переменных в языке Python не объявляются очевидно, тем не менее они присутствуют. Интерпретатор понимает, что записывается в переменную и на основании этого добавляет тип к этой переменной.

Во время исполнения программы есть возможность перезаписывать переменные, а также менять их тип. Если вначале переменной был задан тип float, то потом её можно преобразовать в другой тип, например, string.

first\_num = 23.2 # Тип данных float

first\_num = "1" # Тип данных string

При объединении нескольких переменных с разными типами данных программа спровоцирует ошибку.

Пример:

first\_num = "IloveYou"

second\_num = 13

res = first\_num + second\_num # Скрипт выдаст ошибку

Всего в Python есть 4 базовых типа переменных:

some = 1 Integer - целые числа;

some = 1.12 Float - числа с плавающей точкой;

some = "Привет" String - строки;

some = True Boolean - тип данных принимающий либо False, либо True.

Есть и другие типы, но мы будем их разбирать в последующих уроках.

В одной строке можно создать сразу несколько переменных:

first = sec = third = 1 # Всем трём переменным будет присвоено значение 1

first, sec, third = "Hi", 75, 23.1 # Поочередное присвоение значений

* все, что вводим с клавиатуры является тип string

num\_1 = input ("Enter first num: ") # 25

num\_2 = input ("Enter second num: ") # 14

res = num\_1 + num\_2

print("Result is", res)

ВЫВОД: Result is 2514

num\_1 = int (input ("Enter first num: ")) # 25

num\_2 = int (input ("Enter second num: ")) # 14

res = num\_1 + num\_2

print("Result is", res)

ВЫВОД: Result is 39

num\_1 = int (input ("Enter first num: "))

num\_2 = input ("Enter second num: ")

res = num\_1 + int (num\_2)

print("Result is", res)

del res # удаление переменной

num\_1 = float (input ("Enter first num: "))

num\_2 = input ("Enter second num: ")

res = num\_1 + float (num\_2)

res += 5 # res = res + 5

print("Result is", res)

num\_1 = float (input ("Enter first num: "))

num\_2 = input ("Enter second num: ")

res = num\_1 + float (num\_2)

res += 5

Res = input ("Enter your string: ")

Res \*= 5

print(Res)

print("Result is", res)

ВЫВОД:

Enter first num: 14

Enter second num: 1

Enter your string: Test

TestTestTestTestTest

Result is 20.0

* Чувствителен к регистру, а также (матем.операции???) умножение (деление, вычитание и сложение выдают ошибку) действуют и на строковые переменные

#5 - Условные операторы

Пример создания условия:

a = 2

if a != 0 and a != 1:

print ("Проверка сработала")

На экране будет показана запись лишь в том случае, когда переменная «a» не будет равна значению 0 и значению 1. То есть обе проверки в операторе должны выдать результат – true.

Есть возможность произвести проверку с помощью or - логическое или. При использовании данного оператора достаточным поводом для запуска сообщения «Заработало» станет соответствие хотя бы одного из условий.

Пример:

a = 1.1

if a != 1.1 or a > 0:

print ("Проверка сработала")

if 1:

  print("True\n")

  print("Port")

* if 1 - условие всегда истинно, 0 - ложь

if 0:

  print("True\n")

print("Port")

* выведет только Port

num = input ("Your name: ")

if num == "Test":

  print("True\n")

print("Port")

num = input ("Your number: ")

if int(num) > 0:

  print("True\n")

print("Port")

num = input ("Your number: ")

if int(num) > 0:

  print("Pos\n")

elif int(num) < -10:

  print("Neg")

print("Port")

num = input ("Your number: ")

if int(num) > 0:

  print("Pos\n")

elif int(num) < -10:

  print("Neg")

else:

  print("from -10 to 0")

print("Port")

num = input ("Your number: ")

if int(num) > 0:

  if int(num) > 10:

    print("Pos10")

  else:

    print("Pos0to10")

  print("Pos\n")

elif int(num) < -10:

  print("Neg")

else:

  print("from -10 to 0")

print("Port")

name = input ()

A = 'Yes' if name != "Test" else 'No'

print (A)

#6 - Циклы For, While, а также операторы

Цикл Do While

Во многих языках программирования есть цикл do..while, который один раз запускается, а для повторного выполнения выполняет проверку условия и далее на её основе либо продолжает выполняться, либо выходить из цикла. В Питон подобного цикла не существует, но это не проблема, ведь создать алгоритм для такого цикла не сложно.

Всегда проверяйте, чтобы в коде не было бесконечных циклов. Результатом их работы становится зависание и прекращение работы программы. Если уж и использовать цикл, то нужно устанавливать интервал обновления в несколько секунд. Ещё можно создать бесконечный цикл и прервать его через оператор break.

Пример реализации своего цикла do..while:

a = 10

while True: # Сперва выполняем цикл

a -= 1

if a == 0: # Далее прописываем проверку

break

В данном примере цикл окажется корректным, ведь break в нужный момент завершит его выполнение.

Цикл while:

i = 0

while i < 10:

  print(i)

  i += 2

Цикл for (позволяет проходиться по массиву, строке):

for j in 'hello world':

  print(j \* 2, end = '')

print('\n')

ВЫВОД:

hheelllloo wwoorrlldd

Без end:

for j in 'hello world':

  print(j \* 2)

print('\n')

ВЫВОД:

hh

ee

ll

ll

oo

ww

oo

rr

ll

dd

for j in 'hello world':

  if j == 'w':

    continue # пропускает одну итерацию - не печатает w

  print(j \* 2, end = '')

print('\n')

ВЫВОД:

hheelllloo oorrlldd

for j in 'hello world':

  if j == 'w':

    break # выходит из цикла

  print(j \* 3, end = '')

ВЫВОД:

hhheeellllllooo

for j in 'hello world':

  if j == 'a':

    break # выходит из цикла

else:

  print("Нет буквы а")

#7 - Списки (list)

В уроке мы изучим списки (list) в языке Python. Списки также часто называют массивами в Питоне. Благодаря спискам мы можем хранить большое количество данных всего в одной переменной.

Списки можно также назвать массивами, так как выполняют они схожую роль. В Python существует множество разновидностей таких "массивов". В уроке мы рассмотрели лишь первый тип, который называется списком и имеет следующие характеристики:

* его элементы можно переопределять;
* в него можно добавлять новые элементы;
* из него можно удалять элементы;
* в качестве ключей идут индексы (числа - 0, 1, 2...).

Это не все характеристики списка, тем не менее это ключевые моменты, которые отличают его от других "массивов" в языке Python.

Для списков существует большое количество функций, которые можно применять для добавления, удаления или же поиска элементов.

В списках можно хранить различную информацию. К примеру, вы можете хранить в нем сведения про всех студентов в группе.

Пример:

lis = ["Андрей", "Иван", "Василий", "Петро", "Максим", "Дима"]

l = [] # пустой список

lis = [1, 56, 'x', 34, 2.34, ['S', 't', 'r', 'o', 'n', 'g']]

print(lis)

ВЫВОД:

[1, 56, 'x', 34, 2.34, ['S', 't', 'r', 'o', 'n', 'g']]

a = [a + b for a in 'list' if a != 's' for b in 'soup' if b != 'u']

print(a)

ВЫВОД:

['ls', 'lo', 'lp', 'is', 'io', 'ip', 'ts', 'to', 'tp']

l = [] # пустой список

l.append(23) # добавляет эл-нт в конец списка

l.append(34)

b = [2, 3]

l.extend(b) # добавляет в конец списка эл-нты др. списка

l.insert (1, 56) # в l[1] - вместо 34 ставит 56 вставляет эл-нт по индексу

l.append(34)

l.remove(34)

# удаляет эл-нт по значению, если эл-нт такой не находится выдает valueerror

# удаляет первый 34

l.pop ()

# удаляет по индексу, если ничего не указывать - удаляет посл.эл-нт

print(l.index(56)) # возвр-т индекс значения

print(l.count(34)) # подсчитывает кол-во эл-тов со знач-ем 34

l.sort ()

# сортирует список на основе какой-либо функции

# если ничего не вводить, то просто по возрастанию

l.reverse() # переворачивает список

# l.clear() - очищает список

print(l)

ВЫВОД:

1

0

[56, 23, 3, 2]

#8 - Индексы и срезы

Индексы

Нумерация в списках начинается с нуля, так как список по большей части это просто массив, то как в обычном массиве отсчет ведется от 0. Первый элемент по индексу будет 0, второй - 1, третий - 2 и так далее. Если мы попытаемся взять несуществующий элемент, то это приведет к ошибке.

Пример:

a = [0, 23, "Hi"] # Список

print (a[4]) # Выдаст ошибку, так как элемента не существует

Удобной функцией языка Python является возможность брать элементы с конца при помощи отрицательных индексов. К примеру, если нам нужен второй элемент с конца, то мы можем записать это так:

a = [0, 23, "Hi", 1.56, 9] # Список

print (a[-2]) # Будет выведено 1.56

Срезы

Срезы позволяют обрезать список, взяв лишь те элементы, которые нужны. Они работают по следующей схеме: list[НАЧАЛО:КОНЕЦ:ШАГ] - list[START:STOP:STEP].

Начало - с какого элемента стоит начать (по умолчанию равно 0);

Конец - по какой элемент мы берем элементы (по умолчанию равно длине списка);

Шаг - с каким шагом берем элементы, к примеру, каждый 2 или 3 (по умолчанию каждый 1).

В срезах один, несколько или даже все параметры могут быть пропущены.

list[::3] # Берем каждый третий элемент

list[2::2] # Начиная со второго элемента берем каждый второй элемент

list[4:6:] # Начиная с 4 элемента берем все элементы по 6 элемент

list[::] # Берем все элементы - по умолчанию list[0:длина списка:1]

print(list[:]) # выведется просто список - все параметры по умолчанию

Также могут быть использованы отрицательные числа для срезов.

print(list[::-1]) # список в обратном порядке выйдет

l = [34, 'sd', 56, 34.34]

print(l[-2::-2])

ВЫВОД:

[56, 34]

#9 - Кортежи (tuple)

Кортежи - это по сути те же самые списки, только с несколькими отличиями. Кортежи нельзя изменять после их создания, а также они занимают чуть меньше памяти, нежели списки.

a = (43, 56, 45.23, 'd') # кортеж

b = [42, 56, 45.23, 'd'] # список

print(a.\_\_sizeof\_\_()) # определяет размер

print(b.\_\_sizeof\_\_())

ВЫВОД в байтах:

56

72

Кортеж- это не список-константа. Например, в программе нельзя изменить отдельные элементы кортежа, но можно изменить весь кортеж. т.е. взять и весь его переопределить.

При работе с кортежами действуют абсолютно все методы, о которых было рассказано в уроке про списки. Единственные методы, которые не будут работать, так это методы, изменяющие кортеж: удаление, добавление элементов, сложение кортежей и тому подобные.

Кортежи можно создать двумя способами:

* при помощи ключевого слова a = tuple ();
* при помощи литерала a = () - пустой кортеж.

a = tuple ("hello world")

print (a)

ВЫВОД:

('h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd')

a = ("hello world")

print (a)

ВЫВОД (всего 1 элемент):

hello world

Их главное преимущество в том, что их нельзя изменять, поэтому ни вы, ни кто-либо другой не сможет как случайно, так и намерено изменить кортеж.

#10 - Словари (dict), а также их методы

Словари отличаются способом их создания, а также индексами. Если в обычном списке в качестве индексов выступают числа (0, 1, 2...), то здесь на их замену приходят "ключи". Так, мы можем создать некий ключ (Студент Федя, например) и добавить к нему целый ряд характеристик (список) в качестве одного элемента. Теперь найти нужного нам студента будет намного проще, так как мы просто будет оперировать ключами, а не числами.

Во многих других языках программирования такие словари зачастую называются ассоциативными массивами, поэтому не редко можно услышать и такое название.

Чтобы создать словарь вы можете воспользоваться следующей конструкцией:

words = {'short': 'Гоша', 'long': 'Георгий'}

Есть 4 разных способа создать словарь:

d = {} # пустой словарь

b = {'name': "Robert", 'age': 12}

print(b['age'])

ВЫВОД:

12

b = {'name': "Robert", 'age': 12}

print(b)

ВЫВОД:

{'name': 'Robert', 'age': 12}

d = {} # пустой словарь

b = {'name': "Robert", 'age': 12}

c = dict(short='dict', longer='dictionary')

c['short'] = 34

# была ошибка - выводил c['short'] = 34 NameError: name 'c' is not defined - ввела с на русском языке

f = dict ([("name", "Susan"), ("age", 13)]) # внутри список, в списке - кортеж

e = dict.fromkeys (['a', 'b'])

j = dict.fromkeys (['a', 'b'], 1)

h = {a : a \*\* 2 for a in range(7)} # а в пределах от 0 до 6 будет возводиться в степень

print("b: ", b)

print("c: ", c)

print("f: ", f)

print("e: ", e)

print("j: ", j)

print("h: ", h)

ВЫВОД:

b: {'name': 'Robert', 'age': 12}

c: {'short': 34, 'longer': 'dictionary'}

f: {'name': 'Susan', 'age': 13}

e: {'a': None, 'b': None}

j: {'a': 1, 'b': 1}

h: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}

person = {'name': {'last\_name': 'Rojkov', 'first\_name': 'Nicolay', 'middle\_name': 'Petrovich'}, 'address': ['London', 'Bright Avenue 11', 'app.4'], 'phone': {'home\_phone': '34-44-57', 'mobile\_phone': '8-965-345-56-75'}}

print(person)

print("Name:\n", person['name'])

print("Mobile phone:\n", person['phone']['mobile\_phone'])

print("Keys:\n", person.keys())

print("Values:\n", person.values())

ВЫВОД:

{'name': {'last\_name': 'Rojkov', 'first\_name': 'Nicolay', 'middle\_name': 'Petrovich'}, 'address': ['London', 'Bright Avenue 11', 'app.4'], 'phone': {'home\_phone': '34-44-57', 'mobile\_phone': '8-965-345-56-75'}}

Name:

{'last\_name': 'Rojkov', 'first\_name': 'Nicolay', 'middle\_name': 'Petrovich'}

Mobile phone:

8-965-345-56-75

Keys:

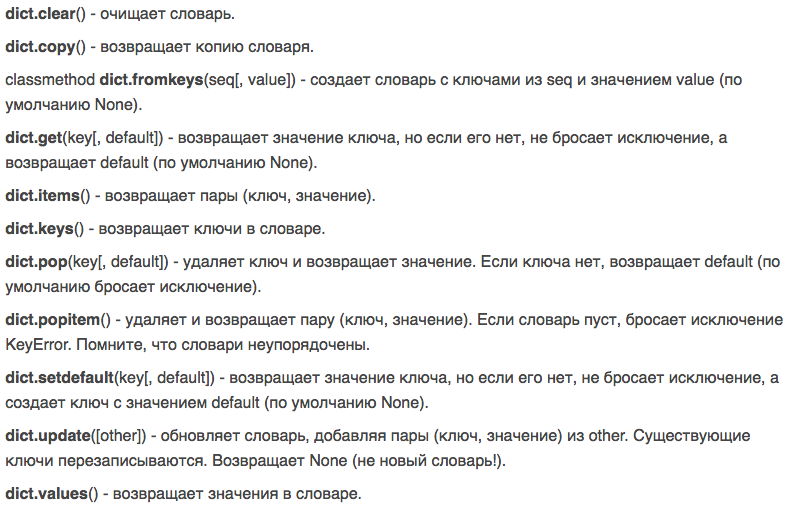
dict\_keys(['name', 'address', 'phone'])

Values:

dict\_values([{'last\_name': 'Rojkov', 'first\_name': 'Nicolay', 'middle\_name': 'Petrovich'}, ['London', 'Bright Avenue 11', 'app.4'], {'home\_phone': '34-44-57', 'mobile\_phone': '8-965-345-56-75'}])

Методы словарей

На фото ниже представлены различные методы, которые можно использовать при работе со словарями:



#11 - Множества (set и frozenset)

Множества также схожи с массивами, но есть несколько отличий. Во-первых, множества создаются в абсолютно случайном порядке каждый раз. Вы можете разместить элементы как вам будет угодно, но они все равно будут расположены впоследствии в случайном порядке. Во-вторых, множества не могут иметь повторяющихся элементов, поэтому все элементы, которые будут одинаковыми не будут выведены повторно.

Их очень удобно использовать, если вы хотите удалить повторяющиеся элементы из списка, например:

some\_list = [12, 56, 91, 12]

set(some\_list) # Результат: 12, 56, 91

a = set () # пустое множество

print (a)

print (type(a))

b = set("hello")

print (b)

print (type(b))

c = {'23', 32} # множества без ключей, если введем ключ, то класс поменяется на словарь

print (c)

print (type(c))

d = {i \*\* 2 for i in range (10)}

print (d)

print (type(d))

e = {}

print (e)

print (type(e))

ВЫВОД:

set()

<class 'set'>

{'e', 'o', 'h', 'l'}

<class 'set'>

{32, '23'}

<class 'set'>

{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}

<class 'set'>

{}

<class 'dict'>

a = set ("Hello")

b = frozenset("Qwerty") # как в кортежах не можем добавлять, удалять и заменять эл-нты

a.add(1)

b.add(1) # не сработает

print (a)

print (b)

ВЫВОД:

Traceback (most recent call last):

File "./pyth.py", line 140, in <module>

b.add(1)

AttributeError: 'frozenset' object has no attribute 'add'

a = ['r', 's', 'w', 'a', 's', 'w']

print (a) # просто список

print (set (a)) # переопределили в множество

a = {32, 45, 43.23, 76}

x = 45

b = 56

c = {67, 12, 90}

d = {32, 45, 76}

e = {32, 45, 76, 43.23}

print (len(a))

print (x in a)

print (b in a)

print (a.isdisjoint(c)) # истина, если не имеют общих эл-нтов

print ("a == d:", a == d)

print ("a == e:", a == e)

ВЫВОД:

4

True

False

True

a == d: False

a == e: True

a = {32, 45, 43.23, 76}

x = {23, 45, 12, 43.23}

# a.intersection\_update(x) # выводит пересечения (одинаковые) числа

# a.difference\_update(x) # выводит разность - все числа множества а, кот-х нет в х

# a.update(x)

# a.symmetric\_difference\_update(x) # строгая дизъюннкция ( исключающее или) - все неодинаковые числа в множествах а и х: есть в а, но нет в х - или есть в х, но нет а

# a.add (23)

# a.remove(32)

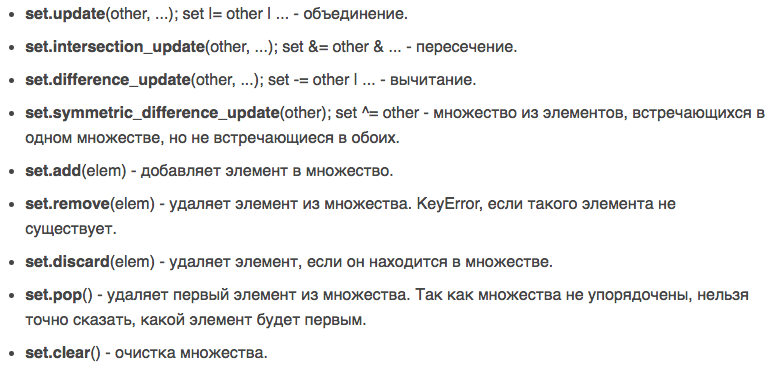
# a.discard(11) # в отличие от add не выдает ошибку, если не находит значение

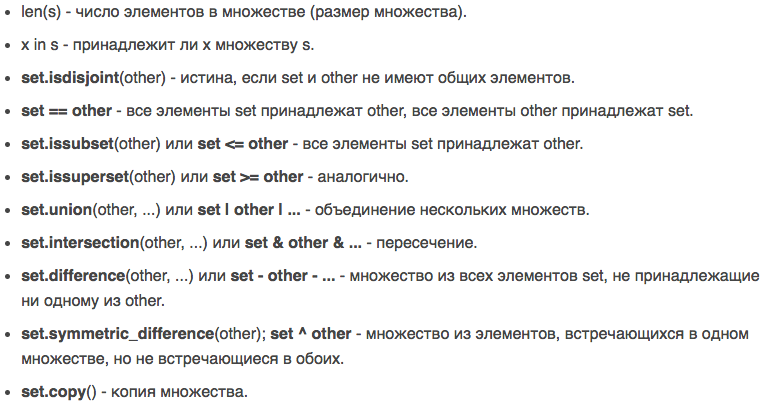
# a.pop() # удаляет первый элемент из множества

# a.clear()

print(a)

Также для множеств существует огромное количество операций, которые приведены ниже:





#12 - Функции (def, lambda, return)

Функции можно назвать небольшими подпрограммами, куда можно вынести повторяющийся код и обращаться к нему, когда это будет нужно. Функции значительно облегчают построение программ, так как нам не надо копировать однотипный код множество раз, а можно просто воспользоваться одной общей функцией.

Многие путают функции и методы и не понимают отличий между ними. На самом деле отличий нет, так как что методы, что функции являются одним и тем же. Функции что записаны вне классов называют функциями, а функции что записаны внутри классов называются методами.

Точно такая же ситуация обстоит с переменным. В классах переменные называются полями, а вне классов - переменными.

def func (x, a):

  res = x + a

  return res

print (func('Hello', ' World!'))

ВЫВОД:

Hello World!

def func (x):

  def add (a):

    return x + a

  return add

test = func(100)

print (test (200))

ВЫВОД:

300

* Это функциональное программирование, детка) Функции могут быть не только блоками инструкции, но и параметром для других функции, которые можно передавать и получать. Это из функционального программирования, называется Каррирование
* Тоже не мог понять, но потом дошло. Гоша ошибся (ИМХО), когда показывал, что параметр test стал функцией func. Это не так. test - это результат функции func, а именно функция add! Если в return будет другая функция или вообще переменная, то test будет равен ей, а никак не функции func! Это важно понять.

В Python функции создаются при помощи ключевого слова def. Каждая функция может иметь какие-либо параметры или же не иметь их вовсе. Функции способны что-либо возвращать в ходе выполнения кода, если это требуется.

def some\_test(): # Функция без параметров

x = 23

return x # Функция возвращает что-либо

def some\_test\_2(a, b, c): # Функция с 3 параметрами

return a \* b \* c # Функция возвращает результат умножение чисел

def print\_something(word, prefix): # Функция с 2 параметрами

print (prefix, "-", word)

pass # Функция ничего не возвращает

Когда функция ничего не возвращает, то необходимо прописывать ключевое слово pass.

Функции могут возвращать другие функции, тем самым вызывая их. Чтобы обратиться к функции необходимо прописать её название и передать параметры, если таковые имеются:

res = some\_test\_2 (2, 67, 12)

В примере выше результат выполнения функции будет помещен в переменную res. Далее с переменной можно работать как с обычным значением в программе.

def func (r, w, y = 2):

# значения по умолч-ю надо передавать в конце иначе б.выходить ошибка

# т.е. def func (r, w = 4, y = 2) никак не def func (r = 3, w, y)

  res = r + w

  res \*= y

  return res

print (func(4,2))

ВЫВОД:

12

def func (\*args): # передали кортеж

  return args

print (func('sd', 45.2, 3))

ВЫВОД:

('sd', 45.2, 3)

def func (\*\*args): # передали словарь

  return args

print (func (a=23, n=56, k=90))

ВЫВОД:

{'a': 23, 'n': 56, 'k': 90}

def func (\*\*args):

  return args

print (func (short='dict', longer='donslmcm'))

ВЫВОД:

{'short': 'dict', 'longer': 'donslmcm'}

Анонимные функции

Если функция не должна выполнять большой объем кода, то можно воспользоваться анонимной функцией. Для этого потребуется функция lambda. Пример создания «lambda» функции:

multiple = lambda a, b: a \* b

multiple(7, 12)

add = lambda x, y: x + y

add2 = lambda x, y: x \* y

print(add(5, 2))

print (add2('q', 4))

print ((lambda u, p: u - p)(8, 3))

ВЫВОД:

7

qqqq

5

Подобная функция не имеет названия, но её можно присвоить к переменной, которую в дальнейшем необходимо вызывать как обычную функцию.

#13 - Исключения (Конструкция try - except)

Что такое исключение?

Предположим, что вы разработали программу «Текстовый редактор». В программе пользователь может создать новый файл, вписать в него данные и далее сохранить файл в системе.

Если код прописан корректно, то никаких ошибок возникать не будет. Но давайте представим ситуацию, что пользователь открыл редактор, открыл нужный файл, записал в него данные, далее вручную удалил файл с компьютера и потом попытался сохранить файл через вашу программу.

При таком раскладе у вас получиться ошибка, которая сломает программу и отобьет любое желание у пользователя работать в вашей программе.

Получается, исключение - это ошибка, что возникает в ходе работы самой программы. Отслеживать такие ошибки при помощи условных операторов не всегда возможно, ведь программа уже запущена, поведение пользователя нам неизвестно заранее, а значит и «ловить» ошибку нам нужно в момент её создания.

Отлов ошибок

Для вышеописанного случая как раз и нужна конструкция try except для отлова ошибок в момент их создания.

При помощи данной конструкции вы можете отслеживать ошибки различных форматов и событий. Можно отследить неверное открытие файла, можно отследить неверное подключение класса, можно отследить неверное написание переменной или функции, да чего уж там, можно отследить даже деление чисел на ноль!

Реализации конструкции try except

Для добавления отслеживания ошибок можно прописать следующий код:

try: # Пробуем что-то сделать

k = 1 / 0 # Деление на ноль

except ArithmeticError: # Отслеживаем на арифметическую ошибку

print ("Найдена ошибка") # Мы нашли ошибку

k = 0 # Наша переменная теперь будет равна 0

print(k) # Просто выводим переменную

В коде выше представлен очень простой пример использования исключений. Не обращайте внимание на его простоту, так как главное это рассмотреть общую конструкцию блока try except.

При использовании такой конструкции, какая бы ошибка не получилась в ходе программы сама программа работать не перестанет и пользователь не получить плохой опыт в её использовании.

Если необходимо отследить несколько классов с ошибками, то можно добавить несколько блоков except. Все они добавляются друг под другом точно также как в условных операторах.

try:

x = int (input ())

except ValueError:

print ("Вы ввели не число")

x = 0

try:

y = int (input ())

except ValueError:

print ("Вы ввели не число")

y = 0

else:

print ("Все верно")

finally:

print ("Выполнится 100%")

try:

res = x / y

except ZeroDivisionError:

print ("Вы ввели 0")

res = 0

print (res)

try:

  x = int(input())

  y = int(input())

  res = x / y

except ZeroDivisionError:

  res = "You can't divide to 0"

except ValueError:

  res = "It's string."

else:

  res = "Okay"

finally:

  print ("Выполнится 100%")

print (res)

#14 - Работа с файлами

Многие языки программирования предоставляют классы для работы с файлами и директориями проекта. Язык Python обладает множеством классов для записи и чтения данных из файлов.

Работа с файлами

При работе с файлами всегда необходимо помнить две вещи:

* Перед началом работы с файлом его необходимо открыть;
* После завершения работы с файлом его необходимо закрыть.

Если файл не открыт или же неверно открыт, то вы не можете полноценно работать с его содержимым.

С закрытием все проще, но и коварнее. Если вы не закроете файл, то программа будет работать верно, тем не менее, чем больше будет открытых файлов, тем больше программа будет перегружена и в какой-то момент она просто зависнет или выключиться.

f = open ('text.txt') # по умолчанию стоит rt, форматы можно совмещать

# f = open ('text.txt', 'rt')

# print (f.read(4)) - выводит 4 символа

# print (f.read()) - выводит все

for line in f:

  print (line)

f.close()

ВЫВОД:

Rom

Nom

Pom

f = open ('text.txt', 'w')

f.write ("Hi, is's me\nNet")

f.close

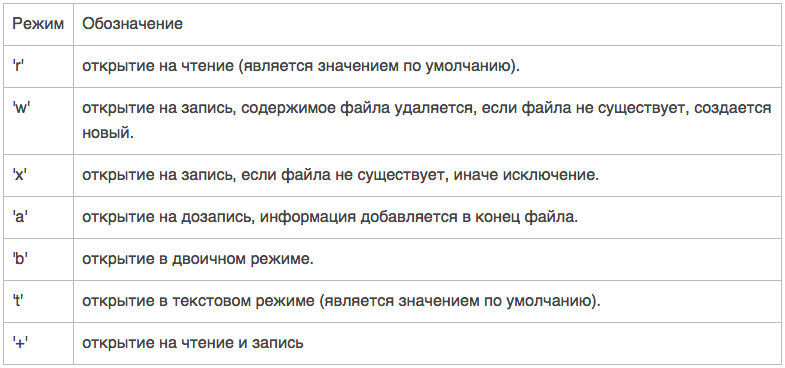
Исключения и файлы

Поскольку не всегда известно будет ли файл в проекте или на компьютере пользователя, то всегда лучше открывать файлы за счёт использования исключений. Выполняйте открытие файлов в блоке try except и тем самым вы обезопасите себя от любых непредвиденных обстоятельств.

Работа с файлами

Для открытия файла существует функция open, которая открывает файл разными способами.

Вот все возможные типы открытия файла:



Для записи текста в файл существует метод write, а для чтения метод read. После того, как с файлом была закончена работа его обязательно необходимо закрыть. Это делается при помощи метода close.

#15 - Менеджеры With ... as

With as – конструкция, которая способна выполнить код сто процентов. Что это означает? Если при работе с файлами мы используем метод open, то дополнительно нам необходимо ещё и закрывать файл, иначе это чревато последствиями.

При работе с With as мы можем быть уверены, что файл точно будет закрыт, даже в случае неправильного выполнения функции.

Пример работы With ... as

with open('test.txt', 'wt', encoding='utf-8') as inFile:

num = int(input())

line = str('1 / ' + str(num) + ' = ' + str(1 / num))

print(line)

inFile.write (line)

With … as - выполняет критические функции. Программа словила ошибку, и все равно закрыла файл и пользователь не имеет уже доступа. Работает и с классами - только выполняет уже другие функции.

with open('test.txt', 'wt', encoding='utf-8') as File: # File - имя переменной

  num = int(input())

  line = str('1 / ' + str(num) + ' = ' + str(1/num))

  print(line)

  File.write(line)

* Критические операторы это Open...Close. Close не нужен, так как с with...as сами завершают прогу.
* encoding = 'utf-8' - восьмибитный юникод, чтобы программа понимала, в какой кодировке читать файл, поскольку, например, в ASCII нет русскоязычных символов, и если в файле они есть, выведется лабуда.

#16 - Модули. Работа с import и from

Модули – это дополнительные файлы, что можно импортировать внутрь других файлов Python. Каждый модуль содержит набор переменных и функций, которые можно использовать после подключения модуля к какому-либо файлу.

Модули можно создавать свои или использовать уже готовые модули Python. С полным перечнем всех стандартных модулей вы можете ознакомиться [по этой ссылке](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_Python).

!!! Модули также могут быть написаны не только на Python, но и на языке С и языке С++ - их можно подключить к своей программе на Python и они будут работать.

Все модули запомнить невозможно, а главное это не нужно. Вам необходимо лишь понимать какой модуль за что отвечает и в случае необходимости вы сможете подсмотреть документацию, которой в интернете полным полно. К примеру, вы хотите создать архив или добавить что-либо в уже существующий? Тогда вам понадобиться модуль ZipFile, который позволяет работать с архивами: создавать их, добавлять файлы, удалять файлы и делать другие вещи.

Таких модулей огромное множество и каждый отвечает за какие-либо функции и новшества.

Чтобы подключить модуль необходимо использовать команду import и указать название модуля.

import math

# модуль (библиотека )с матем. функциями

print (math.pi)

print (math.e)

print (math.cos(1))

ВЫВОД:

3.141592653589793

2.718281828459045

0.5403023058681398

import time, os, math # работа со временем и операц.системой

# считается плохим тоном - подключать библ. ч\з запятую

import time

import os

# улучшает читабельность кода

Наш модуль moduleM.py:

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

if \_\_name\_\_ == "\_\_main\_\_":

  print("Hello")

# глобальная переменная name будет равна main - проверка на работу модуля - чтобы был код, но не срабатывал при импортировании в другом файле

# \_\_name\_\_ – это просто встроенная переменная в Python, которая оценивает имя текущего модуля.

# Когда Python запускает «исходный файл» в качестве основной программы, он устанавливает для специальной переменной (\_\_name\_\_) значение («\_\_main\_\_»).

# Когда вы выполняете функцию main, она читает оператор «if» и проверяет, действительно ли \_\_name\_\_ равно \_\_main\_\_.

# В Python «if\_\_name \_\_ ==« \_\_main\_\_ » позволяет запускать файлы Python как многократно используемые модули или как отдельные программы.

def hi():

  print("Hello World!")

def add(x, y):

  return (x + y)

Наш основной рабочий файл с подключенным модулем:

import time

import os

import random as r # псевдоним для модуля

# если неуверен, что модуль существует, чтобы программа не накрылась

try:

  import nomodule

except ImportError:

  print ("Модуля nomodule не существует")

import moduleM as m

print(time.time())

print (os.getcwd()) # путь - типа pwd

print (os.uname())

print (r.random()) # рандомное число

m.hi()

print (m.add(45,15))

ВЫВОД:

Модуля nomodule не существует

1588258236.4645627

/mnt/d/School21/Docker/02\_bonus[BONUS]

posix.uname\_result(sysname='Linux', nodename='LAPTOP-132SB74K', release='4.4.0-18362-Microsoft', version='#476-Microsoft Fri Nov 01

16:53:00 PST 2019', machine='x86\_64')

0.2010297759196472

Hello World!

60

Пример:

import time # Импорт одного модуля

import random, os # Импорт нескольких модулей

Также вы можете создавать свои модули. Для этого необходимо создать новый файл и по его названию подключать файл как модуль в другие файлы.

Команда From

Чтобы импортировать лишь некоторые методы или же переменные можно использовать команду from. Эта команда записывается следующим образом:

from math import ceil # Импортируем лишь метод ceil

from moduleM import (hi as h, add as a) # ч/з запятую, если несколько функций

h ()

print (a (5, 4))

ВЫВОД:

Hello World!

9

Псевдонимы для модулей

Также при работе с модулями можно использовать псевдонимы. Это позволяет сократить название модуля, если оно слишком большое и вам не хочется писать его каждый раз.

Пример:

* import Template as tem # Теперь можно обращаться к модулю как tem

import random as r # псевдоним для модуля

Файл python.py

import time

import os

import random as r

from module import hi as h, add as a

try:

import nomodule

except ImportError:

print ("Модуля nomodule не существует")

h ()

print (a (45, 15))

Наш модуль module.py

def hi ():

print ("Hello World!")

def add (x, y):

return x + y

if \_\_name\_\_ == "\_\_main\_\_":

print ("Hello")

#17 - Основы ООП Python

ООП - это объектно-ориентированное программирование, в основе которого является создание класса, а также объекта класса. Мы научимся создавать классы и объекты, а также создадим методы и поля для основного класса.

На начальном этапе ООП – это тёмный лес, в котором многое непонятно и слишком усложнено. На самом деле всё вовсе не так. Предлагаем абстрагироваться от специфических (непонятных) определений и познакомиться с ООП простыми словами.

ООП простыми словами

Поскольку на примере все усвоить гораздо проще, то давайте за пример возьмем робота, которого постараемся описать за счёт классов в ООП.

Класс в случае с роботом – это его чертёж. Экземпляром класса (объектом) называет целый робот, который создан точно по чертежу.

Наследование – это добавление полезных опций к чертежу робота. К примеру, берем стандартный чертёж робота и дорисуем к нему лазеры, крылья и броню. Все эти дорисовки мы сделаем в классе наследнике, основной функционал которого взят из родительского класса.

Полиморфизм – это общий функционал для всех роботов и не важно, что каждый робот может очень сильно отличаться друг от друга. К примеру, в главном классе мы указываем возможность передвижения для всех последующих роботов. Далее в классе наследнике мы можем дополнительно указать возможность левитации для робота, в другом же классе укажем возможность передвижения по воде и так далее. Получается, что есть общий функционал что записан в главном чертеже, но его можно переписать для каждого последующего робота (для каждого наследника).

А инкапсуляция является для нас бронёй, защищающей робота. Под пластырем брони находятся уязвимые элементы, вроде проводов и микросхем. После прикрытия брешей с помощью брони (protected или private), робот полностью защищён от внешних вмешательств. По сути, мы делаем доступ ко всем полям лишь за счёт методов, тем самым прямой доступ к полю будет закрыт.

У всех классов методы могут отличаться, как и поля с конструкторами. Каждый класс позволяет создавать любое количество разных объектов, все из них имеют собственные характеристики.

Создание классов

Для создания класса необходимо прописать ключевое слово class и далее название для класса. Общепринято начинать названия классов с буквы в верхнем регистре, но если этого не сделать, то ошибки не будет.

В любом классе можно создавать поля (переменные), методы (функции), а также конструкторы.

Создав новый класс и поместив туда какую-либо информацию, мы можем создавать на основе него новые объекты. Объекты будут иметь доступ ко всем характеристикам класса.

Пример простого класса приведен ниже:

class Book:

pass # Класс может ничего не возвращать

На основе такого класса мы можем создать множество объектов. Каждый объект в данном случае будет представлять из себя конкретную книжку. Для каждого объекта мы можем указать уникальные данные.

Чтобы создать объект нам потребуется следующий код:

obj\_new = Some() # Создание объекта

obj\_second = Some() # Создание 2 объекта

class Person:

  name = "Ivan"

  age = 10

vlad = Person ()

vlad.name = "Vlad"

print (vlad.name)

ivan = Person ()

ivan.age = 45

print (ivan.age)

ВЫВОД:

Vlad

45

class Person:

  name = "Ivan"

  age = 10

  def set(self, name1, age1):

# в каждом методе д.б. аргумент self - обяз.аргумент, содерж-й экземпляр класса, передающийся при вызове метода, без него работать не будет.

# self.name - обращается к Person.name и присваивает ему значение name1

    self.name = name1

    self.age = age1

vlad = Person ()

vlad.set ("Влад", 25) # передали в name1, age1

print (vlad.name + " " + str(vlad.age))

ivan = Person ()

ivan.set ("Иван", 55)

print (ivan.name + " " + str(ivan.age))

class Person:

name = "Ivan"

age = 10

def set(self, name, age):

self.name = name

self.age = age

vlad = Person ()

vlad.set ("Влад", 25)

print (vlad.name + " " + str(vlad.age))

ivan = Person ()

ivan.set ("Иван", 56)

print (ivan.age)

<https://younglinux.info/oopython/encapsulation> - Инкапсуляция

<https://habr.com/ru/post/444338/> - Инкапсуляция в Python 3

<https://pythonworld.ru/osnovy/inkapsulyaciya-nasledovanie-polimorfizm.html> - Инкапсуляция, наследование, 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

**>>>** b.\_B\_\_private()

Это приватный метод!

Python 3 предоставляет 3 уровня доступа к данным:

* публичный (public, нет особого синтаксиса, publicBanana);
* защищенный (protected, одно нижнее подчеркивание в начале названия, \_protectedBanana);
* приватный (private, два нижних подчеркивания в начала названия, \_\_privateBanana).

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'

#18 - Наследование, инкапсуляция, полиморфизм

Наследование

Наследование является одним из ключевых понятий ООП. За счёт наследования можно создать один общий класс (класс родитель) и создать множество других классов (классы наследники), что будут наследовать все поля, методы и конструкторы из главного класса.

За чем использовать наследование?

Предположим, что у нас есть один большой класс «Транспорт». В классе описываются базовые характеристики для всех транспортных средств:

* поля: скорость, вес, запас хода и тому подобное;
* методы: получение информации из полей, установка новых значений;
* конструктор: пустой и по установке всех полей.

На основе класса мы спокойно можем создать объект легковой машины, объект грузовика, объект самолета и так далее. У всех объектов будут одинаковые характеристики и методы.

Мы явно понимаем, что у объекта машина и самолёт будут разные поля и характеристики. Как можно поступить:

Можно создать два отдельных класса: «Car» и «Airplane». В каждом классе будут все методы, поля и конструкторы повторно переписанные из класса «Транспорт», а также будут новые методы, что важны только для конкретного класса;

Можно создать два класса наследника: «Car» и «Airplane». Оба класса будут наследовать всё от класса «Транспорт» и при этом будут содержать свои дополнительные функции. Таким образом повторения кода не будет и код станет меньше и чище.

Создание классов наследников

Для создания класса наследника требуется создать класс и указать наследование от главного класса.

Пример класса наследника:

class Cars:

wheels = 4 # Общее значение для всех объектов,

# так как все машины имеют колеса

class BMW (Cars):

is\_m\_serias = true # Является ли модель "М" серии?

# Переменная нужна только в классе BMW

Инкапсуляция

Инкапсуляция позволяет ограничить доступ к какой-либо функции в классе. Благодаря такому подходу злоумышленники или же мы сами не сможем случайно или намерено вызвать, или изменить метод.

* без подчеркивания аналог public, одно подчеркивание аналог protected (тоесть на уровне объекта юзать нельзя, но можно на уровне класов наследников), двойное подчеркивание аналог private (использование только на уровне текущего класса).

class Person:

  name = "Ivan"

  age = 10

  def \_\_set(self, name1, age1):

    self.name = name1

    self.age = age1

class Student (Person):

  course = 1

igor = Student()

igor.\_Person\_\_set ("Igor", 19)

igor.course = 2

print (igor.course)

vlad = Person ()

vlad.\_Person\_\_set ("Влад", 25)

print (vlad.name + " " + str(vlad.age))

ivan = Person ()

ivan.\_Person\_\_set ("Иван", 55)

print (ivan.name + " " + str(ivan.age))

ВЫВОД:

2

Влад 25

Иван 55

Пример:

class Some:

def \_printWords(self):

print ("Попробуй меня вызвать")

obj = Some()

obj.\_printWords() # Вызов функции ничего не даст

При помощи двойного подчеркивания эффект защиты усиливается, поэтому вызвать функцию будет невозможным.

Полиморфизм

Полиморфизм позволяет изменять функции в классах наследниках. Пример:

class Cars:

def nothing(self): # Пустая функция

pass

class BMW (Cars):

def nothing(self, word):

print (word + "!") # Функция теперь будет работать по новому

a = BMW()

a.nothing("Some")

#19 - Конструкторы, переопределение методов

Конструкторы позволяют задать некие характеристики для объекта сразу же при его создании. К примеру, у вас есть несколько переменных, которые точно должен иметь объект. Вы можете создать конструктор и указать несколько параметров, которые будут переданы при создании объекта.

В одном классе может быть неограниченное количество конструкторов и сам интерпретатор будет понимать к какому конструктору вы обращаетесь. Чтобы создать конструктор необходимо использовать ключевое слово \_\_init\_\_.

* Метод init просто упрощает код, так как употребив его вместо set вы сможете не добавлять отдельную строчку Igor.set(данные), а добавлять эти данные сразу после присваивания переменной класса.

class Person:

  name = "Ivan"

  age = 10

  def \_\_init\_\_(self, name1, age1):

    self.name = name1

    self.age = age1

  def set(self, name1, age1):

    self.name = name1

    self.age = age1

class Student (Person):

  course = 1

  def set(self, name1, age1, course1):

    self.name = name1

    self.age = age1

    self.course = course1

igor = Student("Igor", 19)

# igor.\_Person\_\_set ("Igor", 19)

igor.set("Игорь", 23, 5)

print ("Name: ", igor.name, ", age: ", igor.age, ", course: ", igor.course)

vlad = Person ("Влад", 25)

# vlad.\_Person\_\_set ("Влад", 25)

print (vlad.name + " " + str(vlad.age))

ivan = Person ("Иван", 55)

# ivan.\_Person\_\_set ("Иван", 55)

print (ivan.name + " " + str(ivan.age))

Пример класса с конструктором:

class Cars:

wheels = 0

marka = ""

def \_\_init\_\_(self, wheels, marka):

self.wheels = wheels

self.marka = marka

bmw = Cars(4, "X3") # Сразу добавили характеристики

print (bmw.wheels) # Результат - 4

#20 - Декораторы функций

Декораторы, обертки для функций, бывают очень удобны во многих случаях. Вы можете задать вопросы, а зачем оборачивать какую-либо функцию с добавлением нового кода до и после её выполнения, если можно просто вписать это же в саму функцию?

Давайте на секунду представим, что у нас большая программа и мы используем какую-либо функцию множество раз. Нам не нужно её менять, так как её использует много других подпрограмм. Тем не менее, нам нужно добавить к ней код, который должен сработать до или после её вызова. Тут на помощь и приходят декораторы, которые способны выполнить эту задачу. При этом, декораторы замедляют работу функции.

Одну функцию можно оборачивать несколько раз.

def decorator(func):

  def wrapper():

    print("Код до выполнения функции")

    func()

    print("Код, кот-й сработал после функции")

  return wrapper

def qwe(func):

  def wrapper():

    print("Код до выполнения функции 2")

    func()

    print("Код, кот-й сработал после функции 2")

  return wrapper

@decorator # decorator оборачивает qwe, qwe оборачивает функцию show

@qwe

def show():

  print("Я обычная функция")

show()

# dec = decorator(show)

# dec()

ВЫВОД:

Код до выполнения функции

Код до выполнения функции 2

Я обычная функция

Код, кот-й сработал после функции 2

Код, кот-й сработал после функции

==

def decorator(func):

  def wrapper():

    print("Код до выполнения функции")

    func()

    print("Код, кот-й сработал после функции")

  return wrapper

def show():

  print("Я обычная функция")

show()

dec = decorator(show)

dec()

ВЫВОД:

Я обычная функция

Код до выполнения функции

Я обычная функция

Код, кот-й сработал после функции

#21 - Заключительный урок

В ходе видео курса мы с вами изучили основные моменты в языке Python. Язык Python состоит из множества библиотек, которые расширяют его функционал и добавляют новые возможности. Изучить все библиотеки невозможно, а главное не нужно.

Вам необходимо выбрать определённую сферу, в которой хотите работать и тогда уже изучать специализированные библиотеки для вашей сферы. К примеру, если вы хотите создавать dt, программы, то вам стоит ознакомиться с фреймворком Django и научиться с ним работать.