МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра «Автоматизация производственных процессов»

Язык программирования Python как основа обработки больших данных

Конспект лекций

Ростов-на-Дону
ДГТУ
2022

УДК 681.5

Составитель: Быкадор В.С.

Конспект лекций. – Ростов-на-Дону : Донской гос. техн. ун-т, 2022. – 60 с.

Конспект лекций по дисциплине «Язык программирования Python как основа обработки больших данных» предназначены для студентов очной и заочной форм обучения по направлению подготовки 15.04.04 «Автоматизация технологических процессов и производств» профиль «Интеллектуальные системы сбора и анализа больших данных».

УДК 681.5

Печатается по решению редакционно-издательского совета
Донского государственного технического университета

В печать ___.___.20__г.
Формат 60x84/16. Объем ____ усл. п. л.
Тираж___ экз. Заказ № ___.

Издательский центр ДГТУ
Адрес университета и полиграфического предприятия:
344000, г. Ростов-на-Дону, пл. Гагарина, 1

СОДЕРЖАНИЕ

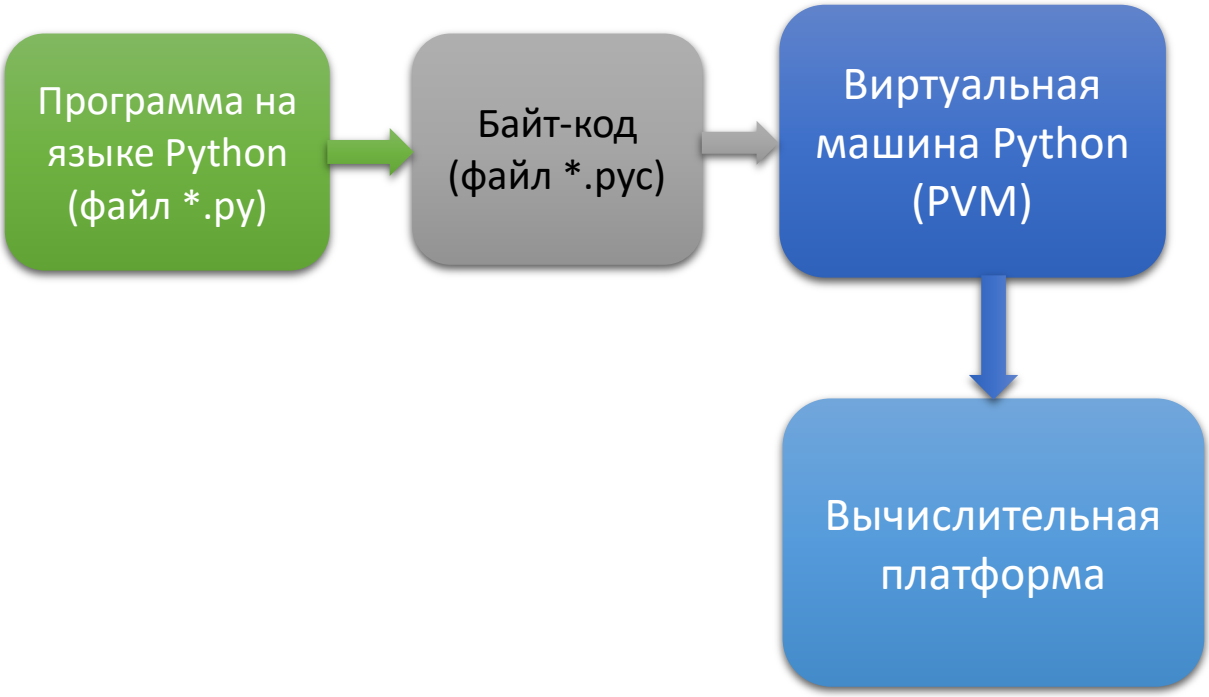
Типы и операции языка программирования Python

Общая характеристика языка программирования Python

1. Поддерживает основные парадигмы программирования.
2. Высокая степень кроссплатформенности.
3. Независимость от вендора.
4. Динамическая типизация.
5. Автоматическое управление памятью.
6. Модульная организация программ.
7. Встроенные типы объектов.
8. Широкая собственная библиотека утилит.
9. Ещё более широкая библиотека утилит и фреймворков сторонних разработчиков.

Понятие об интерпретаторе языка программирования Python

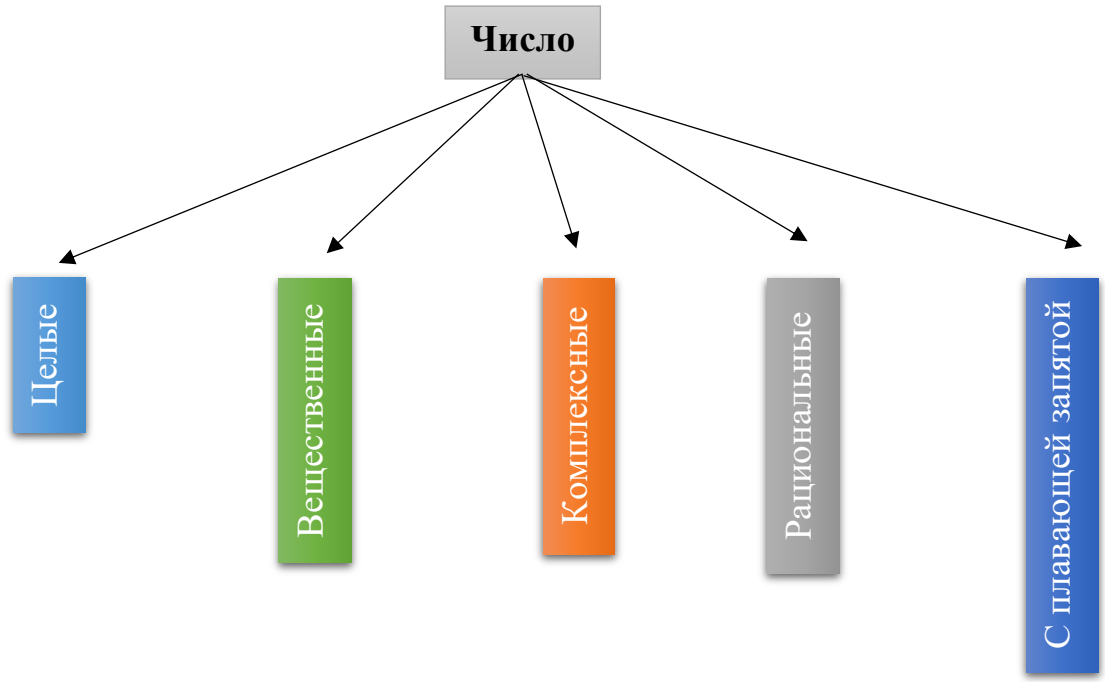
Python является интерпретируемым языком программирования.



Вначале пользователь создает программу на языке программирования Python, после запуска программы на исполнение выполняется преобразование файла программы на языке Python в так называемый файл байт-кода, который представляет собой специальный код, понимаемый только интерпретатором.

Интерпретатор – это, по сути, программа, которая преобразовывает байт-код в двоичный код процессора. Интерпретатор также часто называют виртуальной машиной, в случае языка программирования Python виртуальная машина будет называться PVM – Python Virtual Machine.

Основные типы объектов



Целые, вещественные и комплексные числа:

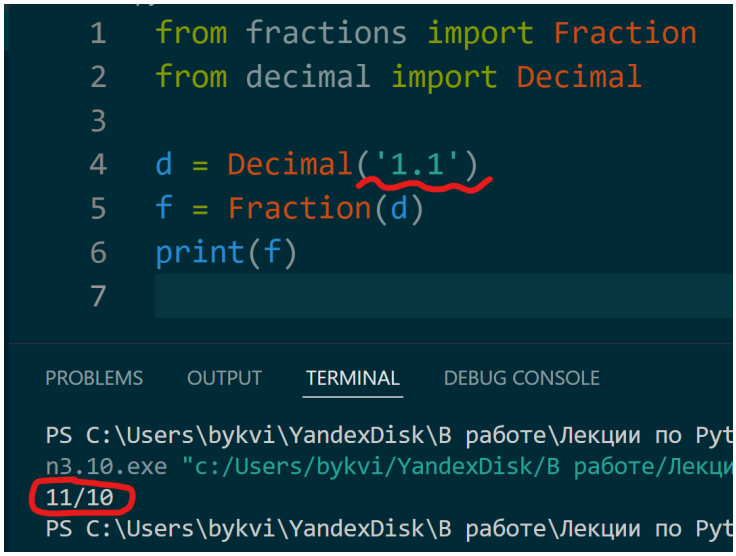
```
    Python 3.10.4
    AMD64)] on win
    Type "help", "
>>>
>>> 123
    123
>>>
>>> 3.14
    3.14
>>>
>>>
>>> 4+2j
    (4+2j)
```

Рациональные числа:

```
1  from fractions import Fraction
2
3
4  f = Fraction(2, 5)
5  print(f)
6
7
8
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Ру
n3.10.exe "c:/Users/bykvi/YandexDisk/В работе/Лекц
2/5
PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Ру
```

Работа с вещественными и рациональными числами:



Тип объекта – Строка:

1. Строки представляют собой последовательности, то есть упорядоченные коллекции других объектов – символов.
2. Индексирование символов в строке начинается с нуля.
3. Индексация обычно выполняется положительными индексами, но можно и отрицательными.
4. Так как строка - последовательность символов, то можно вычислить количество составляющих её символов, так называемую «длину» строки.
5. Можно выполнять «срезы» над строками.
6. Имеется ряд специфичных для типа методов, например, **`find`**, **`replace`**, **`split`**, **`isalpha`**, **`isdigit`**, **`upper`**, **`format`** и ряд других.
7. **Важно! Строка является неизменяемым объектов в памяти.**

Тип объекта – Списки (`list()`)

1. Списки являются обобщенным видом упорядоченных последовательностей, которые могут содержать любое количество любых других объектов при этом в пределах одного списка.
2. Так же список может содержать другие списки:

```
>>> lst = [1, 3+4j, 'spam', [1, 2, 'other']]
>>> lst
    [1, (3+4j), 'spam', [1, 2, 'other']]
>>>
```

3. Так же, как и у строк, у списков есть ряд таких же методов, например, индексация, срезы, длина списка.
4. Но для списков есть и свои специфичные методы, часто используемыми из них являются **append**, **pop**, **sort**.
5. **Важно! При присвоении одного списка другому происходит копирование ссылки на исходный список.**

Тип объекта – Словари (`dict()`)

Словари представляют собой отображения, то есть коллекции обращения к элементам которой происходит по ключу. Словари являются классом изменяемых объектов.

На рисунке ниже показаны типовые действия над словарём:

1. Объявление и инициализация переменной типа «словарь».
2. Получение значения из словаря по ключу `'lastname'`.
3. Метод `keys()` возвращает все ключи словаря.
4. Метод `values()` возвращает все значения словаря.
5. Далее выполняется добавление новой пары «ключ – значение» = `'university' – 'dstu'`, которой ранее не было в словаре. И получение значения по только что добавленному ключу `'university'`.

```
>>>
>>> d = {'name': 'Лиза', 'lastname': 'Иванова', 'age': 26}
>>> d
{'name': 'Лиза', 'lastname': 'Иванова', 'age': 26}
>>>
>>> d['lastname']
'Иванова'
>>>
>>> d.keys()
dict_keys(['name', 'lastname', 'age'])
>>> d.values()
dict_values(['Лиза', 'Иванова', 26])
>>>
>>> d['university'] = 'dstu'
>>>
>>> d
{'name': 'Лиза', 'lastname': 'Иванова', 'age': 26, 'university': 'dstu'}
>>>
>>> d['university']
'dstu'
>>>
```

Инструкции и синтаксис языка программирования Python и введение в стандарт PEP 8

Отступы являются синтаксисом языка программирования Python

Отступы являются важной особенностью языка программирования Python, так как отступы указывают на блоки кода. Другими словами, программный код, который имеет один и тоже отступ является блоком кода. В других языках блоки кода могут быть обозначены какими-нибудь символами, например, в языках C, C++, C#, Java, JavaScript для обозначения блока кода используются фигурные скобки:

```
{
        .....
        .....
        .....
        .....
}
```

В языке программирования Pascal специальные зарезервированные слова `begin` и `end`.

Основное о PEP8

PEP8 является обще принятым стандартом оформления кода, написанного на языке программирования Python. Существует большое количество рекомендаций о том, как нужно оформлять код, написанный на Python. Рассмотрим лишь основные из них:

1. Имена переменных и функций пишут строчными буквами.
2. Если имея состоит из более чем одного слова, то их разделяют подчёркиванием.
3. Имена классов пишут в стиле камелкейс (CamelCase).
4. Между импортом модулей и первой строкой кода оставляют две пустые строки.
5. Между функциями оставляют по одной пустой строке.
6. После последней строки кода в файле оставляют пустую строку.

```python
from clsDiagnosis import Diagnosis
from clsDiagnosticObject import DiagnosticObject
from clsMetricAnalysis import MetricAnalysis


number_variant = '06'

lambdas = {
    'lambda 1': 0.14
    ,'lambda 2': 0.3
    ,'lambda 3': 0.4
    ,'lambda 4': 0.16
}
```

```python
class LoadData:
    def __init__(self, num_variant, xml_file_variants, xml_file_data):
        """
        :param num_variant: номер варианта задания (две последнии цифры зачетной книжки)
        :param xml_file_variants: путь к xml-файлу со схемой задания
        :param xml_file_data: путь к xml-файлу со числовыми данными
        """
        self.__num_var = num_variant
        self.__name_element = None
        self.__root_var = et.ElementTree(file=xml_file_variants).getroot()
        self.__root_dat = et.ElementTree(file=xml_file_data).getroot()

    def __get_text_child_element(self, var):
        # возвращает текст в заданном дочернем элементе в теге 'variants'
        return var.find(self.__name_element).text.split(' ')

    def __get_names_child_elements(self, var):
        # возвращает имена дочерних элементов в теге 'variants'
        res_lst = []
        for child in var:
            m = re.findall(pattern='D[\d*]', string=child.tag, flags=re.IGNORECASE)
```

```python
    def get_data_to_X(self):
        """
        Возвращает вектора объектов распознавания Xi
        :return: список списков координат объектов распознавания
        """

        return self.get_data_to_D('X')

```

Условные оператор if

Для выбора одного из двух вариантов дальнейшего выполнения программы, то есть для реализации ветвления программного кода используется оператора выбора по условию:

```
>>> i = True
>>>
>>> if i == True:
...     print(':))')
... else:
...     print(':(')
...
...
:))
>>>
```

При использовании оператора выбора всё достаточно просто и очень похоже на то как реализуется подобный оператор в других языках программирования – если в условие выполняется, то выполняется блок кода следующий сразу за проверкой условия. Если условие не выполняется, то управление переходит к блоку кода, который следует за ключевым словом `else`.

Обратите внимание на двоеточия и обязательные отступы блоков кода!

Существует тернарная форма условной операции, которая позволяет условную операцию, у которой в блоках кода по одной команде, записывать в более компактной форме:

<истина> if <условие> else <ложь>

Запишем вышеприведённый код в тернарной форме:

```
>>>
>>> print(':))') if i == True else print(':(')
:))
>>>
```

Теперь запишем данный код в более привычной форме:

```
>>>
>>> result = ':))' if i else ':('
>>> print(result)
:))
>>>
```

В переменную `result` записывается результат операции `if` и затем полученный результат выводится при помощи функции `print()`. При этом обратите внимание на то, что нет необходимости явно прописывать условие

`if i == True`, так как по умолчанию выполняется проверка на «истинность» условия. То есть записи:

`if i == True` и `if i` являются эквивалентными.

Оператор цикла for

Оператор цикла for выполняет перебор элементов коллекции, начиная с первого элемента и до последнего элемента.

```
>>>
>>> lst = [1, 'привет', 2.55]
...
>>> for item in lst:
...         print(item)
...
...
1
привет
2.55
>>>
```

Обратите внимание на двоеточие и отступ блока кода.

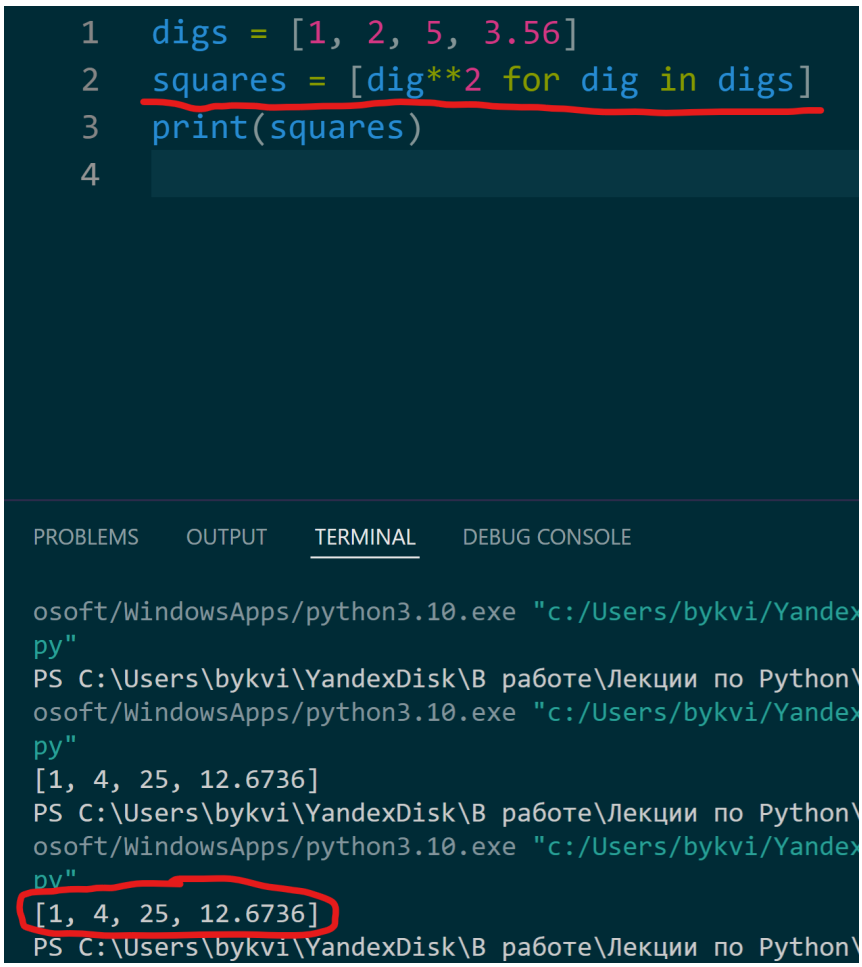
А теперь давайте возьмём список из чисел и возведем каждое из них в квадрат используя цикл `for`:

В языке Python существуют так называемые генераторы. Они используются для различных целей, но тем не менее генераторы реализуют циклы. При помощи генераторов можно предыдущий пример реализовать короче.

Форма генератора, позволяющего записать предыдущий пример в одну строчку:

```
<list> = [element for element in some_list]
```

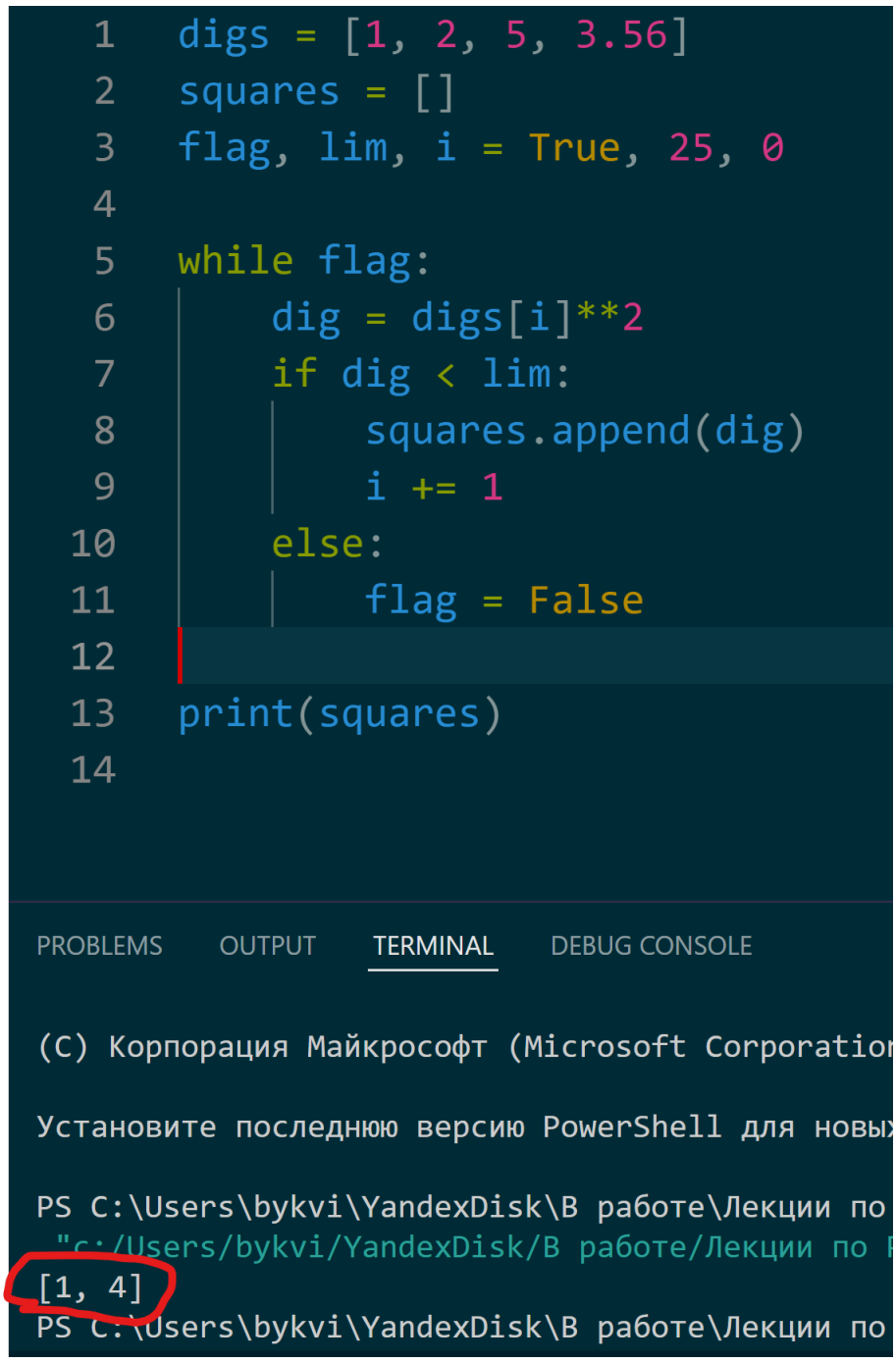
Реализация того же самого примера, но в форме генератора показана ниже:



Оператор цикла while

Если количество элементов заранее неизвестно или вообще требуется выполнять циклические действия до тех пор, пока не наступит какое-либо условие, то для этих целей подойдёт оператор цикла while.

Например, в предыдущем примере будет выполнять возведение чисел в квадрат, но до тех пор, пока мы не получим число равное или больше 25. Как мы получим число равное или больше 25, то прекращаем выполнение цикла.


1 digs = [1, 2, 5, 3.56]
2 squares = []
3 flag, lim, i = True, 25, 0
4
5 while flag:
6 dig = digs[i]**2
7 if dig < lim:
8 squares.append(dig)
9 i += 1
10 else:
11 flag = False
12
13 print(squares)
14
PROBLEMS
OUTPUT
TERMINAL
DEBUG CONSOLE
(C) Корпорация Майкрософт (Microsoft Corporatio
Установите последнюю версию PowerShell для новы
PS C:\Users\bykvi\YandexDisk\B работе\Лекции по
"c:/Users/bykvi/YandexDisk/B работе/Лекции по
[1, 4]
PS C:\Users\bykvi\YandexDisk\B работе\Лекции по

Функции языка программирования Python

Для каких целей нужны функции

1. Минимизация кода за счёт исключения его избыточности, то есть если имеется повторяющийся программный код, то можно его выделить в отдельную функцию и далее вызывать данную функцию там, где это необходимо.
2. Процедурная декомпозиция, которая позволяет разделить сложную программу на составные части, отдельно написать код для каждой части и затем с компоновать из функций целую программу.
3. Функции лежат в основе методов классов, которые являются основной частью объектно-ориентированного программирования.

Общий синтаксис функции на языке Python

```
def имя_функции([входные_параметры]):
        тело_функции
        [return результат_работы_функции]
```

def – обязательное ключевое слово, которое обозначает, что код дальше будет относится к функции.

`имя_функции` – идентификатор функции, по которому в дальнейшем будет выполняться обращение к этой функции.

`[входные_параметры]` – входные параметры позволяют передать в функцию данные в неё из вне. В общем случае входные параметры не являются обязательными и могут быть функции без входных параметров.

`тело_функции` – рабочий код функции, который выполняет определённое преобразование данных.

`[`**`return`** `результат_работы_функции]` – ключевое слово **return** используется для того, чтобы возвратить какие-либо данные из функции. `результат_работы_функции` – это какая-либо переменная, возвращаемая функцией. Так же, как и входные параметры возвращаемый результат не

является обязательным для функции, то есть функция может и не возвращать какие-либо данные.

Базовые примеры работы с функциями

Рассмотрим некоторые примеры функций:

```
def easy_func():
    pass
```

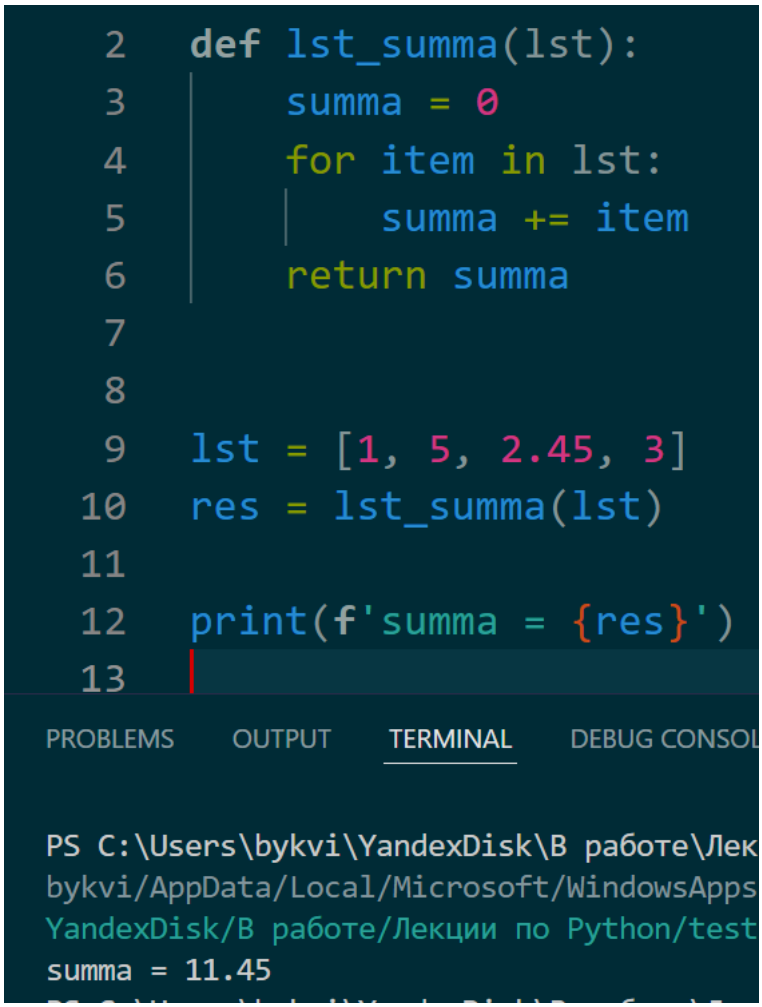
Это пример простейшая функция, которая ничего не принимает, ничего не возвращает и ничего вообще не делает, но тем не менее эта функция синтаксически правильная и если её вызвать, то программный код будет работать.

```
2  def summa(a, b):
3      c = a + b
4      return c
5
6  # Вызов функции
7  res = summa(2, 5)
8
9  print(f'2 + 5 = {res}')
10
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

bykvi/AppData/Local/Microsoft/WindowsApps/py
YandexDisk/B работе/Лекции по Python/tests/m
2 + 5 = 7
PS C:\Users\bykvi\YandexDisk\B работе\Лекции
```

Эта функция принимает два параметра – числа `a` и `b`, которые затем суммируются в функции `summa`, которая затем возвращает результат своей работы.

Так как язык программирования Python является языком с динамической типизацией, то передача в функцию параметров любых типов выполняется одинаково. Давайте передадим в функцию список чисел, выполним суммирование чисел и вернем результат.



Возвращать из функции можно тоже различные объекта языка программирования Python.

Рассмотри пример функции, которой в качестве входного параметра будет передаваться список, в котором будут содержаться целые числа и строки, функция будет выполнять разделение чисел и строк и возвращать список, который будет, содержат ещё два списка – одни с целыми числами, другой со строками.

main.py > ...

```python
def lst_filter(lst):
    lst_digs, lst_strs = [], []

    for item in lst:
        if str(item).isdigit():
            lst_strs.append(item)
        else:
            lst_digs.append(item)

    return [lst_digs, lst_strs]
```

```python
lst = [1, 5, 45, 'hello', 3, 'Мир!']
res = lst_filter(lst)

print(f'{res[0]}')
print(f'{res[1]}')
```

```
PS C:\Users\bykvi\YandexDis
bykvi/AppData/Local/Microso
YandexDisk/В работе/Лекции
['hello', 'Мир!']
[1, 5, 45, 3]
```

ООП в Python. Инкапсуляция

Общее представление об ООП

Объектно-ориентированное (ООП) программирование является самой распространённой парадигмой в создании программного обеспечения. Главной особенностью ООП является возможность создавать программу в терминах объектов реального мира. То есть программа создается как информационная модель (с той или иной степенью точности) части реального мира. Программируя в рамках ООП, можно мыслить, что работа совершается как бы над реальными объекта, это значительно упрощает процесс разработки и поддержания программного обеспечения, особенно для крупных систем.

В классическом ООП («классоориентированном») любой объект должен относиться (быть экземпляром) какого-либо класса. Класс представляет собой набор атрибутов и характеризующих какую-либо сущность и действий который эта сущность может выполнять или над которой эти действия могут выполняться.

Например, рассмотрим такой объект реального мира как работник какой-либо организации. Какие атрибуты работник может иметь? Например, такие:

1. Уникальный идентификатор работника в данной организации.
2. Имя.
3. Отчество.
4. Фамилия.
5. Дата рождения.
6. Пол.
7. Подразделение, в котором работает.
8. Стаж работы.
9. Начисленная зарплата.

А какие действия могут быть присущи работнику? Всё также зависит от задачи, для которой разрабатывается объект. Например, можно определить следующие действия:

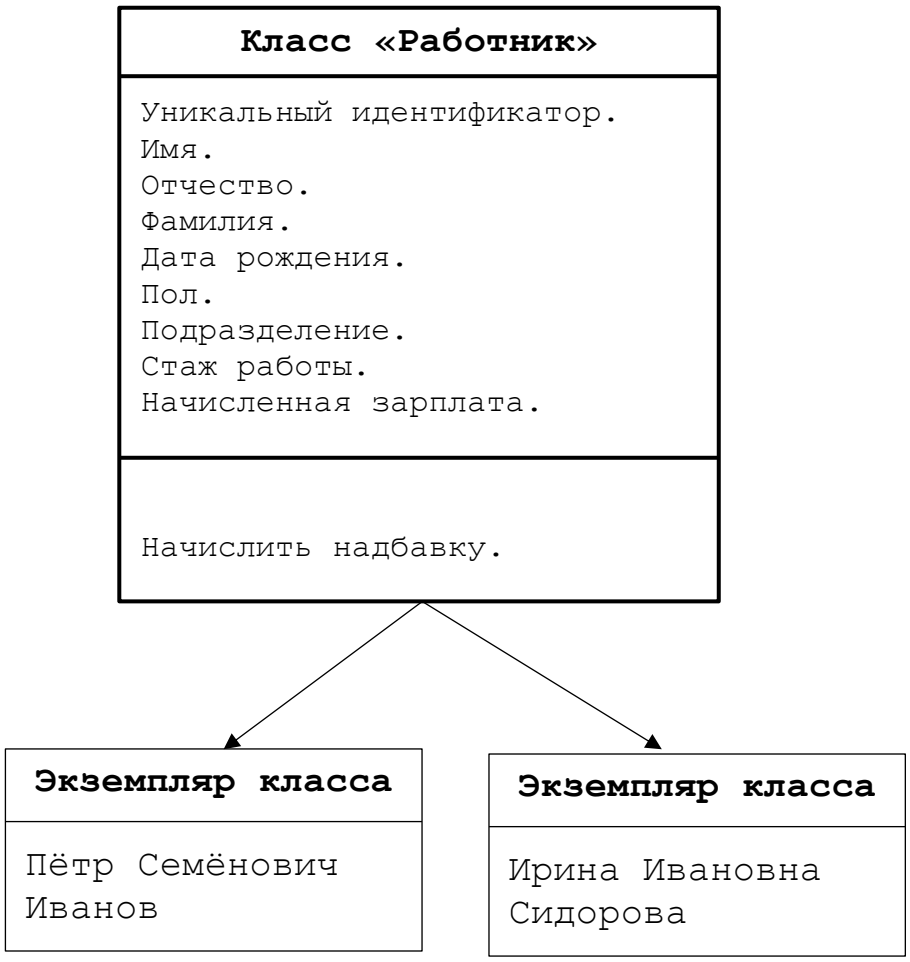
1. Принять на работу в организацию.
2. Уволить с работы.
3. Перевести в подразделение.
4. Начислить зарплату за отработанный месяц.
5. Начислить надбавку за стаж работы.

Таким образом, мы получаем некоторое обобщенное описание какого-то произвольного работника и это обобщенное описание будет являться классом.

Далее, создавая конкретных работников, как экземпляры класса, мы будем заполнять для каждого работника соответствующие атрибуты. При этом действия, которые можно выполнять над работниками, останутся одними и теми же для любого работника их не нужно будет как-то определять для каждого работника отдельно, тем самым достигает значительная «экономия кода», то есть однократно написанный код можно потом много кратно использовать. Механизмы объектно-ориентированного программирования позаботятся о правильном вызове соответствующих частей кода.

В терминах ООП атрибуты часто называют полями, а действия методами (в классической теории ООП действия называют операциями, которые при реализации в виде программы становятся методами).

Таким образом, мы можем показать следующую схему:



Можно сказать, что значения полей у каждого из экземпляров класса, то есть объектов, будут свои собственные, а вот методы все экземпляры класса будут разделять между собой.

Так же отметим, что «класс» часто называют «типом». Если тип поставляется в месте системой, в рамках её системной библиотеки, например типы работы с датами или комплексными числами, то такой тип называют встроенным. Если тип определяет программист – пользователь системы, то такой тип называют пользовательским, то есть рассматриваемый класс «Работник» - это пользовательский тип.

Определение класса в Python

Итак, теперь давайте определим наш класс «Работник» теми языковыми средствами, которые для этого предусмотрены в языке программирования Python. Для того чтобы определить класс в языке программирования Python есть ключевое слово **`class`**. Сразу отметим, что каждый класс лучше создавать в отдельном файле и затем подключать к главному файлу программы через ключевое слово `import` или использовать конструкцию `from … import`….

Получим:

```
1  class Worker:
2      pass
3
```

Пока что данный класс ничего не делает.

Конструктор класса в Python

Конструктор класса – это специальный метод, который вызывается при создании экземпляра класса. Как правило, почти у каждого класса есть конструктор. Семантически конструктор представляется как функция и вообще методы в классах записываются как функции. У конструктора есть определенное название, которое зарезервировано и интерпретатор языка программирования Python понимает, что он имеет дело именно с конструктором класса.

Сигнатура конструктора класса в языке программирования Python выглядит следующим образом:

```
def __init__(self):
```

Это так называемый дандер-метод (методы, которые в своем названии имеют два подчеркивания до и после смыслового названия, в Python

называются дандер-методы или «магические» методы). Таких методов в Python несколько, у всех у них зарезервированные имена и каждый из них нужен для определенных целей, но мы сейчас рассматривает именно конструктор.

Как правило, объявление и инициализацию полей выполняют в конструкторе. Давайте подробнее разберем это утверждение.

«Как правило» — это означает, что в языке программирования Python обычно выполняют инициализацию полей в конструкторе. Например, в других языках объявление выполняют вне конструктора, а инициализацию в конструкторе, но в Python эти два действия совмещены в одном месте и это место как правило конструктор. Так же объявление и инициализацию можно выполнять по месту необходимости, например, в тех методах где соответствующие поля требуются, но тогда в случае внесения правок потребуется затратить время на поиск этих полей по различным методам, а если другой программист не знает, что в классе уже есть соответствующее поле, то он может определить своё поле и тем самым «затереть» уже существовавшее ранее определение поля. Поэтому лучше всё же объявлять и инициализировать поле в конструкторе.

«объявление и инициализация» — это утверждение говорить о том, что существует объявление переменной, в данном случае поля, и его инициализация, то есть присвоение ему значения. Такая ситуация характерна для языков со статической типизацией, особенно это важно для первых языков со статической типизацией, например, языка программирования Си, например:

```
// объявление переменной
int number;

……
……
……

// инициализация переменной, то есть присвоения ей значения
number = 3;
```

Но разделение объявления и инициализации не рекомендуется делать, так как это несёт ряд проблем, особенно в программах на языке программирования Си, а лучше сразу выполнять объявление и инициализацию, то есть:

```
// объявление и инициализация переменной
int number = 3;
```

Не будем сейчас вдаваться в подробности почему лучше сразу объявлять и инициализировать переменную и почему именно для языка Си это имеет такое важное значение, для этого у нас будет отдельный курс. Что касается языка программирования Python, то данный язык имеет динамическую типизацию и определенны синтаксис, который просто не позволяет программисту отдельно выполнять объявление и отдельно выполнять инициализацию переменной – когда Вы пишете код на языке программирования Python, то вы сразу объявляете и инициализируете переменную, например:

```
number = 3
```

Попробуйте отдельно объявить и отдельно инициализировать переменную.

Пример конструктора класса в Python

Итак, давайте добавим конструктор в наш класс и выполним в нем объявление и инициализацию полей.

```
1  import datetime
2
3  class Worker:
4      def __init__(self):
5          self.id = 0
6          self.name = ''
7          self.middle_name = ''
8          self.last_name = ''
9          self.birth_date = datetime.datetime(1900)
10         self.gender = None
11         self.department = None
12         self.seniority = None
13         self.salary = 0
14
```

Может возникнуть вопрос:

- Что это за такое `self` и в качестве параметра конструктора, и перед каждым названием поля?

Формально, `self` — это экземпляр класса. Как уже отмечалось ранее, класс представляет собой общее описание объекта, в данном случае это какой-то рабочий. Но конкретные рабочие все разные, у них свои имена, фамилии, года рождения, заплаты и так далее. А класс один, общий… Так вот, чтобы у каждого конкретного рабочего была нужная структура полей, а значения этих полей были свои собственные нужно передать соответствующую «метку» вот этот самый `self`. Технически это означает, что в ОЗУ ЭВМ выделяется область памяти, для каждого конкретного «рабочего», в которой создаётся структура полей типа, а так как область памяти отдельная, то данные в эту структуру полей для каждого «рабочего» можно записывать свои.

С точки зрения программиста требуется выполнить только запись ключевого слова `self` непосредственно при определении полей и метода в самом классе. При создании экземпляров класса ничего дополнительно делать не нужно – PVM всё сделает сама.

Может возникнут вопрос:

- А зачем тогда вообще прописывать `self` в классе? Пусть PVM тогда сама его «пропишет» как-то.

Дело в том, что могут быть данные, разделяемые между различными экземплярами класса, то есть такие поля, доступ к которым могут иметь все «рабочие», грубо говоря. Такие поля не создаются в области ОЗУ для каждого экземпляра класса, а находятся в общей (для всех экземпляров класса) памяти. В это случае нет необходимости прописывать ключевое слово `self`.

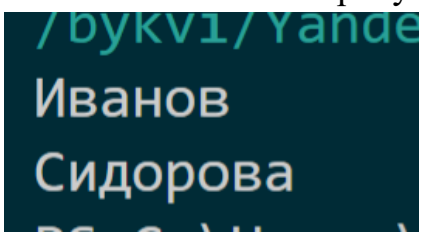
Другими словами, **когда программист в классе использует ключевое слово `self`, то он явно указывает, что данные, которые будут храниться в данном поле у каждого экземпляра объекта будут свои собственные.**

Создание и работа с экземпляром класса в Python

Несмотря на то, что наш класс `Worker` ещё далёк от завершенного вида всё равно давайте уже создадим экземпляры этого класса и попробуем с ними поработать. Ниже представлен листинг программы, в которой создаётся два экземпляра класса `Worker` – `worker01` и `worker02`. Далее поля этих экземпляров класса заполняются. После чего мы можем обратиться к любому полю любого из двух экземпляров и получить ранее записанные туда данные.

```
import datetime
from clsWorker import Worker


worker01 = Worker()

worker01.id = 123343
worker01.name = 'Пётр'
worker01.middle_name = 'Семёнович'
worker01.last_name = 'Иванов'
worker01.birth_date = datetime.datetime(1985, 3, 10)
worker01.gender = 'male'
worker01.department = 24
worker01.seniority = 'senior'
worker01.salary = 150000


worker02 = Worker()

worker02.id = 101000
worker02.name = 'Ира'
worker02.middle_name = 'Ивановна'
worker02.last_name = 'Сидорова'
worker02.birth_date = datetime.datetime(1998, 10, 15)
worker02.gender = 'female'
worker02.department = 24
worker02.seniority = 'junior'
worker02.salary = 50000

print(worker01.last_name)
print(worker02.last_name)
```

Результат работы программы показан ниже на рисунке.



Инкапсуляция данных в классе Python

Инкапсуляция является важнейшим принципом ООП и предполагает, что данные экземпляра класса скрыты от внешней части программы. Это важно, так как экземпляр класса сам владеет и сам модифицирует свои данные, тем самым обеспечивая из целостность и скрывает внутреннею организацию данных от внешней части программы.

Для реализации принципа инкапсуляции данных поля класса объявляются частными, то есть доступ к эти полям будет иметь только данный экземпляр класса внутри себя. В разных языках программирования по-разному синтаксически объявляются частные поля, в Python необходимо перед имением поля поставить два подчёркивания и тогда это поле будет частным, например:

clsWorker.py > Worker > __init__

```python
import datetime

class Worker:
    def __init__(self):
        self.__id = 0
        self.__name = ''
        self.__middle_name = ''
        self.__last_name = ''
        self.__birth_date = datetime.datetime(1900, 1, 1)
        self.__gender = None
        self.__department = None
        self.__seniority = None
        self.__salary = 0
```

Теперь если мы обратимся к любому полю экземпляра класса, то получим ошибку, так из внешней части программы не будет ничего «видно» что там «внутри» экземпляра класса.

Может возникнуть вопрос: «А как тогда присваивать значения полям класса?». Присваивать значения полям можно через методы, которые специально работают с полями – эти методы часто называют **методами доступа**.

Но также можно использовать и конструктор класса, который позволяет сразу задать значения частных полей. Задание значений полей через конструктор используют специально тогда, когда разработчик класса желает гарантировать то, что нужные поля будут точно заданы пользователем класса, так как в противном случае экземпляр класса не создаться и программа не выполнится. Можно задать все поля через конструктор, можно не все – это всё определяется конкретным классом.

В нашем учебном классе давайте через конструктор зададим все те поля которые однозначно относятся к работнику:

```
self.__id
self.__name
self.__ middle_name
self.__last_name
self.__birth_date
self.__gender
```

Для этого в конструктор введем параметры, как в функции.

```
2  class Worker:
3      def __init__(self, id, name, middle_name, last_name, birth_date, gender):
4          self.__id = id
5          self.__name = name
6          self.__middle_name = middle_name
7          self.__last_name = last_name
8          self.__birth_date = birth_date
9          self.__gender = gender
10
11         self.__department = None
12         self.__seniority = None
13         self.__salary = 0
14
```

Инициализация экземпляров класса можно выполнить следующими способами:

```
1  import datetime
2  from clsWorker import Worker
3
4  birth_date01 = datetime.datetime(1985, 3, 10)
5  worker01 = Worker(123343, 'Пётр', 'Семёнович', 'Иванов', birth_date01, 'male')
6
7  worker02 = Worker(
8      id=101000,
9      name='Ира',
10     last_name='Сидорова',
11     middle_name = 'Ивановна',
12     gender = 'female',
13     birth_date = datetime.datetime(1998, 10, 15)
14     )
15
```

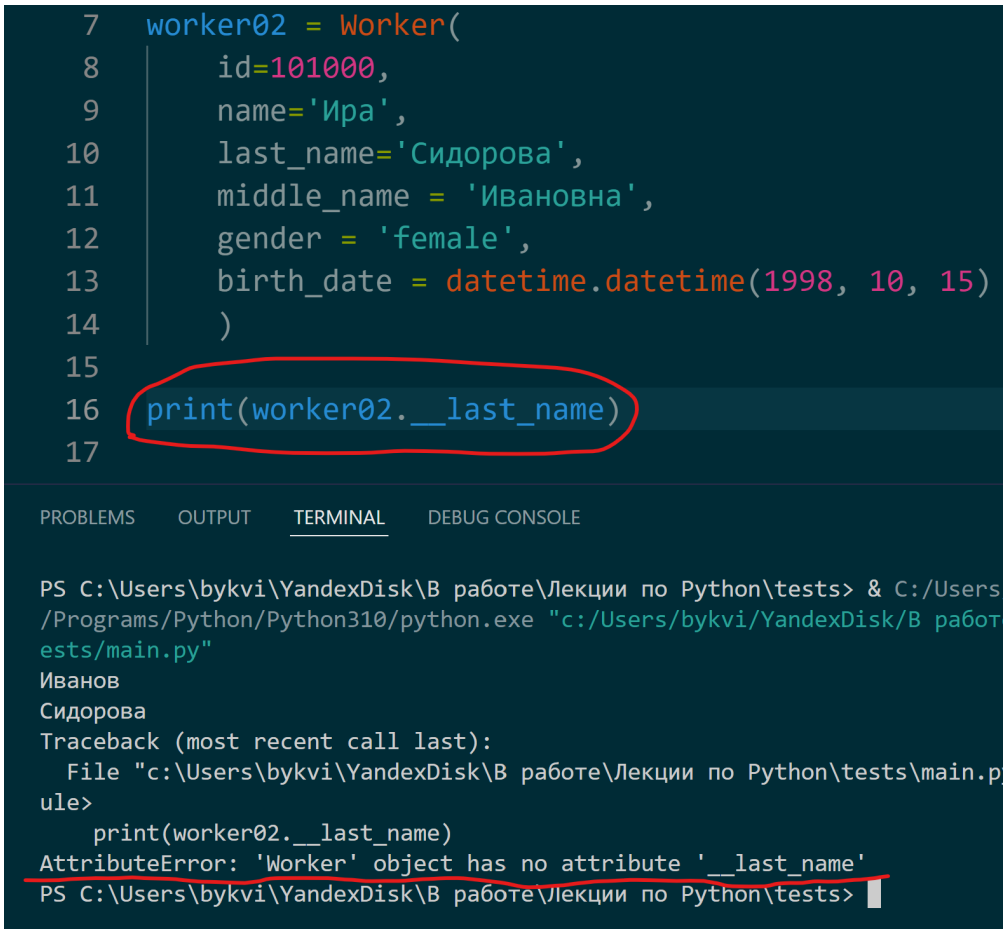
То есть используя позиционный и именованный способ задания параметров. Давайте выведем фамилии сотрудников в консоль, добавив временно соответствующую функцию в конструктор класса.

```
3  class Worker:
4      def __init__(self, id, name, middle_name,
5          self.__id = id
6          self.__name = name
7          self.__middle_name = middle_name
8          self.__last_name = last_name
9          self.__birth_date = birth_date
10         self.__gender = gender
11
12         self.__department = None
13         self.__seniority = None
14         self.__salary = 0
15
16         print(self.__last_name)
```

Тогда в консоли получим:

```
/Programs/Python/Python310/py
ests/main.py"
Иванов
Сидорова
PS C:\Users\bykvi\YandexDisk
```

Хорошо, значения полям присвоены через инициализацию в конструкторе. Но если мы теперь попробуем получить доступ к частному полю непосредственно - вдруг возникла такая необходимость, что произойдёт?



PVM сгенерирует ошибку, так как нельзя общаться к частным (закрытым) атрибутам к которым относятся поля.

А если нужно обратиться к частному полю класса то, как быть? В этом случае нужен метод доступа, который будет публичным, то есть к нему можно будет выполнить обращение из вне класса, но сам метод доступа будет работать с частным полем, то есть будет являться некоторым посредником между внешней частью программы и частными (закрытыми) полями экземпляра класса. Например, для поля `__last_name` мы можем определить следующий метод доступа для получения его значения – метод `get_last_name ()`.

```python
class Worker:
    def __init__(self, id, name, middle_name,
        self.__id = id
        self.__name = name
        self.__middle_name = middle_name
        self.__last_name = last_name
        self.__birth_date = birth_date
        self.__gender = gender

        self.__department = None
        self.__seniority = None
        self.__salary = 0

    def get_last_name(self):
        return self.__last_name

```

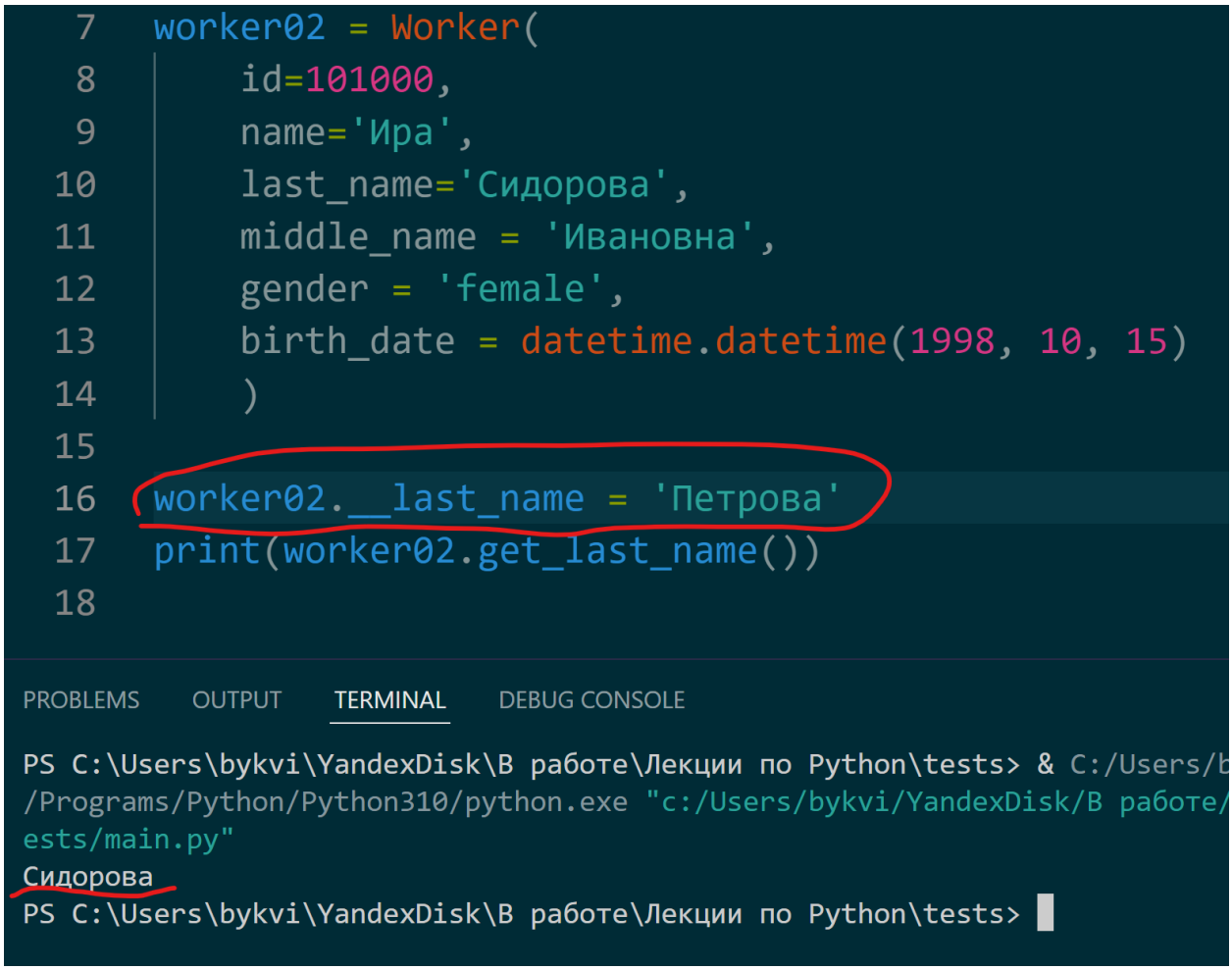
Теперь ещё раз попробуем получить «фамилию», но уже через метод доступа.

```python
worker02 = Worker(
    id=101000,
    name='Ира',
    last_name='Сидорова',
    middle_name = 'Ивановна',
    gender = 'female',
    birth_date = datetime.datetime(1998, 10, 15)
    )

print(worker02.get_last_name())
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests> & C:/Users/b
/Programs/Python/Python310/python.exe "c:/Users/bykvi/YandexDisk/В работе/
ests/main.py"
Сидорова
PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests>
```

Всё прошло успешно.

А если, например, нужно в ходе работы программы поменять значение в каком-либо частном поле, ведь к нему нельзя обратиться напрямую.



Поменять фамилию не получилось. Для этого нужен другой метод доступа к полю `__last_name`, который позволит присвоить новое значение частному полю. Назовём этот метод `set_last_name()`.

```
3   class Worker:
4       def __init__(self, id, name, middle_name,
5           self.__id = id
6           self.__name = name
7           self.__middle_name = middle_name
8           self.__last_name = last_name
9           self.__birth_date = birth_date
10          self.__gender = gender
11
12          self.__department = None
13          self.__seniority = None
14          self.__salary = 0
15
16      def get_last_name(self):
17          return self.__last_name
18
19      def set_last_name(self, value):
20          self.__last_name = value
```

Теперь ещё раз попробуем поменять «фамилию», но уже через метод доступа.

```
7   worker02 = Worker(
8       id=101000,
9       name='Ира',
10      last_name='Сидорова',
11      middle_name = 'Ивановна',
12      gender = 'female',
13      birth_date = datetime.datetime(1998, 10, 15)
14      )
15
16  worker02.set_last_name('Петрова')
17  print(worker02.get_last_name())
18
```

```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests> & C:/Users/
/Programs/Python/Python310/python.exe "c:/Users/bykvi/YandexDisk/В работе
ests/main.py"
Петрова
PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests>
```

Всё прошло успешно.

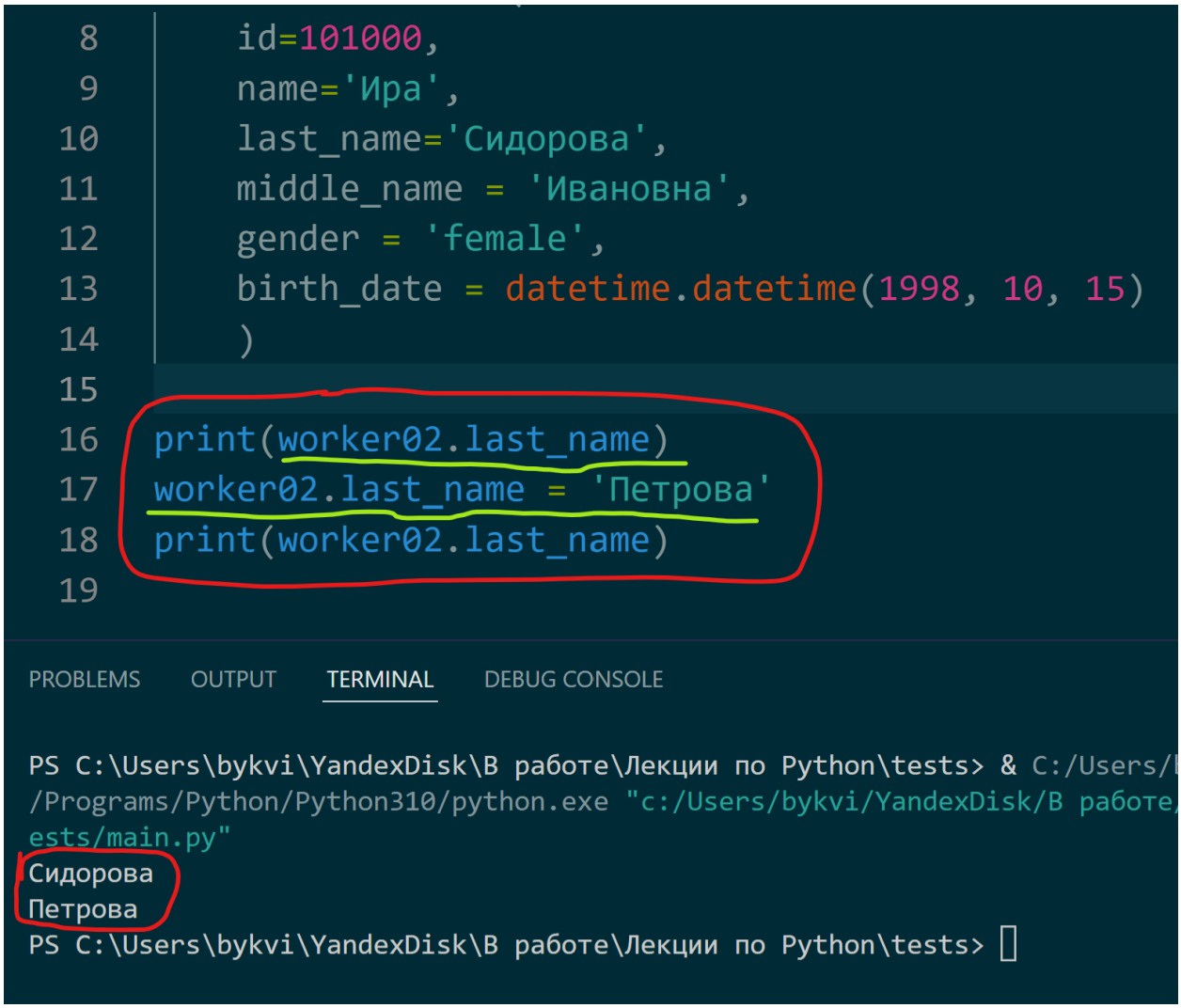
Методы доступа, которые позволяют получить данные из частного поля класса называются «геттерами», а методы доступа которые позволяют записать данные в частное поле класса называются «сеттерами».

Работа с частными поля классов через методы доступа является классическим подходом, но язык Python, как и многие другие современные языки программирования поддерживающие объектно-ориентированное программирование предоставляет так называемые свойства.
Свойства – это те же по сути «гетторы» и «сетторы», но синтаксически оформленные немного по-другому, что позволяет пользователю класса работать немного удобнее с классом. Давайте перепишем предыдущий пример на свойства.

```
3   class Worker:
4       def __init__(self, id, name, middle_name,
5           self.__id = id
6           self.__name = name
7           self.__middle_name = middle_name
8           self.__last_name = last_name
9           self.__birth_date = birth_date
10          self.__gender = gender
11
12          self.__department = None
13          self.__seniority = None
14          self.__salary = 0
15
16      @property
17      def last_name(self):
18          return self.__last_name
19
20      @last_name.setter
21      def last_name(self, value):
22          self.__last_name = value
23
```

Обратите внимание на синтаксис!

Теперь работать с частными полями экземпляра класса можно более естественно.



Обычно, когда говорят про поля, то подразумевают что это частные (закрытые) атрибуты класса, когда говорят про свойства или методы доступа, то подразумевают публичные (открытые) атрибуты класса.

Может возникнуть вопрос, связанный с тем, зачем вводить абстракцию частных полей и публичных свойств, почему сразу не сделать поля публичными как это было сделано в самом начале рассмотрения примера с классом?

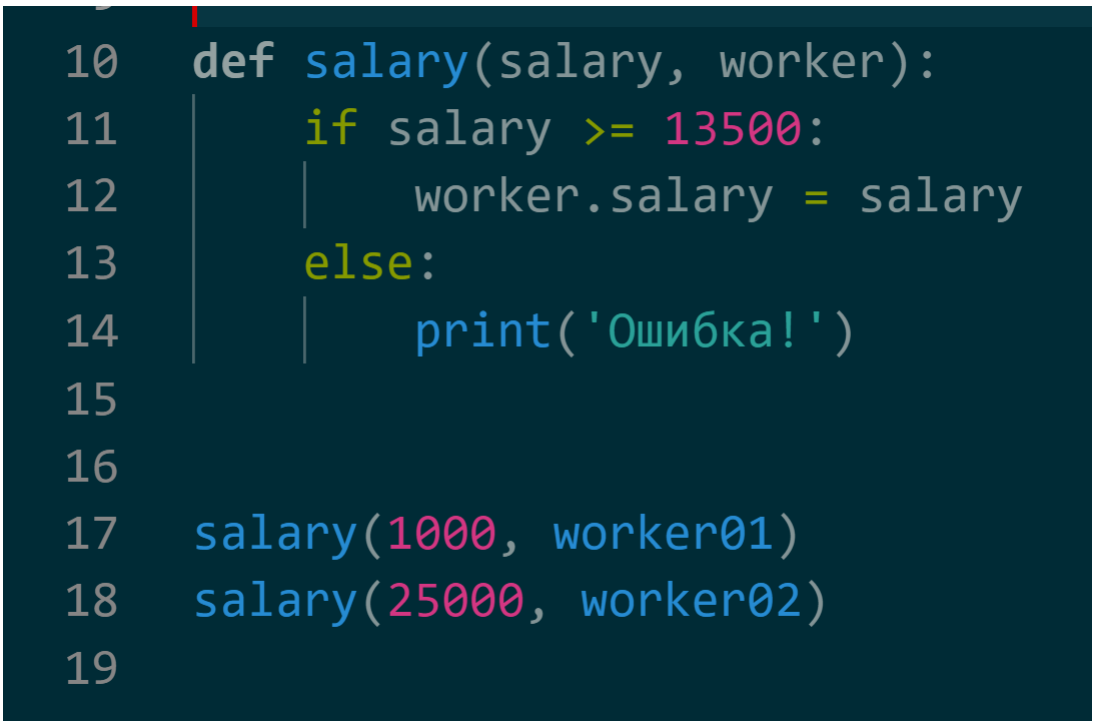
Частные поля могут требоваться для работы самого класса, то есть такие поля как переменные используются внутри класса и не предназначены для публикации, более того доступ к таким полям из вне может помешать нормальной работе класса. Например, при создании экземпляра класса, класса обращается к базе данных и получает значение минимально возможной оплаты труда, которое потом используется в его работе. И нельзя позволить из вне поменять это значение иначе расчёты будут неправильные.

Другой причиной разделение на поля и свойства является то, что у полей нет информации о том, как работать с теми данными, которые им присваиваются.

Например, у класса `Worker` есть поле `__salary`. Допустим это поле будет публичным, то есть `salary`. И мы присваиваем ему значение, но это значении не должно быть меньше минимальной оплаты труда. Но как этого добиться? Можно, конечно, выполнить предварительную проверку, например так:

```
10  new_salary = 1000
11  if new_salary >= 13500:
12      worker01.salary = new_salary
13  else:
14      print('Ошибка!')
15
16  new_salary = 25000
17  if new_salary >= 13500:
18      worker02.salary = new_salary
19  else:
20      print('Ошибка!')
```

Но это не эффективно, особенно если есть большое количество работников. Можно, конечно, выделить специальную функцию, например так:



```
10  def salary(salary, worker):
11      if salary >= 13500:
12          worker.salary = salary
13      else:
14          print('Ошибка!')
15
16
17  salary(1000, worker01)
18  salary(25000, worker02)
19
```

Так уже лучше, но есть ряд проблем:

1) Программист, который использует класс `Worker` должен в принципе знать, что есть некоторая минимальная граница оплаты труда. А если таких полей 10 и ещё около 20 разных классов…, то получается, что тот, кто использует класс должен об этом всём знать?
2) Если один и тот же класс используется в разных проектах, то такую функцию проверки нужно тем или иным образом переносить из проекта в проект. При большом количестве таких функций можно какую-нибудь забыть перенести. А если новый проект начинает делать другой программист, который вообще может не знать про существование таких функций проверки?

Эти проблемы возникают, потому что публичное поле ничего не знает о тех данных, которые ему присваиваются и нет возможности как-то в этом поле написать правила обработки данных этого же поля. Такие правила часто называют «бизнес-логика» или «бизнес-правила».

И вот здесь как раз может помочь концепция разделения на частные поля и публичные свойства. Поля будет данные хранить, а свойства реализовывать всякие проверки (бизнес-логику) перед тем как эти данные будет записаны в поле из вне или считаны из поля для внешнего потребителя.
Поэтому давайте обратно сделаем рассматриваемое поле частным, то есть `__salary` частным добавим соответствующие свойства или методы

доступа, в которых пропишем нужную «бизнес-логику» работы с этим полем.

```
12          self.__department = None
13          self.__seniority = None
14          self.__salary = 13500
15
16      @property
17      def salary(self):
18          return self.__salary
19
20      @salary.setter
21      def salary(self, value):
22          if value >= 13500:
23              self.__salary = value
24          else:
25              print('Ошибка!')
```

Теперь внешний пользователь класса может обращаться к данным класса, а класс реализует необходимую «бизнес-логику» работы с данными.

```
12
13  worker01.salary = 1000
14  worker02.salary = 25000
15  print(worker02.salary)
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests> & C:/
/Programs/Python/Python310/python.exe "c:/Users/bykvi/YandexDisk/B
ests/main.py"
Ошибка!
25000
PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests>
```

Таким образом, программисту, который пользуется классом, нет необходимости заботится о различных особенностях, которые должен учитывать класс. Об этом заботится тот, кто класс разрабатывает, так как он лучше понимает специфику класса, по крайней мере так должно быть. И с переносом кода по разным проектам тоже нет проблем, так как «бизнес-логика» переносится вместе с классом автоматически.

Теперь давайте реализуем единственный метод «Начислить надбавку». Методы в ООП семантически представляют собой функции с обязательными параметром self, который указывает на экземпляр класса. В отличии от свойств методу можно передавать параметры. Методы могут быть как публичные, который можно вызывать из вне экземпляра класса, так и частными, которые вызываются только внутри экземпляра класса и предназначены для «внутреннего» использования.

Итак, давайте реализуем «начисление надбавки» в нашем классе:

```
19      @property
20      def salary(self):
21          return self.__salary
22
23      @salary.setter
24      def salary(self, value):
25          if value >= 13500:
26              self.__salary = value
27          else:
28              print('Ошибка!')
29
30      def add_to_salary(self, procent):
31          self.salary += (procent / 100) * self.salary
32
```

Теперь воспользуемся только что реализованным методом в наших экземплярах класса:

```
13  worker01.salary = 15000
14  worker02.salary = 25000
15
16  print('')
17  print('Зарплата:')
18  print(worker01.salary)
19  print(worker02.salary)
20
21  worker01.add_to_salary(10)
22  worker02.add_to_salary(12)
23
24  print('')
25  print('--------------------------------')
26  print('Зарплата после надбавки:')
27  print(worker01.salary)
28  print(worker02.salary)
```

Результат выполнения выше приведенного кода:

```
Зарплата:
15000
25000

--------------------------------
Зарплата после надбавки:
16500.0
28000.0
```

ООП в Python. Наследование

Общее представление об наследовании в ООП

Наследование в ООП представляет собой механизм расширения возможностей класса. При использовании механизма наследования исходный класс, то есть класс от которого будет производится наследование называется как правило «родительским», а тот класс, которой будет создан в результате наследования называется «дочерним».

Механизм наследования позволяет расширить или изменить функциональность родительского класса придерживаясь двух важных принципов:

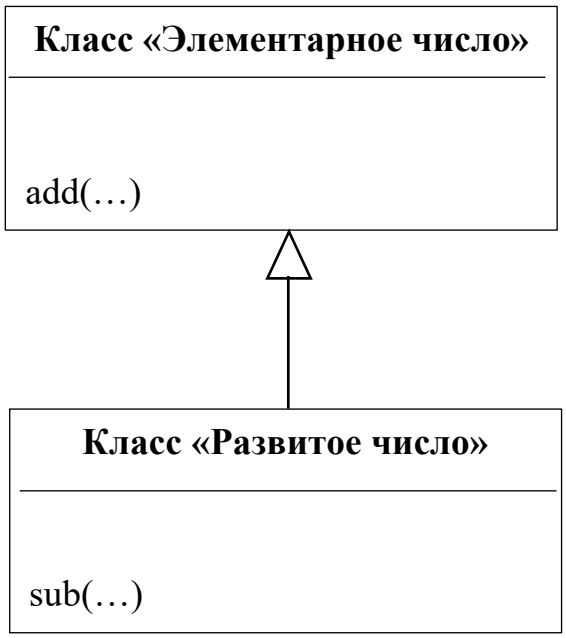
1. Не вносить изменения в исходный – родительский класс.
2. В дочернем классе должно быть минимум кода, это код должен касаться только новой функциональности вся та функциональность, которая должна повторять то, что есть в родительском классе автоматически будет взята из родительского класса.

Итак, рассмотрим синтетический пример, чтобы проиллюстрировать наследование в действии.

Пусть имеется класс «Элементарное число», которое только может суммироваться с таким же «Элементарными числом» и больше ничего не может делать.

Затем нам вдруг потребовалось число более развитое, которое может не только суммироваться с себе подобными, но может ещё и вычитаться, назовем такой класс как «Развитое число».

Для решения данной задачи мы можем поступить двумя способами: просто скопировать существующий код и класса «Элементарное число» в класс «Развитое число» и добавить в класс «Развитое число» код с недостающим методом, но это не путь ООП. А путь ООП это взять и унаследовать класс «Развитое число» от класса «Элементарное число», тогда классу «Развитое число» будет доступны методы класса «Элементарное число» автоматически, дописать нужно будет только метод для вычитания.
На схеме можно это показать так:



Давайте рассмотрим программный код для класса «Элементарное число» - ElementaryDigital.

```
1
2   class ElementaryDigital:
3       def __init__(self, value):
4           self._value = value
5
6       @property
7       def value(self):
8           return self._value
9
10      @value.setter
11      def value(self, val):
12          self._value = val
13
14      def add(self, elementary_digit):
15          return ElementaryDigital(self._value + elementary_digit.value)
16
```

Обратите внимание на то, что поле `_value` записывается с одним символом подчёркивания, а не с двумя как ранее. Что это означает? Это означает, что поле является защищённым и такое определение поля является не что средним между частным и публичным. К защищённому полю нельзя обратиться из вне класса, но к этому полю могут обращаться потомки этого класса. Тогда можно свести наши знания о модификаторах доступа атрибутов в следующую таблицу:

Модификатор атрибута	Обозначение в Python	Значение атрибута	Пример атрибута

Частный (private)	Два подчёркивания	Обращаться можно только внутри того класса, в котором данный атрибут был определён.	`self.__value = 5`
			`def __add(self):`
			`  pass`

Защищённый (protected)	Одно подчёркивание	Обращаться можно внутри того класса, в котором данный атрибут был определён, а также к атрибуту могут обращаться наследники данного класса.	`self._value = 5`
			`def _add(self):`
			`  pass`

Публичный, общедоступный (public)	Нет каких-либо специальных обозначений	Обращаться можно в любой части кода:
		в классе, где атрибут определён, в потомках класса, из внешнего кода, где есть ссылка на класс.
			`self.value = 5`
			(для соблюдения принципа инкапсуляции лучше публичными делать не поля, а свойства)
			`def add(self):`
			`  pass`

Вернёмся к нашему классу.

Ниже приведён результат выполнения кода с использованием экземпляров класса ElementaryDigital:

```python
from clsElementaryDigital import ElementaryDigital

a = ElementaryDigital(5)
b = ElementaryDigital(3)

c = a.add(b)

print(f'a = {a.value}')
print(f'b = {b.value}')
print(f'c = {c.value}')

```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests> & C:/User
s/bykvi/AppData/Local/Microsoft/WindowsApps/python3.10.exe "c:/Users/by
kvi/YandexDisk/В работе/Лекции по Python/tests/main.py"
a = 5
b = 3
c = 8
```

Теперь добавим класс «Развитое число» - AdvancedDigital, которое расширяет свой родительский класс «Элементарное число» - ElementaryDigital.

```python
from clsElementaryDigital import ElementaryDigital

class AdvancedDigital(ElementaryDigital):
    def sub(self, advanced_digit):
        return AdvancedDigital(self._value - advanced_digit.value)

```


Класс «Развитое число» - AdvancedDigital был унаследован от класса «Элементарное число» - ElementaryDigital, поэтому класса «Развитое число» получил всё что есть у класса «Элементарное число» (кроме частных атрибутов, если такие имеются). Далее, был добавлен новый публичный метод

`sub(...)` в класс наследник «Развитое число», тем самым класс наследник расширил класс родителя.

Посмотрим в работе на класс «Развитое число»:

```
 1  from clsAdvancedDigital import AdvancedDigital
 2
 3  a = AdvancedDigital(5)
 4  b = AdvancedDigital(3)
 5
 6  d = a.sub(b)
 7  c = a.add(b)
 8
 9  print(f'a = {a.value}')
10  print(f'b = {b.value}')
11  print(f'd = {d.value} sub(...)')
12  print(f'c = {c.value} add(...)')
13
```

```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests>
osoft/WindowsApps/python3.10.exe "c:/Users/bykvi/YandexDisk/B
py"
a = 5
b = 3
d = 2 sub(...)
c = 8 add(...)
```

Как можно видеть, экземпляру класса «Развитое число» доступен как его собственный метод sub(…), так и метод add(…) родительского класса «Элементарное число».

ООП в Python. Полиморфизм

Общее представление о полиморфизме в ООП

Полиморфизм на греческом языке означает «поли» - много, а «морфизм» - форма, то есть полиморфизм дословно можно перевести как «многоформ».

В языках программирования с динамической типизацией, например, в языке программирования Python, полиморфизм часто встречается так как «утиная» типизация этому широко способствует.

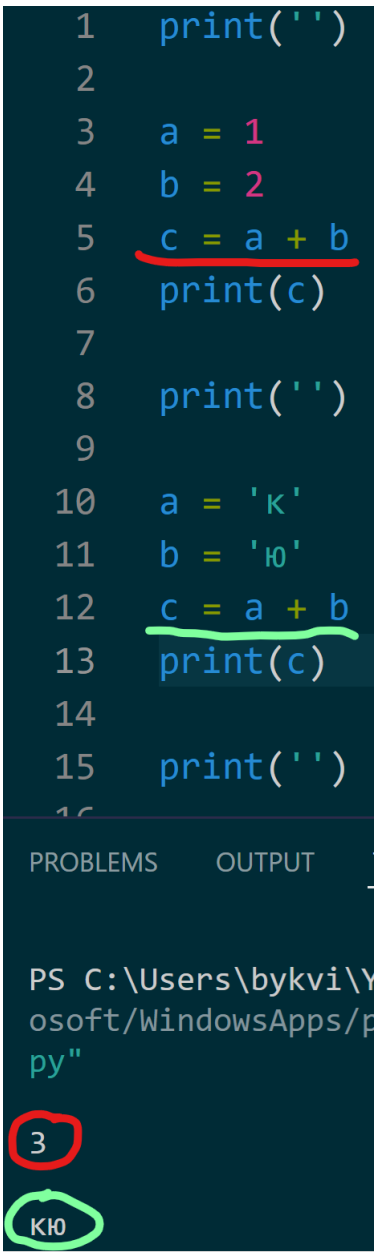
В классическом смысле полиморфизм делят на параметрический (или истинный) ad-hoc-полиморфизм (специальный или специализированный). Часто полиморфизм дают в следующем определении: один интерфейс — множество программных реализаций. Под это определение подпадает ad-hoc-полиморфизм.

Ad-hoc-полиморфизм во многих языках поддерживается по средствам перегрузки функций и методов, а в слабо типизированных языках (например, в языках с динамической типизаций поддерживающих принцип «утиной» типизации - как язык программирования Python) – по средствам (неявного) приведения типов.

Полиморфизм основанный на неявном анализе типов в языке программирования Python

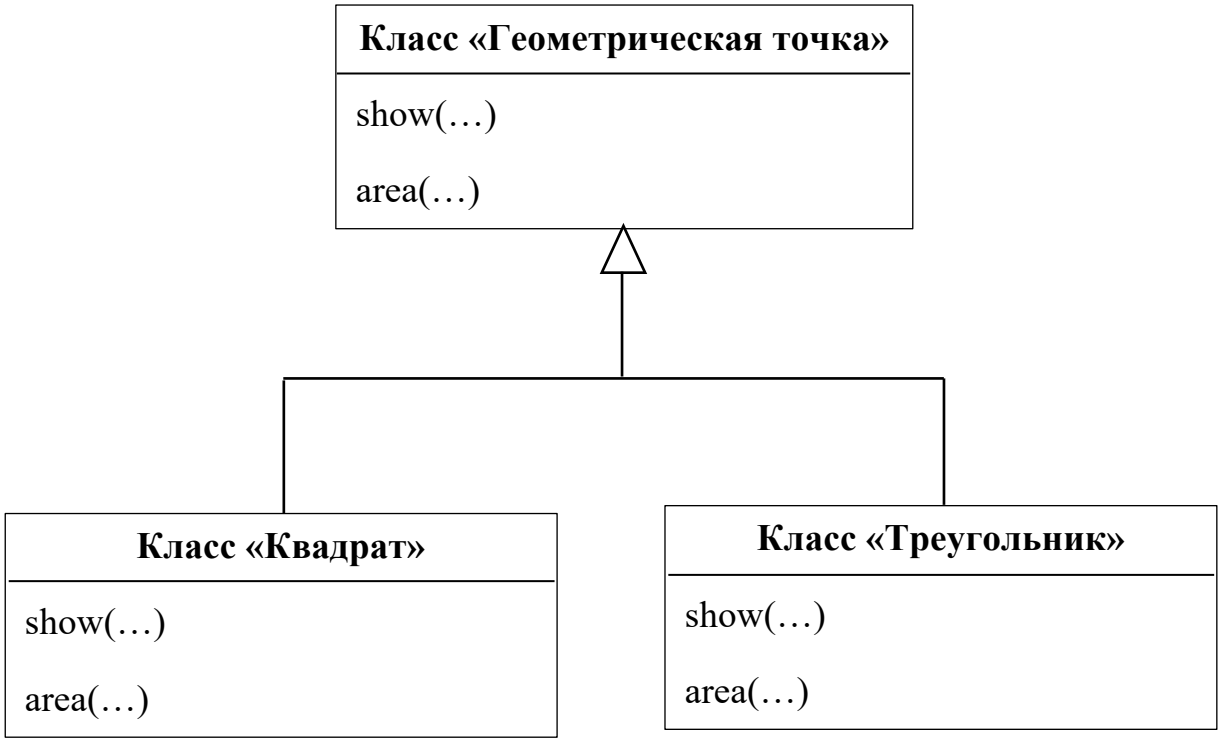
Рассмотрим следующий программный код на языке программирования Python.

В начале переменным `a` и `b` присваиваются целые числа и выполняется операция из суммирования этих чисел. Далее переменным `a` и `b` присваиваются символы и выполняется операция конкатенации – соединения символов. То есть, хотя операция между двумя переменными обозначается одинаково – знаком «**+**», тем не менее в зависимости от типа переменных PVM вызывает разные функции для работы с переменными `a` и `b`. В этом и заключатся полиморфизм в широко распространённом виде: «выглядит одинаково, а действует по-разному» - полиморфное поведение.



Давайте рассмотрим полиморфизм в рамках ООП на языке программирования Python. Пусть у нас имеется класс «Геометрическая точка» - Point у которой должны быть реализованы методы «Показать» - Show() и «Рассчитать площадь» - Area().

Пусть от класса «Геометрическая точка» будет унаследовано два класса «Квадрат» - Square и «Треугольник» - Triangle, как показано на схеме:



Реализуем класс «Геометрическая точка».

```
2   class Point:
3       def _space(self):
4           print('')
5
6       def _print_area(self, value):
7           self._space()
8           print(f'S = {value}')
9           self._space()
10
11      def show(self):
12          self._space()
13          print('*')
14          self._space()
15
16      def area(self):
17          self._print_area(0)
18
```

А теперь создадим экземпляр класса «Геометрическая точка» и выполнить два метода show() и area().

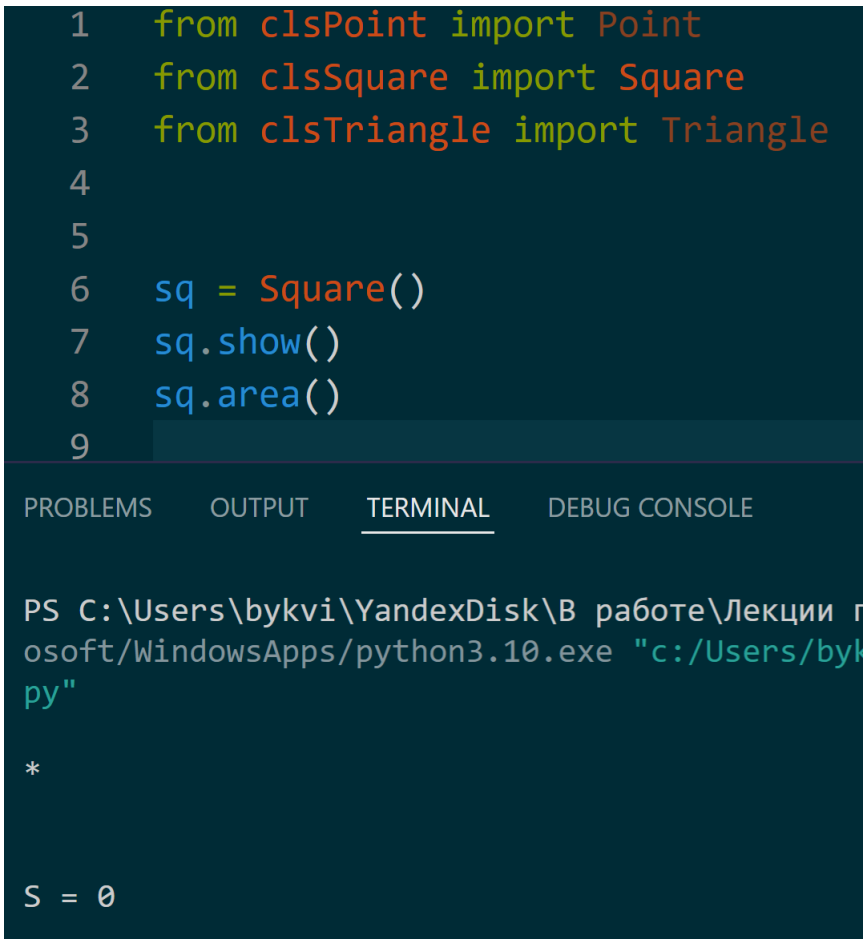
```
from clsSquare import Square
from clsTriangle import Triangle


pt = Point()
pt.show()
pt.area()

```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS C:\Users\bykvi\YandexDisk\В работе\Лекции по
osoft/WindowsApps/python3.10.exe "c:/Users/bykvi
py"

*


S = 0
```

Создадим класс «Квадрат», который унаследован от класса «Геометрическая точка».

```
from clsPoint import Point


class Square(Point):
    pass

```

В таком виде класс «Квадрат» ничем не отличается от своего родительского класса «Геометрическая точка» и имеет уже унаследованные методы show() и area(), а также все защищённые методы. Если мы создадим экземпляр класса «Квадрад», то он ничем не будет отличаться от своего родительского класса.



Теперь давайте переопределим методы show() и area() родительского класса «Геометрическая точка», а также конструктор родительского класса.

```python
from clsPoint import Point


class Square(Point):
    def __init__(self, width):
        self.__width = width

    def show(self):
        self._space()
        for h in range(self.__width):
            s = ''
            for w in range(self.__width):
                if (h == 0) or (h == self.__width - 1):
                    s += '*'
                else:
                    if (w == 0) or (w == self.__width - 1):
                        s += '*'
                    else:
                        s += ' '
            print(s)
        self._space()

    def area(self):
        self._print_area(self.__width**2)

```

А теперь заново вызовем методы show() и area() у экземпляра класса «Квадрат».

А теперь реализуем класс «Треугольник», а также создадим его экземпляр и посмотрим его работу.

```python
from clsPoint import Point


class Triangle(Point):
    def __init__(self, size):
        self.__size = size

    def show(self):
        self._space()
        for h in range(self.__size):
            s = ''
            for w in range(self.__size):
                if h == 0:
                    s += '*'
                else:
                    if (w == 0) or (w == self.__size - h - 1):
                        s += '*'
                    else:
                        s += ' '
            print(s)
        self._space()

    def area(self):
        self._print_area(0.5 * self.__size**2)

```

Таким образом у всех трёх классов имеются методы с одинаковой сигнатурой

```
1   from clsPoint import Point
2   from clsSquare import Square
3   from clsTriangle import Triangle
4
5
6   pt = Point()
7   pt.show()
8   pt.area()
9
10  sq = Square(5)
11  sq.show()
12  sq.area()
13
14  tr = Triangle(5)
15  tr.show()
16  tr.area()
```

За счёт переопределения методов родительского класса получаем полиморфизм в действии – одинаковый вид, но разное действие, которое более подходит для конкретного класса.

При этом для пользователя таких классов создаются удобные условия их использования, необходимо знать только сигнатуры родительского класса всё остальное сделают конкретные классы-потомки, используя механизм полиморфизма.

Список использованных источников

1. Лутц М. Изучаем Python, 4-ое издание. СПб.: Символ-Плюс, 2011. – 1280 с., ил.