Оглавление

Неделя 1

Введение в Python

1.1. Введение

Python --- это простой, гибкий, популярный, и, что самое главное, востребованный язык программирования, на котором можно написать практически всё. Создавать веб-проекты, заниматься машинным обучением и анализом данных, автоматизировать задачи системного администрирования и многое другое. Вас ждёт путь от знакомства с интерпретатором Python, структурами данных и функциями до разработки собственного сетевого клиент-серверного приложения. В процессе изучения курса вы погрузитесь в мир объектно-ориентированного программирования, познакомитесь с многопоточным и асинхронным программированием. Авторы курса стремились покрыть все темы, необходимые разработчику каждый день. Кроме того, курс содержит большое количество практических примеров и задач.

1.2. Первые шаги

1.2.1. О языке

В конце 80-х годов прошлого века сотрудник голландского центра математики и информатики Гвидо ван Россум решил создать свой собственный язык программирования, отличающийся простотой и выразительностью. В 1991 году он опубликовал исходники языка, который получил название Python --- в честь популярного телешоу «Летающий цирк Монти Пайтона». Гвидо удалось создать интерпретируемый язык с динамической типизацией и автоматической сборкой мусора, на котором по-настоящему приятно писать код. Также большим плюсом Python-а является то, что за годы существования языка для него было написано множество готовых библиотек на все случаи жизни.

Важным событием в истории Python-а был момент, когда разработчики выпустили третью версию, обратно несовместимую со второй, чтобы решить некоторые архитектурные недостатки второй версии языка. Обратная несовместимость привела к тому, что до сих пор многие продакшн-системы используют Python версии 2. Тем не менее, в 2020-м году официальная поддержка Python 2 прекратится, поэтому новые проекты стоит начинать именно на Python 3. В курсе будет рассказано о Python 3, а именно --- Python 3.6.

Стоит отметить, что Python --- это название спецификации языка, основная его реали-

зация написана на языке C и называется CPython. Есть и другие реализации спецификации языка, однако в данном курсе под словом Python имеется в виду именно реализация на C --- CPython. Исходный код CPython открыт и доступен по ссылке на github. Также проект Python имеет великолепную документацию с большим количеством примеров, доступную на официальном сайте.

Прежде чем начать программировать, осталось решить два вопроса: как установить Python 3 в систему и в каком редакторе писать код на Python.

1.2.2. Начинаем программировать

Python --- интерпретируемый язык программирования, и для него есть интерактивный интерпретатор, доступный из командной строки. Для его запуска, нужно набрать в командной строке команду `python` или `python3` (в зависимости от вашей системы и способа установки интерпретатора).

В интерактивной оболочке Python-а можно быстро проверить какую-нибудь гипотезу, поэкспериментировать или просто воспользоваться ей как калькулятором. Также можно напечатать какую-нибудь строку, используя для этого встроенную функцию `print`:

```
>>> 1 + 1
2
>>> print("Hey")
Hey
```

Чтобы получить справку по какой-либо функции, можно воспользоваться функцией `help()`, например, так:

```
>>> help(print)
```

Чтобы выйти из интерпретатора:

```
>>> exit()
```

В реальных проектах код на Python-е пишут не в интерактивном интерпретаторе, а в файлах. Давайте создадим простейший файл, который будет содержать код на Python-е и запустим его интерпретатором Python. Файл должен иметь расширение `.py` (например, `example.py`). Внутри файла напишем простейшую инструкцию `print("hello")` и сохраним его. Исполним файл, набрав в командной строке:

```
python3 example.py
```

Неотъемлемой частью языка являются переменные. Переменная — это то, что позволяет сохранить результат выполнения выражения для того, чтобы использовать его в дальнейшем. Для того, чтобы объявить переменную, используют следующий код:

```
num = 1    # присвоили переменной num значение 1,
         # знак = является оператором присваивания
```

В Python имена переменных могут содержать буквы, цифры и символ нижнего подчеркивания. При этом начинаться переменная должна либо с буквы, либо с символа нижнего подчеркивания. Если переменная состоит из нескольких слов, её принято называть `snake_case`-ом.

Блоки кода в Python-е отделяются с помощью отступов. Для примера напишем небольшую программу в интерактивном интерпретаторе, которая будет печатать на экран числа от 0 до 3. Блок кода внутри конструкции for отделён с помощью четырех пробелов.

```
>>> for i in range(4):
...     print(i)
0
1
2
3
```

Комментарии в Python-е пишутся с использованием специального символа #, после которого идёт текст комментария. В программах также встречаются многострочные комментарии:

```
print("Hello") # это комментарий
"""
А это
многострочный
комментарий.
"""
```

Однако, многострочный комментарий не игнорируется интерпретатором Python. Это строковый литерал, который можно увидеть как часть документации функции или класса. Этот строковый литерал становится частью объекта, к которому можно достучаться, используя специальное свойство `__doc__` у объекта. Например, функция `print` содержит атрибут `__doc__`, в котором содержится строка документирования, определённая у этой функции.

```
>>> print.__doc__
"print(value, ..., sep=' ', end='\\n', file=sys.stdout,
flush=False)\n\nPrints the values to a stream, or to sys.stdout by
default.\nOptional keyword arguments:\nfile:  a file-like object
(stream); defaults to the current sys.stdout.\nsep:   string inserted
between values, default a space.\nend:   string appended after the
last value, default a newline.\nflush: whether to forcibly flush the
stream."
```

1.3. Базовые типы и конструкции

1.3.1. Базовые типы: численные типы

Переменные в Python-е бывают разных типов. Например, тип `int` используется для целых чисел.

```
num = 13
print(type(num))
```

```
<class 'int'>
```

Функция `type()` из этого примера возвращает тип переменной.

Python умеет работать с числами с плавающей точкой. Такие переменные имеют тип `float`, целая и дробная части отделяются точкой. Также Python поддерживает экспоненциальную нотацию записи вещественного числа.

```
num = 13.4
print(num)

# 1.5 умножить на 10 в степени 2
num = 1.5e2
print(num)
```

```
13.4
150.0
```

Python поддерживает конвертацию типов.

```
num = 150.2
print(type(num))
```

```
<class 'float'>
```

```
num = int(num)
print(num, type(num))

num = float(num)
print(num, type(num))
```

```
150 <class 'int'>
150.0 <class 'float'>
```

Обратите внимание, что в процессе конвертации переменной `num` мы потеряли дробную часть вещественного числа.

Python умеет работать с комплексными числами:

```
num = 14 + 1j

print(type(num))
print(num.real)
print(num.imag)
```

```
<class 'complex'>
14.0
1.0
```

Также в Python есть модули `decimal` (позволяет работать с вещественными числами с фиксированной точностью) и `fractions` (позволяет работать с рациональными числами, т.е. дробями).

Как мы уже знаем, Python поддерживает арифметические операции с числами. При этом сумма или разность двух целых чисел будет целой, а если хотя бы одно из чисел вещественное --- результат также будет вещественным. Результат деления (обозначается символом `/`) всегда вещественный. Также существуют следующие операции:

```
>>> 4 * 5.25 # умножение
21.0
>>> 2 ** 4 # возведение в степень
16
>>> 10 // 3 # целочисленное деление
3
```

В более сложных выражениях можно использовать скобки, причём порядок операций в Python-е соответствует правилам арифметики.

Python также поддерживает побитовые операции.

```
x = 4
y = 3

print("Побитовое или:", x | y)
print("Побитовое исключающее или:", x ^ y)
print("Побитовое и:", x & y)
print("Битовый сдвиг влево:", x << 3)
print("Битовый сдвиг вправо:", x >> 1)
print("Инверсия битов:", ~x)
```

```
Побитовое или: 7
Побитовое исключающее или: 7
Побитовое и: 0
Битовый сдвиг влево: 32
Битовый сдвиг вправо: 2
Инверсия битов: -5
```

Для примера рассмотрим следующую задачу. Задача: найти расстояние между двумя точками в декартовых координатах. Решение:

```
x1, y1 = 0, 0
x2 = 3
y2 = 4

distance = ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5
print(distance)
```

```
5.0
```

Когда одной переменной присваивается значение другой переменной, необходимо помнить об изменяемых и неизменяемых типах в языке Python. Об этом будет подробнее сказано в дальнейшем.

1.3.2. Базовые типы: логический тип

В Python существует логический тип `bool`. Переменные этого типа могут принимать значение либо `True`, либо `False`. Логические типы нужны тогда, когда необходимо проверить на истинность некоторое выражение.

```
print(3 > 4)
print(3 <= 3)
print(6 >= 6)
print(6 < 5)
```

```
False
True
True
False
```

Из переменных можно составлять логические выражения, используя логические операторы:

```
x and y # логическое и
x or y # логическое или
not y # логическое отрицание
```

Так реализуются составные логические выражения:

```
x, y, z = True, False, True
result = x and y or z
print(result)
```

```
True
```

Задача: определить, является ли год високосным. Год является високосным если он кратен 4, но при этом не кратен 100, либо кратен 400. С использованием логических выражений решений реализуется в три строчки.

```
year = 2017
is_leap = year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)
print(is_leap)
```

```
False
```

Эту задачу мы могли решить ещё проще --- всего в две строчки, используя модуль стандартной библиотеки `calendar`. (Python предоставляет очень много возможностей в своей стандартной библиотеке.) Внутри модуля `calendar` есть нужная нам функция `isleap`. Проверим:

```
import calendar

print(calendar.isleap(1980))
```

```
True
```

1.3.3. Базовые типы: строки и байтовые строки

Строка – это неизменяемая последовательность юникодных символов.

```
example_string = "Курс про Python на Coursera"
print(example_string)
```

```
Курс про Python на Coursera
```

```
print(type(example_string))
```

```
<class 'str'>
```

Также в Python-е есть концепция сырых строк. Мы можем с помощью сырых строк объявить строку и не экранировать специальные символы, которые находятся внутри этой строки.

```
example_string = "Файл на диске c:\\\\"
print(example_string)

example_string = r"Файл на диске c:\\"
print(example_string)
```

```
Файл на диске c:\\
Файл на диске c:\\
```

Как объединить 2 строки в одну?

```
"Можно" + " просто " + "складывать" + " строки"
```

```
'Можно просто складывать строки'
```

Важно помнить, что строки --- неизменяемый тип.

Срезы строк (`<string>[start:stop:step]`) удобны для того, чтобы из строки вырезать какую-то подстроку. Синтаксис срезов посмотрим на примере:

```
example_string = "Курс про Python на Coursera"
example_string[9:15]
```

```
'Python'
```

У строк есть методы: `count()`, `capitalize()`, `isdigit()` и т.д. Оператор `in` позволяет проверить наличие подстроки в строке:

```
"3.14" in "Число Пи = 3.1415926"
```

```
True
```

Выражение `for .. in` позволяет итерироваться по строке:

```
example_string = "Привет"
for letter in example_string:
    print("Буква", letter)
```

```
Буква П
Буква р
Буква и
Буква в
Буква е
Буква т
```

К строкам можно применить конвертацию типов. Например, мы можем преобразовать вещественное число в переменную типа `str`:

```
num = 999.01

num_string = str(num)

print(type(num_string))
num_string
```

```
<class 'str'>
'999.01'
```

Если мы попробуем преобразовать непустую строку к логическому типу, получим `True`. Пустая же строка (не содержащая ни одного символа) при конвертации в логический тип возвращает `False`.

Существует несколько способов форматирования строк. 1-ый способ форматирования:

```
template = "%s — главное достоинство программиста. (%s)"
template % ("Лень", "Larry Wall")
```

```
'Лень — главное достоинство программиста. (Larry Wall)'
```

2-ой способ:

```
"{} не лгут, но {} пользуются формулами. ({})".format(
    "Цифры", "лжецы", "Robert A. Heinlein"
)
```

Еще способ:

```
"{num} Кб должно хватить для любых задач. ({author})".format(
    num=640, author="Bill Gates"
)
```

И ещё один (возможно, самый удобный) свособ форматирования строк, доступный начиная с версии Python 3.6:

```
subject = "оптимизация"
author = "Donald Knuth"

f"Преждевременная {subject} — корень всех зол. ({author})"
```

```
'Преждевременная оптимизация — корень всех зол. (Donald Knuth)'
```

Модификаторы форматирования позволяют указывать, как именно будет выводиться в строке какая-либо переменная. Посмотрим на примере:

```
num = 8
f"Binary: {num:#b}"  # число будет выводиться в двоичном представлении
```

```
'Binary: 0b1000'
```

```
num = 2 / 3
print(num)

print(f"{num:.3f}") # выводим число с точностью до 3-го знака
```

```
0.6666666666666666
0.667
```

Модификаторов форматирования очень много, подробнее про них можно прочесть в документации.

Встроеннная функция `input()` позволяет получить ввод пользователя в виде строки:

```
name = input("Введи свое имя: ")

f"Привет, {name}!"
```

```
Введи свое имя: Александр
'Привет, Александр!'
```

Байт --- минимальная единица хранения и обработки цифровой информации. Последовательность байт представляет собой какую-либо информацию (текст, картинку, мелодию...) Байтовая строка --- это неизменяемая последовательность чисел от 0 до 255. b-литерал для объявления байтовой строки:

```
example_bytes = b"hello"
print(type(example_bytes))
```

```
<class 'bytes'>
```

Давайте убедимся, что мы действительно получили байтовую строку:

```
for element in example_bytes:
    print(element)
```

```
104
101
108
108
111
```

Однако, если мы, например, попытаемся преобразовать строку `"привет"` в байты, мы получим ошибку. Здесь строка состоит из юникодных символов, которые преобразуются в байты при помощи кодировок. Самая известная кодировка --- это UTF-8. С помощью Python мы можем преобразовать строку в байтовую строку с использованием кодировки UTF-8.

```
example_string = "привет"
print(type(example_string))
print(example_string)
```

```
<class 'str'>
привет
```

```
encoded_string = example_string.encode(encoding="utf-8")
print(encoded_string)
print(type(encoded_string))
```

```
b'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'
<class 'bytes'>
```

Можно проверить, что 2 первых символа `\xd0\xbf` действительно кодируют букву "п".

Чтобы декодировать байтовую строку обратно в строку, можно воспользоваться методом `decode`:

```
decoded_string = encoded_string.decode()
print(decoded_string)
```

```
привет
```

Метод `decode` принимает параметром кодировку, но значение `'utf-8'` --- это значение по умолчанию, поэтому мы ничего дополнительно не прописываем.

1.3.4. Базовые типа: объект None

Объект None является единственным значением специального типа NoneType, который используется для того, чтобы подчеркнуть отсутствие значения. Если вы знаете язык C, то вы можете рассматривать None как эквивалент нулевому указателю. Важно отметить, что при преобразовании None к типу `bool` всегда получается `False`.

Переменная со значением None может пригодиться, когда необходимо отличать её от нуля. Например, обозначим некоторой переменной доход от продаж. Он будет равен нулю, если мы ничего не заработали, и None, если даже не начали продавать.

В условиях сравнения можно проверять, что некоторая переменная является None:

```
if income is None:
    print("Еще не начинали продавать")
elif not income:
    print("Ничего не заработали")
```

Важно отметить, что мы проверяем равенство переменной `income` объекту None с помощью оператора `is`. Это идиоматичный способ проверки соответствия переменной объекту None в Python. Писать `if income == None` неидиоматично в Python, потому что в Python оператор == можно переопределить с помощью магических методов, о которых будет сказано в дальнейшем.

Вы часто увидите None в именованных аргументах функций или методах классов как значение по умолчанию --- это обычно применяется тогда, когда аргумент является не обязательным --- функция или метод будут работать и тогда, когда аргумент явным образом не передали при вызове. Также часто этим значением инициализируют атрибуты экземпляров классов, чтобы потом в нужный момент переопределить каким-то значением. Про функции и классы будет рассказано на следующих неделях курса.

1.3.5. Конструкции управления потоком

Для проверки условий и выполнения соотвествующих действий используется конструкции `if-else` со следующим синтаксисом:

```
company = "google.com"

if "my" in company:
    print("Условие выполнено!")
else:
    print("Условие не выполнено!")
```

```
Условие не выполнено!
```

Оператор `elif` используется, когда нужно проверить несколько разных условий друг за другом:

```
company = "google.com"

if "my" in company:
    print("Подстрока my найдена")
elif "google" in company:
    print("Подстрока google найдена")
else:
    print("Подстрока не найдена")
```

```
Подстрока google найдена
```

Можно писать и так:

```
score_1 = 5
score_2 = 0

winner = "Argentina" if score_1 > score_2  else "Jamaica"

print(winner)
```

```
Argentina
```

Оператор `while` позволяет выполнять блок кода до тех пор, пока выполняется условие:

```
i = 0

while i < 100:
    i += 1

print(i)
```

```
100
```

Встроенный объект `range` позволяет итерироваться по целым числам:

```
for i in range(3):
    print(i)
```

```
0
1
2
```

Мы не увидели число 3, т.к. объект `range()` не включает в себя последнее значение.

Более сложный пример --- задача, которую Фридрих Гаусс однажды очень быстро решил на уроке математики. С помощью Python мы можем решить её не менее быстро:

```
result = 0

for i in range(101):
    result += i

print(result)
```

```
5050
```

Можно инициализировать объект `range()` с двумя аргументами --- чтобы указать, с какого числа начинать итерироваться и каким закончить:

```
for i in range(5, 8):
    print(i)
```

```
5
6
7
```

Если же проинициализировать `range()` с тремя аргументами, последним из них будет шаг итерации:

```
for i in range(1, 10, 2):
    print(i)
```

```
1
3
5
7
9
```

Также есть возможность итерироваться по числам в порядке уменьшения, используя отрицательное значение шага:

```
for i in range(10, 5, -1):
    print(i)
```

```
10
9
8
7
6
```

Для циклов существует ещё несколько полезных комманд: `pass` (определяет пустой блок, который ничего не делает), `break` (позволяет выйти из цикла досрочно) и `continue` (перейти к следующей итерации цикла без выполнения оставшихся инструкций в блоке).

1.3.6. Пример на управление потоком

Рассмотрим использование циклов на примере.

```
import random

number = random.randint(0, 100)

while True:
    answer = input('Угадайте число: ')
    if answer == "" or answer == "exit":
        print("Выход из программы")
        break

    if not answer.isdigit():
        print("Введите правильное число")
        continue

    answer = int(answer)

    if answer == number:
        print('Совершенно верно!')
        break

    elif answer < number:
        print('Загаданное число больше')
    else:
        print('Загаданное число меньше')
```

```
Угадайте число: ора
Введите правильное число
Угадайте число: exit
Выход из программы
```

Кратко рассмотрим, как написать подобную программу в среде разработки Pycharm. Сначала в Pycharm создаём новый проект, прописываем путь до проекта и до питоновского интерпретатора. В проекте создаём новый файл с расширением `.py`. Откроется редактор, в котором можно писать код.

1.4. Организация кода и окружение

1.4.1. Модули и пакеты

В Python разделение кода осуществляется с помощью модулей и пакетов. Модуль в Python --- это обычный файл с расширением `.py`, который содержит определения переменных, функций, классов, и который также может содержать импорты других модулей. Импорт выполняется стандартной командой `import`.

Python импортирует модуль и выполняет все инструкции, которые в этом модуле на верхнем уровне определены.

Иногда модулей недостаточно, используются более крупные объединения --- пакеты. Пакеты в Python --- это директория, которая содержит один или больше модулей. Внутри пакетов могут содержаться другие пакеты. Внутри директории с пакетом должен находиться файл `__init__.py` (обычно пустой). Этот файл выполняется каждый раз, когда мы импортируем пакет.

Конструкция `from mymodule import my_function` позволяет импортировать только определённые объявления из модуля. Чтобы импортировать все объявления из модуля, пишут `from mymodule import *`.

Установка сторонних пакетов производится с помощью утилиты pip, а описания библиотек можно посмотреть на сайте pypi.org.

1.4.2. Виртуальное окружение (Virtualenv). Установка и запуск Jupyter Notebook

Чтобы поставить сторонний пакет на операционную систему, пользуются утилитой `pip`. В командной строке можно набрать `pip install` и название сторонней библиотеки, которую вам хочется поставить.

Хорошей практикой при программировании на Python является использование виртуальных окружений. Виртуальное окружение в Python --- это окружение, которое позволяет изолировать зависимости для определённого проекта. Обычно для каждого проекта создаётся своё виртуальное окружение, в которое ставятся все пакеты, используемые в данном проекте.

Чтобы создать виртуальное окружение, воспользуемся модулем venv. Синтаксис: `python3 -m venv` и далее название директории, в которой будет создано новое виртуальное окружение. Обычно папка с виртуальным окружением создаётся рядом с папкой проекта.

Виртуальное окружение активируется следующей командой:

```
source <название директории с виртуальным окружением>/bin/activate
```

После этого в виртуальное окружение можно устанавливать пакеты известной нам командой `pip install`.

Например, поставим известное приложение Jupyter Notebook. Jupyter Notebook --- очень распространённое приложение, которое применяется повсеместно разработчиками на Python, а также учёными для презентации своей работы и интерактивного запуска кода.

```
pip install jupyter
```

Вместе с `jupyter` установится полезный пакет `ipython` --- расширенная версия интерактивного интерпретатора Python, которая может служить его альтернативой.

Чтобы запустить Jupyter Notebook, воспользуйтесь командой `jupyter-notebook`. Это приложение запускается в веб-браузере. Внутри него можно создавать "ноутбуки", где в ячейках можно писать код и интерактивно его исполнять.

1.4.3. Пример. Пишем программу.

Ресурс `freegeoip.net` позволяет получить по IP-адресу информацию о нашем положении. Давайте получим эту информацию с помощью Python и напечатаем на экран. Для этого нужно создать рабочую директории, а в ней --- папку с виртуальным окружением и затем его активировать.

Для правильной работы программы необходимо установить пакет `requests` --- пакет для работы с http-запросами.

```
import requests

def get_location_info(): # объявление функции
    return requests.get("http://freegeoip.net/json/").json()

if __name__ == "__main__":
# чтобы программа работала только тогда,
# когда мы запускаем её интерпретатором Python
    pprint(get_location_info())
```

1.4.4. Объектная структура в Python

Когда мы создаем переменную в языке Python, мы не просто присваиваем переменной значение, а создаём связь имени переменной с объектом. У переменных каждого типа существуют методы, например, у переменной типа `int` есть специальный метод `__add__` которые можно перечислить с помощью команды `dir()`.

```
num = 13
num.__add__(2)
```

```
15
```

```
print(dir(num))
```

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__',
'__class__', '__delattr__', '__dir__', '__divmod__', '__doc__',
'__eq__', '__float__', '__floor__', '__floordiv__', '__format__',
'__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
'__index__', '__init__', '__init_subclass__', '__int__',
'__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__',
'__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__',
'__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
'__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__',
'__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__',
'__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__',
'__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes',
'imag', 'numerator', 'real', 'to_bytes']
```

Как мы видим, у переменной num огромное множество методов. Напрашивается вопрос о том, как вообще переменные могут обладать методами. Ответ --- в особенности устройства языка Python. Переменные могут обладать методами только потому, что на самом деле они являются объектами в языке Python. На C-уровне определена структура PyObject:

```
typedef struct _object {
    _PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;          // Счетчик ссылок
    struct _typeobject *ob_type;   // Указатель на тип объекта
} PyObject;
```

Эта структура содержит два важных поля. Поле ob_ refcnt --- счётчик ссылок, когда мы создаём новую связь переменной с объектом, он увеличивается на единицу. Если какая-то связь становится не нужна, соответствующий счётчик ссылок на единицу уменьшается. Когда он достигает нуля, объект удаляется из памяти. Так в Python организована автоматическая сборка мусора. Второе важное поле --- указатель на тип объекта. Этот указатель может менять своё значение в процессе работы программы, и именно таким образом в Python реализована динамическая типизация.

Также есть структура PyVarObject, которая добавляет в структуру PyObject поле ob_ size. Это поле может, например, использоваться как контейнер для длинуы строки (чтобы каждый раз её не расчитывать). В различных типах это поле используется поразному.

```
typedef struct {
    PyObject ob_base;
    Py_ssize_t ob_size; // Кол-во элементов в переменной части
} PyVarObject;
```

Также на C-уровне определены два макроса, которые соответствуют PyObject и PyVarObject. Эти макросы используются для того, чтобы создавать новые объекты в Python. Все встроенные типы в Python расширяют структуры PyObject, либо PyVarObject.

```
#define PyObject_HEAD  PyObject ob_base;
#define PyObject_VAR_HEAD  PyVarObject ob_base;

typedef struct PyMyObject {
   PyObject_HEAD
   ...
}
//или
typedef struct PyMyObject {
   PyObject_VAR_HEAD
   ...
}
```

Так реализованы не только какие-то простые типы (int, float, boolean), но и модули, функции и классы. Все они являюся объектами со своими методами и атрибутами. Из этого следует, то что в Python-е любой объект можно присвоить переменной или передать как аргумент в функцию.

1.4.5. Байткод

При запуске функций, импортированных из модулей и пакетов, Python создаёт файлы с расширением `pyc`. В этих файлах и находится байткод, то есть скомпилированное представление вашего кода в форме, удобной Python для интерпретирования.

Допустим, нами был создан модуль `mymodule`, содержащий единственную фукнцию `multiply`:

```
# mymodule.py
def multiply(a, b):
    return a * b
```

Теперь эту функцию можно импортировать.

Функция, как и всё в Python, является объектом, и сделав её импорт, мы можем перечислить её методы с помощью функции `dir()`:

```
from mymodule import multiply

dir(multiply)
```

```
['__annotations__',
 '__call__',
 '__class__',
 '__closure__',
 '__code__',
 '__defaults__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__get__',
 '__getattribute__',
 '__globals__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__kwdefaults__',
 '__le__',
 '__lt__',
 '__module__',
 '__name__',
 '__ne__',
```

```
 '__new__',
 '__qualname__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__']
```

Один из атрибутов функции --- это `__code__`, в котором содержится `code object` --- объект, который оборачивает написанный код:

```
multiply.__code__
```

```
<code object multiply at 0x10d983ed0, file
"/Users/fz/projects/coursera/week_01/playground/mymodule.py", line 1>
```

У этого объекта есть метод *co_code*, который возвращает байткод, сгенерированный Python для этой функции (то есть инструкции для интерпретатора Python):

```
multiply.__code__.co_code
```

```
b'|\x00|\x01\x14\x00S\x00'
```

Чтобы привести байткод в читаемый вид (т.е. дизассемблировать), используется функция `dis.dis()` из пакета `dis`.

```
import dis

dis.dis(multiply)
```

```
2           0 LOAD_FAST                0 (a)
            2 LOAD_FAST                1 (b)
            4 BINARY_MULTIPLY
            6 RETURN_VALUE
```

Мы видим, что функция преобразовывается в байткод из четырех инструкций. Каждая из этих инструкций содержит определённое действие. Первая --- это `LOAD_FAST`, которая загружает содержимое переменной `a` в память и помещает его на стек. Как только переменная `a` помещена на стек, мы переходим к аналогичной инструкции для переменной `b`. Следующая инструкция --- `BINARY_MULTIPLY` --- говорит интерпретатору Python взять два самых верхних значения на стеке и произвести их умножение. Заметьте, что мы могли передать в функцию значение любого типа: целые числа, вещественные числа или строки, байткод бы от этого не поменялся. Типизация начинает работать тогда, когда Python выполняет ту или иную инструкцию. В данном случае, когда выполняется инструкция `BINARY_MULTIPLY`, Python смотрит на тип объектов, и, например, если это два числа, производит умножение двух чисел. Результат умножения он поместит на вершину стека.

И последняя инструкция --- это RETURN_VALUE, которая и возвращает переменную с вершины стека.

Любой код на Python компилируется в байткод. Все возможные инструкции, которые Python выполняет на уровне байткода, можно посмотреть следующим образом:

```
import opcode
print(opcode.opmap)
```