Table of Contents

Введение

Эта книга описывает набор уроков для экспресс-курса по питону. Курс состоит из описания синтаксиса и матпакетов. Эта книга - первая часть курса - описание синтаксиса.

Предполагается знание одного из функциональных языков программирования и принципов ООП.

Если нужно подробнее проработать тему, то есть курс Основы Python и Python как первый язык программирования Питон шаг за шагом

Почему Python

Язык простой, универсальный, обилие прикладных пакетов.

Характеристика

Python - интерпретируемый язык, возможна предварительная компиляция и оптимизация кода.

Не нужно заботиться о выделении и освобождении памяти. Освобождением памяти занимается **сборщик мусора**.

Установка

Рекомендуется установить для работы PyCharm Edu (в его составе будут так же несколько кратких курса по питону на английском языке).

Где писать программы

Интерпретатор - попробуем в нем как работает кусок кода. Еще его можно использовать как онлайн-калькулятор.

Запуск файла (или файлов) программ - из командной строки или IDE. Файлы с расширением .py - пишем обычные или математические программы. Кодировка файла ASCII или UTF-8. Если используете русские буквы в строках или комментариях к коду,

выбирайте кодировку UTF-8.

Notebook - в них будем писать математические программы и статьи.

Дополнительные источники (с комментариями)

- **Марк Саммерфилд "Программирование на Python 3. Подробное руководство"** (Programming in Python 3. A Complete Introduction to the Python Language by Mark Summerfield). - Действительно подробное руководство. Курс во многом основан на его книге.
- A Byte of Python (Russian) by Swaroop C H (Translated by Vladimir Smolyar)
- Лутц М. Изучаем Питон. (Mark Lutz. Learning Python) - O'Reilly
- Python Cookbook, 3rd Edition by Brian K. Jones, David Beazley - O'Reilly
- Think Python How to Think Like a Computer Scientist by Allen Downey - рекомендую читать, если изучаете первый язык программирования.
- LEARN PYTHON THE HARD WAY A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code by Zed A. Shaw (тут учат через упражнения, рекомендую).
- http://www.diveintopython.net/ - книга Dive into Python by Mark Pilgrim (есть перевод на русский). На этом сайте много ссылок на материалы и тьюториалы по питону.
- pythontutor.ru - питон с нуля (рекомендую для начинающих). Авторы сайта преподают программирование умным школьникам. Система визуализации кода и автоматической проверки задачи. Много задач. Только первые шаги в питоне. Тут можно изучать питон как первый язык программирования. После него обязательно читать Саммерфилда. Или в параллель читать Think Python, ибо некоторые предлагаемые конструкции не python-way.
- Что читать дальше
  - **Марк Саммерфилд Python in Practice** - паттерны ООП на питоне
  - Python 3 patterns, recipes and idioms

Урок 1. Введение в синтаксис

Создание и запуск программ

Создайте файл hello.py с содержимым

```
print('Hello')
```

Файл можно создать в любом редакторе (Notepad++, vim и так далее) или в специальной IDE (PyCharm).

Запустим файл средствами IDE или из командной строки командой

```
python hello.py
```

При запуске напечатается текст Hello.

Заметим, что в языке не требуется для выполнения создавать функций со специальными именем.

В написанном файле команды выполняются одна за другой.

print - встроенная функция языка. Для ее работы не нужно ничего подключать.

help

Для любой функции можно написать в интерпретаторе help(имя фукнции) и получить ее описание, например **help(print)**.

Для выхода из справки нажмите q

Для справки по операторам напишите их в кавычках, например **help('return')**.

Комментарии

Однострочные комментарии начинаются со знака **#**

Многострочные комментарии можно писать в тройных кавычках.

Переменные и типы данных

Переменные не нужно объявлять заранее. **У переменных нет типа. Тип есть только у данных, на которые указывают переменные**. Для опредения типа используют функцию **type()**

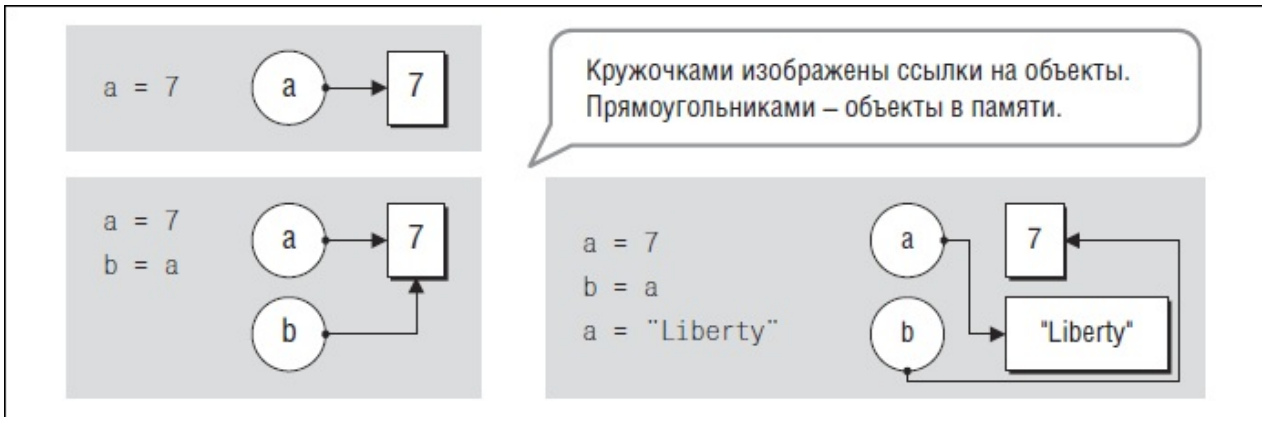
```
x = 5
print(x)          # 5
print(type(x))    # <class 'int'>

x = 3.14
print(x)          # 3.14
print(type(x))    # <class 'float'>

x = 'Hello'
print(x)          # Hello
print(type(x))    # <class 'str'>
```

int, float, str, bool, complex - встроенные типы данных языка. Они **unmutable** (неизменяемые). (Только неизменяемые данные могут быть ключами в словаре).

Ссылки на объекты

Все переменные содержат только ссылки на объекты.

Оператор = связывает переменную с объектом в памяти через ссылку. Если переменная уже существует, то справа от = напишем ссылку на объект, которая будет храниться в переменной. Если переменной еще нет, то оператор = создает переменную и записывает в нее ссылку (которая указана справа от =).



Для упрощения рисунков дальше вместо ссылок на неизменяемые объекты будем рисовать переменные со значением.

```
>>> a = 3
```

1. Создается объект, представляющий число 3 (так как объект неизменяемый, то он создается только если его еще нет, но с логической точки зрения можете считать, что каждый раз создается новый объект).
2. Создается переменная a, если ее еще нет.
3. В переменную a записывается ссылка на объект, представляющий число 3.

Термины: **Переменная** - запись в системной таблице, где предусмотрено место для хранения ссылок на объекты. **Объект** - это область памяти с объемом, достаточным для представления значения этого объекта. **Ссылка** – это автоматически разыменовываемый указатель на объект.

Объект содержит *описатель типа* и *счетчик ссылок*. В описателе типа хранится информация о типе объекта. Счетчик ссылок нужен для автоматического удаления объектов.

Сборка мусора

В Python **объекты удаляются автоматически сборщиком мусора**, если на них нет ссылок. Сборка мусора (**garbage collection**) работает так же для каскадов объектов (на которые нет ссылок, или набор объектов с циклическими ссылками).

TODO: Иллюстрация понятия живых объектов и "мусора".

Принципы работы сборщика мусора:

- Убирается не сразу, когда объект стал мусором, а когда ему вздумается;
- Может не убираться, если есть хватает памяти;
- Можно "попросить убраться".

gc - модуль garbage collector.

Бывает ли memory leak в питоне?

Формально, вы не можете "потерять память". Но если вы держите ссылки на ненужные объекты, они остаются в памяти.

```
x = get_big_big_object()  # x ссылается на большой объект
...
x = None                  # когда объект не нужен, можно перестать на него ссылаться
del x                     # или вообще удалить переменную
```

None - этого не может быть

```
>>> x = None
>>> x
>>> print(x)
None
>>> type(x)
<class 'NoneType'>
>>> x is None
True
>>> 12 is not None    # проверка, что это не None (PEP-8)
True
>>> not 12 is None    # тоже работает, но PEP-8 это не одобряет
True
>>> x == None
True
>>> x != None
False
```

Идентификаторы

Описаны в PEP 3131

- Первый символ должен быть либо _ либо алфавитным символом a-zA-Z либо алфавитным символом большинства алфавитов
- следующие символы могут так же содержать любой непробельный символ (например, цифры и символ Каталана)
- не быть ключевым словом (if, for, while и тп.)
- не пересекаться с предопределенными именами
  - чтобы узнать какие имена предопределены, используйте функцию dir(__builtins__)
- начинаются с __ и заканчиваются им специальные методы: __init__, __lt__ и прочие
- _ и __ в начале идентификатора может влиять на область его видимости

Физические и логические строки. Точка с запятой

Физическая строка - то, что вы видите на экране, набирая код программы.

Логическая строка - то, что питон видит как единое предложение.

Неявно предполагается, что одной физической строке соотвествует одна логическая.

Последним символом строки можно поставить \ и написать одну логическую строку на двух физических.

Можно можно на одной физической строке написать несколько логических, разделяя их ;

```
i = i+5; print(i)  # можно поставить ; в конце каждой строки, как вы привыкли в C++
```

Не ставьте лишние символы ; Они зашумляют чтение кода и показывают, что это "ваша первая программа на питоне".

Отступы

Отступы - это пробелы или табуляции в *начале* строки. Они важны.

Уровни вложенности в питоне обозначаются не скобками, а отступами.

Придерживайтесь одного стиля отступов - ЛИБО пробелы (рекомендовано 4), ЛИБО табуляции (одна). Не смешивайте их. В другом редакторе могут быть другие настройки табуляции и отступов (например, 1 табуляция = 8 пробелов) и тогда ваша логическая структура программы приобретет совсем другой вид.

Этот код не будет работать:

```
i = i+5
 print(i)       # Ошибка, лишний пробел в начале строки
print('Hello')
```

Поменять местами значение переменных x и y

```
x, y = y, x
```

Как это работает обсудим позже.

str

Строки можно писать в одинарных, двойных и тройных кавычках. Строки могут содежать любые символы юникода.

```
'Hello'
"Здесь могут быть русские буквы или другие символы юникода"
'''Текст на
несколько строк'''
```

Строки - неизменяемый тип данных. Нельзя изменить существующую строку, можно создать новую.

Подробнее строки и операции над ними будут рассмотрены позже.

Все типы, описанные в разделе, являются **неизменяемыми**.

```
print(7/2)    # 3.5 - обычное математическое деление
print(7//2)   # 3 - целочисленное деление
print(7%2)    # 1 - остаток от деления
```

Целочисленные типы

- Логические выражения:
  - False и 0 - это False
  - True и любые числа кроме 0 - это True
- Численные выражения:
  - True - это 1
  - False - это 0

```
i += True     # синтаксически верно и увеличивает на 1, но НЕ ПИШИТЕ ТАК
i += 1        # то же самое, лучше писать так

x = bool(7)   # x == True
x = bool(-7)  # x == True
x = bool(0)   # x == False
```

int

int - целые положительные и отрицательные числа. Размер не ограничен.

```
x = 2**217  # 2 в степени 217
print(x)    # 210624583337114373395836055367340864637790190801098222508621955072
```

Как создаются целочисленные объекты?

Для других типов объектов похожие правила. Вызывается функция int

- без аргументов int() - создаст целое число 0. Аналогично любые другие встроенные типы данных без аргумента создают какой-то объект со значением по умолчанию;
- с одним аргументом
  - int(12) - поверхностная копия
  - int('123'), int(4.56) - если тип аргумента подходящий, то происходит преобразование. Если оно неудачное, например, int('Hello'), то возбуждается исключение ValueError, если тип неподходящий, например, int([1, 2, 3]), то

исключение TypeError

- с двумя аргументами int(x, base) - аналогично, ибо в одном аргументе по умолчанию base=10 (подробнее далее)

Функции преобразования целых чисел

Синтаксис	Описание
bin(i)	Возвращает двоичное представление целого числа i в виде строки, например, bin(1980) == '0b11110111100'
hex(i)	Возвращает шестнадцатеричное представление целого числа i в виде строки, например, hex(1980) == '0x7bc'
int(x)	Преобразует объект x в целое число; в случае ошибки во время преобразования возбуждает исключение ValueError, а если тип объекта x не поддерживает преобразование в целое число, возбуждает исключение TypeError. Если x является числом с плавающей точкой, оно преобразуется в целое число путем усечения дробной части.
int(s, base)	Преобразует строку s в целое число, в случае ошибки возбуждает исключение ValueError. Если задан необязательный аргумент base, он должен быть целым числом в диапазоне от 2 до 36 включительно.
oct(i)	Возвращает восьмеричное представление целого числа i в виде строки, например, oct(1980) == '0o3674'

```
>>> s1 = bin(10)   # получаем из числа 10 строки для представления числа в бинарной, в
осьмеричной и шестнадцатеричной системах счисления
>>> s1
'0b1010'
>>> s2 = oct(10)
>>> s2
'0o12'                  # Заметьте, что префикс **0o**, а не просто 0, как в языке C.
>>> s3 = hex(10)
>>> s3
'0xa'
>>> x1 = int('10')      # из строки - десятичного представления числа - получаем целое
 число типа int
>>> x1; type(x1)
10
<class 'int'>
>>> x1 = int('  10  ')  # **допускает пробельные символы до и после числа**
>>> x1
10
>>> x2 = int(s1, 2)     # обратно из бинарной строки в целое число
>>> x2; type(x2)
10
<class 'int'>
>>> x2 = int('111', 2)  # обратно из бинарной строки в целое число
>>> x2
7
>>> x3 = int('z', 36)   # параметр base может быть от 2 до 36, цифры тогда 0-9a-zA-Z
>>> x3
35
>>> x3 = int('Az', 36)
>>> x3
395
```

Для преобразования стандартных типов данных используются функци

- **int(x, base=10)**
- **float(x)**
- **str(x)**
- **bool(x)**
- **complex(re, im)**

Действия над числами

Синтаксис	Описание
x + y	Складывает число x и число y
x - y	Вычитает число y из числа x
x * y	Умножает x на y
x / y	Делит x на y – результатом всегда является значение типа float (или complex, если x или y является комплексным числом)
x // y	Делит x на y, результат округляется вниз -7//2 == -4
x % y	Возвращает модуль (остаток) от деления x на y. Для комплексных чисел преобразуется с помощью abs() к float
x ** y	Возводит x в степень y, смотрите также функцию pow()
-x	Изменяет знак числа. 0 остается нулем.
+x	Ничего не делает, иногда используется для повышения удобочитаемости программного кода
abs(x)	Возвращает абсолютное значение x
divmod(x, y)	Возвращает частное и остаток деления x на y в виде кортежа двух значений типа int
pow(x, y)	Возводит x в степень y; то же самое, что и оператор **
pow(x, y, z)	Более быстрая альтернатива выражению (x ** y) % z
round(x, n)	Возвращает значение типа int, соответствующее значению x типа float, округленному до ближайшего целого числа (или значение типа float, округленное до n-го знака после запятой, если задан аргумент n)

Комбинированные операторы присвоения

+= -= *= /= //= %= и прочие операторы

Как работает ++ и --

Никак. Их нет в языке. Используйте x += 1 и x -= 1. В циклах можно обойтись без них.

Побитовые операции над целыми числами

Синтаксис	Описание	
i | j	Битовая операция OR (ИЛИ) над целыми числами i и j; отрицательные числа представляются как двоичное дополнение	
i ^ j	Битовая операция XOR (исключающее ИЛИ) над целыми числами i и j	
i & j	Битовая операция AND (И) над целыми числами i и j	
i << j	Сдвигает значение i влево на j битов аналогично операции i * (2 ** j) без проверки на переполнение	
i >> j	Сдвигает значение i вправо на j битов аналогично операции i // (2 ** j) без проверки на переполнение	
~i	Инвертирует биты числа i	

bool

True и **False** - константы.

Операторы and, or, not.

НЕ используйте 1 и 0 вместо True и False (понижает читаемость кода).

Числа с плавающей точкой

В стандартной библиотеке это float, complex и decimal.Decimal

float

Константы записываются как 0.0, 4., 5.7, –2.5, –2e9, 8.9e–4.

Обычно реализованые типом double языка C. Подробнее смотри sys.float_info Поэтому часть чисел может быть представлена точно (0.5), а часть - только приблизительно (0.1).

Можно написать простую функцию, которая сравнивает числа с машинной точностью:

```
def equal_float(a, b):
    return abs(a - b) <= sys.float_info.epsilon
```

Округление

Все numbers.Real типы (int и float) могут быть аргументами следующих функций :

Операция	Результат
int(x)	Округляет число в сторону нуля. Это стандартная функция, для ее использования не нужно подключать модуль math
round(x)	Округляет число до ближайшего целого. Если дробная часть числа равна 0.5, то число округляется до ближайшего четного числа
round(x, n)	Округляет число x до n знаков после точки. Это стандартная функция, для ее использования не нужно подключать модуль math
math.floor(x)	Округляет число вниз («пол»), при этом floor(1.5) = 1, floor(-1.5) = -2
math.ceil(x)	Округляет число вверх («потолок»), при этом ceil(1.5) = 2, ceil(-1.5) = -1
math.trunc(x)	Отбрасывает дробную часть
float.is_integer()	True, если дробная часть 0.
float.as_integer_ratio()	Возвращает числитель и знаменатель дроби

Примеры:

Выражение	Результат
int(1.7)	1
int(-1.7)	-1
ceil(4.2)	5
ceil(4.8)	5
ceil(-4.2)	-4
round(1.3)	1
round(1.7)	2
round(1.5)	2
round(2.5)	2
round(2.65, 1)	2.6
round(2.75, 1)	2.8
trunc(5.32)	5
trunc(-5.32)	-5

```
>>> x = 3.0
>>> x.is_integer()
True
>>> x = 2.75
>>> a, b = x.as_integer_ratio()  # помним, что функция может возвращать несколько знач
ений
>>> a                            # числитель
11
>>> b                            # знаменатель
4
```

Сюрпризы:

```
round(2.85, 1)   # 2.9
```

подробнее: Floating Point Arithmetic: Issues and Limitations

Функции и константы модуля math

Не забудьте сделать import math

Синтаксис	Описание
math.acos(x)	Возвращает арккосинус x в радианах

math.acosh(x)	Возвращает гиперболический арккосинус x в радианах		
math.asin(x)	Возвращает арксинус x в радианах		
math.asinh(x)	Возвращает гиперболический арксинус x в радианах		
math.atan(x)	Возвращает арктангенс x в радианах		
math.atan2(y, x)	Возвращает арктангенс y/x в радианах		
math.atanh(x)	Возвращает гиперболический арктангенс x в радианах		
math.ceil(x)	Возвращает наименьшее целое число типа int, большее и равное x, например, math.ceil(5.4) == 6		
math.copysign(x,y)	Возвращает x со знаком числа y		
math.cos(x)	Возвращает косинус x в радианах		
math.cosh(x)	Возвращает гиперболический косинус x в радианах		
math.degrees(r)	Преобразует число r, типа float, из радианов в градусы		
math.e	Константа e, примерно равная значению 2.7182818284590451		
math.exp(x)	Возвращает e в степени x, то есть math.e ** x		
math.fabs(x)	Возвращает \	x\	, то есть абсолютное значение x в виде числа типа float
math.factorial(x)	Возвращает x!		
math.floor(x)	Возвращает наименьшее целое число типа int, меньшее и равное x, например, math.floor(5.4) == 5		
math.fmod(x, y)	Выполняет деление по модулю (возвращает остаток) числа x на число y; дает более точный результат, чем оператор %, применительно к числам типа float		
math.frexp(x)	Возвращает кортеж из двух элементов с мантиссой (в виде числа типа float) и экспонентой (в виде числа типа int)		
	Возвращает сумму значений в		

math.fsum(i)	итерируемом объекте i в виде числа типа float
math.hypot(x, y)	Возвращает расстояние от точки (0,0) до точки (x,y) $$\sqrt{x^2 + y^2}$$
math.isinf(x)	Возвращает True, если значение x типа float является бесконечностью $$± \infty$$
math.isnan(x)	Возвращает True, если значение x типа float не является числом
math.ldexp(m, e)	Возвращает $$m \cdot 2^x$$ – операция, обратная math.frexp
math.log(x, b)	Возвращает $$\log_{b} {x}$$, аргумент b является необязательным и по умолчанию имеет значение math.e
math.log10(x)	Возвращает $$\log_{10} {x}$$
math.log1p(x)	Возвращает $$\log_{e} (1+x)$$; дает точные значения, даже когда значение x близко к 0
math.modf(x)	Возвращает дробную и целую часть числа x в виде двух значений типа float
math.pi	Константа $$\pi$$, примерно равна 3.1415926535897931
math.pow(x, y)	Возвращает $$x^y$$ в виде числа типа float
math.radians(d)	Преобразует число d, типа float, из градусов в радианы
math.sin(x)	Возвращает синус x в радианах
math.sinh(x)	Возвращает гиперболический синус x в радианах
math.sqrt(x)	Возвращает $$\sqrt{x}$$
math.sum(i)	Возвращает сумму значений в итерируемом объекте i в виде числа типа float
math.tan(x)	Возвращает тангенс x в радианах
math.tanh(x)	Возвращает гиперболический тангенс x в радианах
math.trunc(x)	Возвращает целую часть числа x в виде значения типа int; то же самое, что и int(x)

Бесконечность и NaN (not a number)

Создаем бесконечность, отрицательную бесконечность и NaN

```
>>> a = float('inf')
>>> b = float('-inf')
>>> c = float('nan')
>>> a
inf
>>> b
-inf
>>> c
nan
>>>
```

Проверяем эти особые значения функциями math.isinf() и math.isnan()

```
>>> math.isinf(a)
True
>>> math.isinf(b)
True
>>> math.isnan(c)
True
>>> b < 0
True
```

Операции над бесконечностью интуитивно понятны математику:

```
>>> a = float('inf')
>>> a + 45
inf
>>> a * 10
inf
>>> 10 / a
0.0
```

Когда получается nan ?

```
>>> a = float('inf')
>>> a/a
nan
>>> b = float('-inf')
>>> a + b
nan
```

Из nan получается только nan

```
>>> c = float('nan')
>>> c + 23
nan
>>> c / 2
nan
>>> c * 2
nan
>>> math.sqrt(c)
nan
```

Не сравнивайте nan

```
>>> c = float('nan')
>>> d = float('nan')
>>> c == d
False
>>> c is d
False
```

Получить исключение, когда результат nan

Иногда при отладке нужно поймать, где возникло "не то значение". Используйте модуль fpectl Floating point exception control модуль не входит в стандартную поставку.

complex

Неизменяемый тип данных. Состоит из двух float чисел - действительной и мнимой части.

Для представления комплесных чисел есть встроенный тип complex. Мнимая единица обозначается как j или J. Если действительная часть 0, то ее можно не писать.

Примеры комплексных чисел: 3.5+2j, 0.5j, -7.24+0j

```
>>> x = 3.5 + 2.1j
>>> x.real, x.imag  # действительная и мнимая часть
(3.5, 2.1)
>>> x.conjugate()   # изменили знак мнимой части
(3.5-2.1j)
```

При конвертации из строки вокруг + или - НЕ должно быть пробелов

```
x = complex(3, 5) # 3+5j
x = complex(3.5) # 3.5+0j
x = complex(' 3+5j ') # 3+5j
x = complex('3 + 5j') # ОШИБКА
```

С комплексными числами работают привычные операторы и функции, за исключением //, %, divmod, pow с тремя аргументами (см. таблицу далее)

Для работы с комплексными числами используйте функции из модуля cmath.

Decimal

Если нужна бОльшая точность, чем дает float, используйте decimal.Decimal.

Эти числа обеспечивают уровень точности, который вы укажете (по умолчанию 28 знаков после запятой), и могут точно представлять периодические числа, такие как 0.11, но скорость работы с такими числами существенно ниже, чем с обычными числами типа float. Вследствие высокой точности числа типа decimal.Decimal прекрасно подходят для производства финансовых вычислений.

Различные типы в бинарных операторах

Тип операторов	Тип результата
int float	float
float complex	complex
Decimal int	Decimal

Decimal может использоваться только с Decimal или int, нельзя его использовать с float и другими неточными типами.

Функции

Функции - это часть кода, к которой мы обращаемся по имени.

Нужны для:

- повторное использование кода (а не его copy-paste);
- разбить задачу на подзадачи

Вы уже пользовались функциями языка python. Это print(), input(), int(), float().

Можно написать функцию самим.

Простая функция

Напишем функцию, у которой нет аргументов и которая ничего не возвращает. Придумаем имя функции hi. Функция печатает hello.

```
# делаем функцию.
# def - ключевое слово
# hi - придумали (сами) имя функции
def hi():
    print("hello")   # код функции пишем с отступами

# закончились отступы - закончилась функция.
hi()   # вызов функции hi, функция печатает hello
hi()   # вызов функции hi, функция печатает hello
```

Не забывайте : после)

Передаем в функцию числа

Напишем вычисление периметра и площади прямоугольника через функции. Тогда можно будет просто посчитать периметр и площадь разных прямоугольников в одной программе.

У функции могут быть аргументы. Для вычисления периметра и площади прямоугольника нужно передать функции стороны прямоугольника.

Один раз создали функцию. Много раз можем использовать функцию.

```
def perimetr(a, b):  # создали первую функцию perimetr, в нее передают два числа a и b
  res = (a+b)*2
  return res         # возвращает число
                     # первая функция закончилась

def area(a, b):      # создали другую функцию area, в нее передают два числа a и b
  res = a*b
  return res

p = perimetr(3,5)              # результат функции perimetr поместили в переменную p
print("Периметр = %d" % (p))   # напечатали p (Периметр = 16)
s = area(3,5)                  # результат функции area поместили в переменную s
print("Площадь = %d" % (s))    # напечатали s (Площадь = 15)

                               # можно сразу печатать результат функции
print("Периметр = %d" % (perimetr(3,5)))
print("Площадь = %d" % (area(3,5)))

p = perimetr(3.3, 5)           # функция может считать и дробные числа
print("Периметр = %f" % (p))   # напечатали p по формату %f (Периметр = 16.6)
s = area(3.3, 5)               # результат функции area поместили в переменную s
print("Площадь = %f" % (s))    # напечатали s по формату %f (Площадь = 16.5)
```

Возвращаем несколько значений

Функция может возвращать несколько значений. Их пишут через запятую (,)

На самом деле передается один кортеж (tuple). TODO: ссылку на главу с новым термином.

Функции height передаем рост в сантиметрах, а возвращает функция рост в метрах и сантиметрах

```
def height(h):         # функция height, в нее передают одно число h
   m = h // 100        # подсчитали рост в метрах
   sm = h % 100        # подсчитали рост в сантиметрах
   return m, sm        # вернули сразу метры и сантиметры

# дальше программа. Пользуемся функцией height и проверяем ее.
                       # мой рост 169 см. Посчитаем его в метрах и сантиметрах
mym, mysm = height(169)   # результаты функции поместили в переменные mym и mysm
print("мой рост %d метров %d сантиметров" % (mym, mysm))

you = int(input())        # прочитали ваш рост
ym, ysm = height(you)     # результаты функции поместили в переменные ym и ysm
print("ваш рост %d метров %d сантиметров" % (ym, ysm))
```

Функция вызывает функцию

Напишем программу, которая по координатам 2 точек на плоскости считает расстояние между ними.

```
from math import sqrt

def length(x1, y1, x2, y2):       # создали функцию length
  dx = x1 - x2
  dy = y1 - y2
  res = sqrt(dx*dx + dy*dy)
  return res

x1, y1, x2, y2 = map(int, input().split()) # прочитали сразу много чисел из 1 строки

dist = length(x1, y1, x2, y2)     # результат работы функции length записали в dist
print(dist)
```

Функция length(x1, y1, x2, y2) считает расстояние между 2 точками на плоскости.

Теперь напишем другую программу. Которая по координатам 3 точек на плоскости считает площадь треугольника по формуле Герона.

Нужно писать мало кода. Возьмем старую функцию length и используем ее.

```
from math import sqrt

def length(x1, y1, x2, y2):         # функция length уже написана и проверена
  dx = x1 - x2
  dy = y1 - y2
  res = sqrt(dx*dx + dy*dy)         # из функции length вызываем функцию sqrt
  return res

def area3(x1, y1, x2, y2, x3, y3):  # новая функция area3
  a = length(x1, y1, x2, y2)        # из функции area3 вызываем функцию length
  b = length(x1, y1, x3, y3)        # из функции area3 вызываем функцию length
  c = length(x3, y3, x2, y2)        # из функции area3 вызываем функцию length
  p = (a+b+c)/2                     # записываем формулы
  res = sqrt(p*(p-a)*(p-b)*(p-c))
  return res

x1, y1, x2, y2, x3, y3 = map(int, input().split())
s = area3(x1, y1, x2, y2, x3, y3)
print(s)
```

Не обязательные аргументы

Задача: написать функцию, которая считает расстояние до точки (x,y) на плоскости от начала координат (0,0)

Вариант 1. Самый плохой, потому что нужно писать много кода и отлаживать его. Можем ошибиться при написании формулы.

```
def length0(x, y):       # создали функцию length0
  res = sqrt(x*x + y*y)
  return res
```

Вариант 2. Лучше. Пишем еще одну функцию, которая использует функцию length

```
def length0(x, y):       # создали функцию length0
  return length(x, y, 0, 0)
```

Вариант 3. Хорошо. Не нужно писать новый код.

Когда пишем функцию length(x1, y1, x2, y2) записываем в x2 и y2 значения по умолчанию 0. Аргументы x2 и y2 стали не обязательными. Можно вызвать функцию без этих аргументов, а она будет работать так, будто их значение 0.

Значение по умолчанию можно сделать любое. Не обязательно 0.

```
def length(x1, y1, x2=0, y2=0):       # создали функцию length
  dx = x1 - x2
  dy = y1 - y2
  res = sqrt(dx*dx + dy*dy)
  return res
```

Когда вызываем функцию length, можем передавать все параметры, а можем не передавать x2 и y2. Тогда их значение будет по умолчанию 0.

```
d = length(3, 4, 3, -4)     # расстояние между точками (3, 4) и (3, -4)
d = length(3, 4, 3)         # расстояние между точками (3, 4) и (3, 0)
d = length(3, 4)            # расстояние между точками (3, 4) и (0, 0)
```

Именованные аргументы

Мы передавали аргументы в функцию по их позиции.

```
d = length(3, 4, 3, -4)     # расстояние между точками (3, 4) и (3, -4)
```

И понимали, что -4 - это значение аргумента y2.

Можно передавать аргументы в функцию по имени аргумента.

```
d = length(3, 4, x2=5, y2=-4)     # расстояние между точками (3, 4) и (5, -4)
d = length(3, 4, y2=5, x2=-4)     # расстояние между точками (3, 4) и (-4, 5)
                                  # порядок вызова аргументов по имени НЕ важен
d = length(x1=3, x2=4, y1=5, y2=-4)     # расстояние между точками (3, 5) и (4, -4)
                                  # это тоже работает, потому что аргументы вызваны по
 имени и порядок не важен
d = length(3, y1=4, x2=5, y2=-4)  # расстояние между точками (3, 4) и (5, -4)
                                  # любой аргумент можно вызвать по имени
d = length(x1=3, 4, x2=3, y2=-4)  # ОШИБКА! сначала аргумент по имени, потом - нет.
```

Если вызван аргумент по имени, все аргументы после него должны вызваться по имени

Проверить функцию - assert

Напишем функцию вычисления периметра и проверим, что она правильная.

```
def perimetr(a, b):  # создали первую функцию perimetr, в нее передают два числа a и b
  res = (a+b)        # Ошибка! Забыли *2
  return res         # возвращает число

# Проверим функцию perimetr
print(perimetr(3,5)) # периметр должен быть равен 16
```

Программа напечатает 8. Мы посмотрим на perimetr(3,5) и посчитаем, что периметр должен равняться 16.

Нашли, что функция perimetr работает неправильно. Надо исправить.

И проверить еще раз.

Легче проверять, если печатать что посчитали и какое число должно быть.

```
print(perimetr(3,5), 16) # должно напечатать 16 и 16
print(perimetr(7,2), 18) # должно напечатать 18 и 18
print(perimetr(5,5), 25) # должно напечатать 25 и 25
```

Печатаются числа. Надо посмотреть, что числа одинаковые. Если числа разные - ошибка.

Можно заставить проверять компьютер. **assert**(выражение) - проверяет, правильное выражение или нет. Если правильное, то ничего не делает. Если неправильное, печатает где ошибка.

```
assert(perimetr(3,5)==16) # проверить, что perimetr(3,5) вернул 16
assert(perimetr(7,2)==18) # проверить, что perimetr(7,2) вернул 18
assert(perimetr(5,5)==25) # проверить, что perimetr(5,5) вернул 25
```

Если функция возвращает несколько значений, то их пишем в ( ) через ,

Функция msm из роста в сантиметрах (157) вычисляет рост в метрах (1) и сантиметрах (57). Возвращает метры и сантиметры (1, 57)

Напишем функцию и проверим ее.

```
def msm(h):
    m = h//100
    sm = h % 100
    return m, sm

print(msm(157), 1, 57)      # напечатает (1 57) 1 57 - можно проверить глазами
assert(msm(157))==(1, 57))  # программа сама проверит, что msm(157) вернет 1 и 57
```

Кратко о локальных и глобальных переменных

Подробнее о namespace читайте в следующем разделе. Здесь будет короткое описание основных моментов.

Область видимости (пространство имен) - область, где хранятся переменные. Здесь определяются переменные и делают поиск имен.

Операция = связывает имена с областью видимости (пространством имен)

Пока мы не написали ни одной функции, все переменные в программе глобальные.

Глобальные переменные видны во всех функциях программы. Они должны быть сначала созданы, а потом их можно читать и менять.

Создаются переменные присвоением =

Обычно пишут программу так:

```
a = 1           # создали раньше, чем ее использовали

def f():
    print(a)    # читаем глобальную переменную a (не изменяя ее значения)

f()             # тут (дальше) мы использовали переменную a
```

Этот код будет работать так же:

```
def f():
    print(a)    # глобальная переменная a
a = 1           # создали раньше, чем ее использовали
f()             # тут (дальше) мы использовали переменную a
```

Напечатает 1. Глобальная переменная a *сначала* была создана, а *потом* была вызвана функция f(). В функции f() видна глобальная переменная a.

Особенность интерптерируемого языка. *Сначала* - это раньше в процессе выполнения, а не "на строке с меньшим номером".

Локальная переменная создается внутри функции или блока (например, if или while). Локальная переменная видна только внутри того блока (функции), где была создана.

```
def f():
    a = 1   # локальная переменная функции f
f()
print(a)    # ошибка, локальная переменная a не видна вне функции f.
```

Ошибка "builtins.NameError: name 'a' is not defined"

- Локальные переменные создаются =.
- Каждый вызов функции создает локальную переменную (свою, новую) (каждый вызов функции создает свой новый namespace)
- после завершения функции ее локальные переменные уничтожаются.
- **аргументы функции тоже являются локальными переменными** (при вызове функции идет = параметру значения).

Итого: **Если в функции было =, то мы создали локальную переменную. Если = не было, то читаем глобальную переменную.**

Можно создавать в разных функциях локальные переменные с одинаковыми именами. В функциях foo и bar создали переменные с одинаковыми именами a.

Можно (но не надо так делать!) создавать локальную переменную с тем же именем, что и глобальную. `pylint` поможет найти такие переменные.

```
def f():
    a = 1               # создана локальная переменная a=1
    print(a, end=' ')   # печатаем локальную переменную a=1
a = 0                   # создана глобальная переменная a=0
f()
print(a)                # печатаем глобальную переменную a=0
```

Напечатает `1 0` .

1. создается глобальная переменная a = 0
2. вызывается f()
3. в f создается локальная переменная a = 1 (теперь нельзя доступиться из функции f к глобальной переменной a)
4. в f печатается локальная переменная a = 1
5. завершается f
6. печатается глобальная переменная a = 0

Переменная в функции будет считаться локальной, если она будет создана внутри условного оператора, который никогда не выполнится:

```
def f():
    print(a)    # UnboundLocalError: local variable 'a' referenced before assignment
    if False:
        a = 0   # тут создаем локальную переменную a внутри функции f
a = 1           # глобальная переменная a
f()
```

global говорит, что переменная относится к глобальному namespace. (В этот момент переменная НЕ создается). Переменную можно создать позже.

```
def f():
    global a
    a = 1
    print(a, end=' ')
a = 0
f()
print(a)
```

выведет "1 1", т.к. значение глобальной переменной будет изменено внутри функции.

Рекурсивный вызов функции

Так как каждый *вызов* функции создает свое собственное пространство имен, можно писать функции рекурсивно.

Например, n! = n * (n-1)!, 0! = 1. Запишем это математическое определение факториала в виде кода.

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n-1)

print(fact(5))
```

При вызове fact(5) создается namespace с n=5, далее идет вызов f(4) и создается еще один namespace, в нем n=4 (это другая переменная n, она в другом пространстве имен и та n=5 из этого пространства не доступна).

Вложенные области видимости

Можно определять одну функцию внутри другой.

Чтение переменной внутри функции. Ищем имя:

- в локальной области видимости функции;
- в локальных областях видимости объемлющих функций **изнутри наружу**;
- в глобальной области видимости модуля;
- в builtins (встроенная область видимости).

`x = value` внутри функции:

- создает или изменяет имя x в текущей локальной области видимости функции;
- если был `unlocal x`, то = создает или изменяет имя в *ближайшей* области видимости объемлющей функции.
- если был `global x`, то = создает или изменяет имя в области видимости объемлющего модуля.

```
X = 99              # Имя в глобальной области видимости: не используется
def f1():
    X = 88          # Локальное имя в объемлющей функции
    def f2():
        print(X)    # Обращение к переменной во вложенной функции
    f2()
f1()                # Выведет 88: локальная переменная в объемлющей функции
f2()                # Ошибка! функция f2 здесь не видна!
```

В f2() нельзя изменить значение X, принадлежащей функции f1(). Вместо этого будет создана еще одна локальная переменная, но уже в пространстве имен функции f2(). Напечатает `77 88`:

```
X = 99              # Имя в глобальной области видимости: не используется
def f1():
    X = 88          # Локальное имя в объемлющей функции
    def f2():
        X = 77      # создаем локальную переменную
        print(X)    # 77 - обращение к локальной переменной функции f2()
    f2()
    print(X)        # 88 - обращение к локальной переменной функции f1()
f1()
```

Если нужно *изменять* значение переменной X, которая принадлежит пространству имен объемлющей (enclosed) функции, то добавляют **unlocal**

```
X = 99              # Имя в глобальной области видимости: не используется
def f1():
    X = 88          # Локальное имя в функции f1
    def f2():
        unlocal X   # X принадлежит объемлющей функции
        X = 77      # изменяем переменную функции f1
        print(X)    # 77 - обращение к локальной переменной объемлющей функции f1()
    f2()
    print(X)        # 77 - обращение к локальной переменной функции f1()
f1()
```

Правило LEGB

При определении, к какому namespace относится имя, используют правило LEGB:

- Когда внутри функции выполняется обращение к неизвестному имени, интерпретатор пытается отыскать его в четырех областях видимости – в локальной (local, L), затем в локальной области любой объемлющей инструк- ции def (enclosing, E) или в выражении lambda, затем в глобальной (global, G) и, наконец, во встроенной (built-in, B).
    - Поиск завершается, как только будет найдено первое подходящее имя.
    - Если требуемое имя не будет найдено, интерпретатор выведет сообщение об ошибке.

Функции подробнее

Описано на основе

- Лутц, Изучаем Python. Глава 16. Основы фукций; Глава 17

Преимущества функциий vs copy-paste

- пишем один раз, используем много раз (отлаживаем тоже один раз!);
- аргументы позволяют работать с разными входными данными;
- если нашли ошибку в функции, испраляем ее в одном месте, а не по всему коду программы (развитие функций);
- функции в модуле можно поместить в другую программу и использовать там (еще больше reuse).

Принципы работы функций в питоне

- **def - исполняемый программный код**
  - Функция не существует, пока до нее не дошел интерпретатор и не выполнил ее.
  - Можно вставлять def внутрь циклов, if или в другие инструкции def.
- **def создает объект и присваивает ему имя**
  - как в операции =, имя - ссылка на объект-фукнцию.
  - можно сделать несколько ссылок, сохранить в списке и тп.
  - к функции можно прикрепить определяемые пользователем атрибуты.
- **выражение lambda создает объект и возвращает его в виде результата**
  - это лямбда-фукнции
- **return передает объект результата вызывающей программе**
  - вызывающий код приостанавливает свою работу, запоминается откуда вызывается функция;
  - управление передается в функцию;
  - выполняется функция и быть может возвращает значение (если есть return);
  - значение вызванной функции равно возвращаемому значению при этом вызове.
- **yield передает объект результата вызывающей программе и запоминает, где был произведен возврат**

    - это функции-генераторы
- **аргументы передаются присваиванием и по ссылке**
    - переменной-аргументу присваивается ссылка на передаваемый объект.
- **global объявляет переменные, глобальные для модуля, без присваивания им значений**
    - по умолчанию все имена, которым присваиваются значения, **являются локальными для этой функции и существуют только во время выполнения фукнции**.
    - `global myvar` - внутри функции делает переменную myvar видимой в текущем модуле.
- **nonlocal объявляет переменные, находящиеся в области видимости объемлющей функции, без присваивания им значений** (Python 3)
    - храним в них информацию о *состоянии* - информация восстанавливается в момент вызова функции, можно обойтись без `global`
- **аргументы, возвращаемые значения и переменные НЕ объявляются**

Далее рассмотрим эти концепции с примерами кода.

def исполняют во время выполнения

Нет понятия компиляции кода. Можно написать так:

```
if test:
    def func(): # Определяет функцию таким способом
...
else:
    def func(): # Или таким способом
...
...
func()          # Вызов выбранной версии
```

def похож на оператор присваивания (=).

- def НЕ интерпретируются, пока они не будут достигнуты;
- код *внутри* функции НЕ выполняется, пока функция не вызвана.

Свяжем код функции с другим именем:

```
myfunc = func  # связывание объекта-функции с именем myfunc
myfunc()       # вызов функции (не важно по какому имени)
```

Атрибуты функции

К функции можно присоединить атрибут. В них можно сохранять информацию.

```
def func(): ...   # Создает объект функции
func()            # Вызывает объект
func.attr = value # Присоединяет атрибут attr к объекту и записывает в него значение v
alue
```

Ничего не возвращает

Не обязательно в фукнции писать return (или yield). Функция может ничего не возвращать.

Результат вызова такой функции **None**

```
>>> def hi(name):
...     print('Hello,', name)
...
>>> x = hi('Mike')
Hello, Mike
>>> x
None
```

Полиморфизм

Создание функцию:

```
>>> def times(x, y):  # Создать функцию и связать ее с именем
...     return x * y  #Тело, выполняемое при вызове функции
```

Вызов функции:

```
>>> times(2, 4)
8
>>> times('Hi', 3)
HiHiHi
```

Одна и та же функция times используется для разного: умножения чисел и повторения последовательностей.

Полиморфизм означает, что смысл операций зависит от типов операндов.

Динамическая типизация и полиморфизм - основа языка питон. Не нужно в функциях писать ограничения на типы. Либо операция поддерживает тип операнда, либо автоматически возбуждается исключение. Не ломайте этот механизм.

Возможный минус - несовпадение типов обнаруживается только при выполнении (и только если этот код вызван). Покрывайте код тестами.

Функция ищет пересечение двух последовательностей:

```
def intersect(seq1, seq2):
    res = []
    for x in seq1:
        if x in seq2:
            res.append(x)
    return res
```

Заметим, что в функции первый аргумент должен быть итерируемым (с ним дожен работать for), а второй аргумент - поддерживать оператор in. Других ограничений на объекты нет.

Этот же код (как будет рассказано дальше), можно записать в виде генератора списков:

```
[x for x in seq1 if x in seq2]
```

Локальные переменные

Локальная переменная - имя, которое доступно только внутри инструкции def и существует только во время выполнения функции (после окончания функции эти переменные уничтожаются).

```
def intersect(seq1, seq2):
    res = []
    for x in seq1:
        if x in seq2:
            res.append(x)
    return res
```

Здесь локальные переменные:

- *res* - потому что ей присваивают;
- аргументы передаются через операцию присваивания, поэтому *seq1* и *seq2* - локальные переменные;

- цикл for присваивает значения переменной *x*, поэтому она тоже локальная переменная.

Они появляются при вызове фукнции и исчезают после ее окончания.

Область видимости (пространство имен, namespace)

Область видимости (пространство имен) - область, где хранятся переменные. Здесь определяются переменные и делают поиск имен.

Операция = связывает имена с областью видимости (пространством имен)

Каждая функция добавляет свое пространство имен:

- имена внутри def видны только внутри def.
- можно делать одинаковые имена (x) в разных простанствах имен (другом def или во всем модуле).

Области видимости:

- **локальная** переменная - присвоение внутри def; видна только внутри def.
- **нелокальная** переменная - в пределах объемлющей инструкции def; там и видна.
- **глобальная** переменная - вне всех def; глобальная для всего файла.

Лексическая область видимости - потому что определяется тем, где первый раз было =.

Модули и функции

- **Модуль - глобальная область видимости**
  - пространство имен, где создаются переменные на верхнем уровне в файле модуля.
  - глобальные переменные с точки зрения:
    - внешнего мира - это атрибуты модуля;
    - внутри модуля - это обычные переменные.
- Глобальная область видимости - это ОДИН файл.
  - слышим "глобальный" подразумеваем "один модуль", более глобального в питоне нет.
- Каждый **вызов функции** создает **новую локальную область** видимости
  - рекурсия - основывается на множественности (и разных!) областях видимости.
- **Операция присваивания создает локальные имена, если они не были объ-**

явлены глобальными или нелокальными
    - иначе используем global, nonlocal
- Все остальные имена являются локальными в области видимости объ- емлющей функции, глобальными или встроенными.
    - Не было присвоения? Ищи в каком пространстве имен эта переменная!
        - объемлющей локальной области видимости (внутри объемлющей инструкции def);
        - глобальной (в пространстве имен модуля);
        - встроенной (предопределенные имена в модуле buildins)

Присвоил? Определил область локальности.

Операции непосредственого изменения объекта - это НЕ присвоение! Они не делают переменную локальной.

```
x.append(7)      # остается глобальной
y = [1, 2, 3]    # y стала локальной в текущем пространстве
```

Разрешение имен: правило LEGB

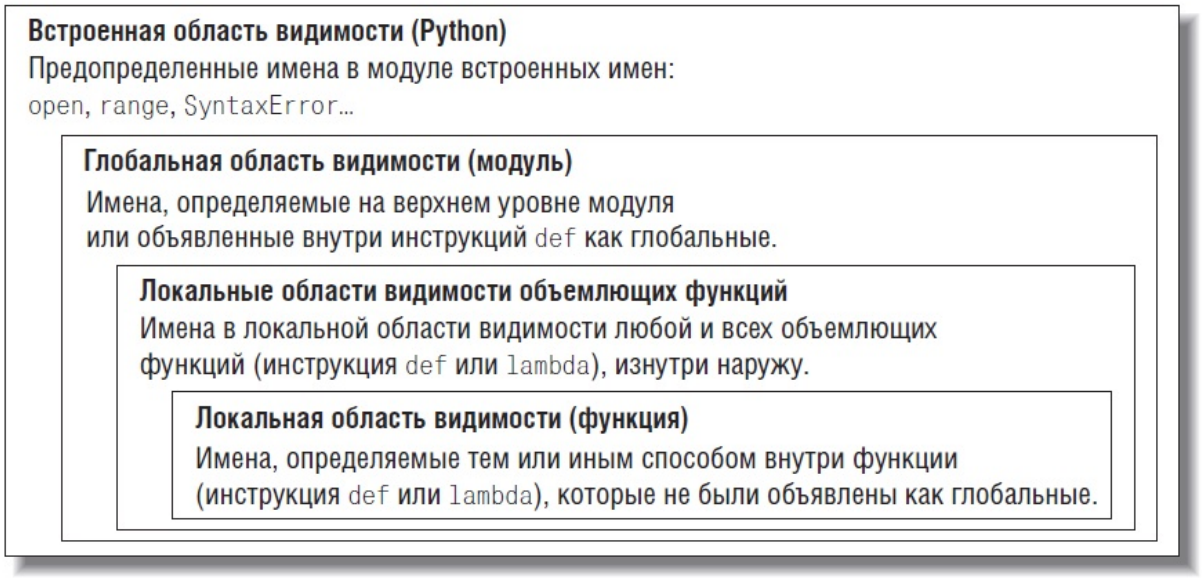
Для инструкции def:

- Поиск имен ведется самое большее в четырех областях видимости: локальной, затем в объемлющей функции (если таковая имеется), затем в глобальной и, наконец, во встроенной.
- По умолчанию операция присваивания создает локальные имена.
- Объявления global и nonlocal отображают имена на область видимости вмещающего модуля и функции соответственно.

Правило LEGB:

- Когда внутри функции выполняется обращение к неизвестному имени, интерпретатор пытается отыскать его в четырех областях видимости – в локальной (local, L), затем в локальной области любой объемлющей инструк- ции def (enclosing, E) или в выражении lambda, затем в глобальной (global, G) и, наконец, во встроенной (built-in, B).
    - Поиск завершается, как только будет найдено первое подходящее имя.
    - Если требуемое имя не будет найдено, интерпретатор выведет сообщение об ошибке.

- Когда внутри функции выполняется операция присваивания (а не обращение к имени внутри выражения), интерпретатор всегда создает или изменяет имя в локальной области видимости, если в этой функции оно не было объявлено глобальным или нелокальным.
- Когда выполняется присваивание имени за пределами функции (то есть на уровне модуля или в интерактивной оболочке), локальная область видимости совпадает с глобальной – с пространством имен модуля.



Это правила поиска имен переменных. Для атрибутов объектов применяются другие правила (см. Наследование).

```
# Глобальная область видимости
X = 99          # X и func определены в модуле: глобальная область
def func(Y):    # Y и Z определены в функции: локальная область
    # Локальная область видимости
    Z = X + Y   # X – глобальная переменная
    return Z

func(1)         # func в модуле: вернет число 100
```

- Глобальные имена: X, func (так как объявлены на верхнем уровне модуля)
- Локальные имена: Y (аргументы передаются через присвоение), Z (создается через =)

```
x = 88          # глобальная переменная x

def func():
    x = 1       # создали локальную переменную x, переопределяет глобальную

func()
print(x)        # 88, печатаем глобальную переменную
```

Встроенная область видимости (biuldins)

В действительности, встроенная область видмости – это всего лишь встроенный модуль с именем builtis, но для того, чтобы использовать имя builtis, необходимо импортировать модуль builtis, потому что это имя само по себе не является встроенным.

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis',
...множество других имен опущено...
'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set',
'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple',
'type', 'vars', 'zip']
```

Использование встроенных функций:

```
>>> zip             # Обычный способ
<class zip>
>>> import builtins # Более сложный способ
>>> builtins.zip
<class zip>
```

НЕ переопределяйте встроенные имена!

```
def hider():
    open = 'spam'     # Локальная переменная, переопределяет встроенное имя
    ...
    open('data.txt')  # В этой области видимости файл не будет открыт!
```

global и nonlocal

global и nonlocal НЕ объявляет переменную, а определяет пространство имен переменной. Позволяет изменять переменные за пределами текущей def.

Глобальные имена – это имена, которые определены на верхнем уровне вмещающего модуля.

- читать такое имя можно и без global;
- global нужно, чтобы присваивать глобальным именам.

Нелокальные имена - внешние для текущей def, но не верхний уровень модуля.

```
X = 88          # Глобальная переменная X
def func():
    global X
    X = 99      # Глобальная переменная X: за пределами инструкции def
func()
print(X)        # Выведет 99
```

Здесь все переменные тоже глобальные:

```
y, z = 1, 2         # Глобальные переменные в модуле
def all_global():
    global x        # Объявляется глобальной для присваивания
    x = y + z       # Объявлять y, z не требуется: применяется правило LEGB
```

Если x не существовала в момент вызова функции, то операция = создаст переменную x в области видимости модуля

Меньше глобальных переменных

Почему нужно стремиться делать меньше глобальных переменных?

Пусть мы хотим модифицировать этот модуль или использовать в другой программе. Чему равно x? Это зависит от того, какие функции вызывались. Сложнее понять код, ибо нужно знать как выполняется ВСЯ программа (кого вызывали последней - func1 или func2, или их вообще не вызывали).

```
X = 99

def func1():
    global X
    X = 88

def func2():
    global X
    X = 77
```

В многопоточном программировании глобальные переменные используют в качестве общей памяти для разных потоков (т.е. средства связи между потоками).

Не изменяйте глобальные переменные непосредственно в других модулях

Пусть сначала написали модуль first.py, потом написали модуль second.py и далее пытаемся понять, как работает программа. Изменение переменных модуля в другом модуле напрямую ухудщает читаемость кода.

```
# first.py
X = 99          # в first.py не знаем о second.py
# second.py
import first
print(first.X)  # читать - хорошо
first.X = 88    # изменять - плохо
```

Программист, поддерживающий first.py пытается найти кто его импортировал (и в какой директории) и изменил.

Меньше связей между модулями - проще поддерживать и изменять модули.

Лучше делать связи через вызовы функций и возвращаемые значения (проще контролировать).

```
# first.py
X = 99
def setX(new):      # интерфейс модуля
    global X
    X = new

# second.py
import first
first.setX(88)
```

Вложенные области видимости

Чтение переменной внутри функции. Ищем имя:

- в локальной области видимости функции;
- в локальных областях видимости объемлющих функций **изнутри наружу**;
- в глобальной области видимости модуля;
- в builtins (встроенная область видимости).

`x = value` внутри функции:

- создает или изменяет имя x в текущей локальной области видимости функции;
- если был `unlocal x`, то = создает или изменяет имя в *ближайшей* области видимости объемлющей функции.
- если был `global x`, то = создает или изменяет имя в области видимости объемлющего модуля.

```
X = 99              # Имя в глобальной области видимости: не используется
def f1():
    X = 88          # Локальное имя в объемлющей функции
    def f2():
        print(X)    # Обращение к переменной во вложенной функции
    f2()
f1()                # Выведет 88: локальная переменная в объемлющей функции
```

Ищем в объемлющих областях, даже если объемлющая функция фактически уже вернула управление:

```
def f1():
    X = 88
    def f2():
        print(X)    # Сохраняет значение X в объемлющей области видимости
    return f2       # Возвращает f2, но не вызывает ее

action = f1()       # Создает и возвращает функцию
action()            # Вызов этой функции: выведет 88
```

Функция f2 помнит переменную X в области видимости объемлющей функции f1, которая уже неактивна.

Замыкание (фабричная функция)

Объект функции, который сохраняет свое значение в объемлющей области видимости, даже когда эти области перестали существовать.

(Для сохранения состояний лучше использовать классы).

Например, фабричные функции иногда используются в программах, когда необходимо создавать обработчики событий прямо в процессе выполнения, в соответствии со сложившимися условиями (например, когда желательно запретить пользователю вводить данные).

Определим внешнюю фукнцию, которая создает и возвращает внутреннюю функцию, *не вызывая ее*.

```
>>> def maker(N):
...     def action(X):      # Создать и вернуть функцию
...         return X ** N   # Функция action запоминает значение N в объемлющей
...     return action       # области видимости
...
```

Вызовем внешнюю функцию. Она возвращает ссылку на созданную ею вложенную функцию, созданную при выполнении вложенной инструкции def:

```
>>> f = maker(2)            # Запишет 2 в N
>>> f
<function action at 0x014720B0>
```

Вызовем по этой ссылке функцию (внутреннюю!):

```
>>> f(3)                    # Запишет 3 в X, в N по-прежнему хранится число 2
9
>>> f(4)                    # 4 ** 2
16
```

Вложенная функция продолжает хранить число 2, значение переменной N в функции maker даже при том, что к моменту вызова функции action функция maker уже завершила свою работу и вернула управление. В действительности имя N из объемлющей локальной области видимости сохраняется как информация о состоянии, присоединенная к функции action, и мы получаем обратно значение аргумента, возведенное в квадрат. Теперь, если снова вызвать внешнюю функцию, мы получим новую вложенную функцию уже с другой информацией о состоянии, присоединенной к ней, - в результате вместо квадрата будет вычисляться куб аргумента, но ранее со храненная функция по-прежнему будет возвращать квадрат аргумента:

```
>>> g = maker(3)    # Функция g хранит число 3, а f – число 2
>>> g(3)            # 3 ** 3
27
>>> f(3)            # 3 ** 2
9
```

при каждом обращении к фабричной функции, как в данном примере, произведенные ею функции сохраняют свой собственный блок данных с информацией о состоянии. В нашем случае благодаря тому, что каждая из функций получает свой собственный блок данных с информацией о состоянии, функция, которая присваивается имени g, запо минает число 3 в переменной N функции maker, а функция f – число 2.

Где используется?

- lambda
- декораторы

Лучше для хранения информации подходят классы или глобальные переменные.

Избавьтесь от вложенности

Плоское лучше вложенного - один из принципов питона.

```
def f1():
    X = 88
    def f2():
        print(X)    # Сохраняет значение X в объемлющей области видимости
    return f2       # Возвращает f2, но не вызывает ее

action = f1()       # Создает и возвращает функцию
action()            # Вызов этой функции: выведет 88
```

Напишите лучше без вложенных функций (делает то же самое):

```
>>> def f1():
...     x = 88  # Передача значения x вместо вложения функций
...     f2(x)   # Опережающие ссылки считаются допустимыми
...
>>> def f2(x):
...     print(x)
...
>>> f1()
88
```

Вложенные области видимости и lambda-выражения

lambda (как и def) порождает новую область видимости.

```
def func():
    x = 4
    action = (lambda n: x ** n) # запоминается x из объемлющей инструкции def
    return action

x = func()
print(x(2))                     # Выведет 16, 4 ** 2
```

То же самое, с ручной передачаей значения (работет и в старых версиях питона):

```
def func():
    x = 4
    action = (lambda n, x=x: x ** n) # Передача x вручную
    return action
```

Когда нужно передать значение в lambda вручную? Если мы используем lambda в цикле. Хотим, чтобы каждая lambda запомнила свое значение i. Получилось, что все lambda запомнили **последнее** значение i (4).

```
>>> def makeActions():
...     acts = []
...     for i in range(5): # Сохранить каждое значение i
...         acts.append(lambda x: i ** x) # Все запомнят последнее значение i!
...     return acts
...
>>> acts = makeActions()
>>> acts[0]
<function <lambda> at 0x012B16B0>
>>> acts[0](2) # Все возвращают 4 ** 2, последнее значение i
16
>>> acts[2](2) # Здесь должно быть 2 ** 2
16
>>> acts[4](2) # Здесь должно быть 4  ** 2
16
```

Поиск переменной в объемлющей области видимости производится позднее, при **вызове вложенных функций**, в результате все они получат одно и то же значение (значение, которое имела переменная цикла на последней итерации). То есть каждая функция в списке будет возвращать 4 во второй степени, потому что во всех них переменная i имеет одно и то же значение.

Надо явно сохранять значение из объемлющей области видимости в виде аргумента со значением по умолчанию вместо использования ссылки на переменную из объемлющей области видимости. Значения по умолчанию вычисляются в момент создания вложенной функции (а не когда она вызывается), поэтому каждая из них сохранит свое собственное значение i:

```
>>> def makeActions():
...     acts = []
...     for i in range(5): # Использовать значения по умолчанию
...         acts.append(lambda x, i=i: i ** x) # Сохранить текущее значение i
...     return acts
...
>>> acts = makeActions()
>>> acts[0](2) # 0 ** 2
0
>>> acts[2](2) # 2 ** 2
4
>>> acts[4](2) # 4 ** 2
16
```

На подобный эффект можно натолкнуться в коде, которые генерируют функции-обработчики событий для GUI.

nonlocal - "пропустить локальную область видимости при поиске имен"

Для чтения значений переменных не нужно. Только для =.

- **global**
  - вынуждает интерпретатор начинать поиск имен с области объемлю- щего модуля;
  - позволяет присваивать переменным новые значения.
  - Область поиска простирается вплоть до встроенной области видимости, если искомое имя не будет найдено в модуле,
  - при этом операция присваивания значений глобальным именам всегда будет создавать или изменять переменные в области видимости модуля.
- **nonlocal**
  - ограничивает область поиска областями видимости объемлющих функций;
  - требует, чтобы перечисленные в инструкции имена **уже существовали**,
  - позволяет присваивать им новые значения;
  - в область поиска **не входят глобальная и встроенная области видимости**.

```
>>> def tester(start):
...     state = start           # Обращение к нелокальным переменным действует как обы
чно
...     def nested(label):
...         print(label, state) # Извлекает значение state из области видимости объемл
ющей функции
...         state += 1          # создается ЛОКАЛЬНАЯ переменная - хотели изменить нел
окальную
...     return nested
...
>>> F = tester(0)
>>> F('spam')
spam 0
>>> F('ham')
ham 0
```

По умолчанию нельзя изменять значения переменной объемлющей области видимости

unlocal - позволяет изменять, даже если функция tester завершила работу к моменту вызова функции nested через переменню F:

```
>>> def tester(start):
...     state = start               # В каждом вызове сохраняется свое значение state
...     def nested(label):
...         nonlocal state          # Объект state находится
...         print(label, state)     # в объемлющей области видимости
...         state += 1              # Изменит значение переменной, объявленной как non
local
...     return nested
...
>>> F = tester(0)
>>> F('spam') # Будет увеличивать значение state при каждом вызове
spam 0
>>> F('ham')
ham 1
>>> F('eggs')
eggs 2
```

Напоминаем, каждый вызов фабричной фукнции tester будет создавать отдельную копию переменной state. Объект state, находящийся в объемлющей области видимости, фактически прикрепляется к возвращаемому объекту функции nested - каждый вызов функции tester создает новый, независимый объект state, благодаря чему изменение state в одной функции не будет оказывать влияния на другие. Вызываем еще эти же функции:

```
>>> G = tester(42)  # Создаст новую функцию, которая начнет счет с 42
>>> G('spam')
spam 42
>>> G('eggs')       # Обновит значение state до 43
eggs 43
>>> F('bacon')      # Но в функции F значение state останется прежним
bacon 3             # Каждая новая функция получает свой экземпляр state
```

unlocal переменная должна уже существовать (global - не обязательно, создаем новую)

```
>>> def tester(start):
...     def nested(label):
...         nonlocal state  # Нелокальные переменные должны существовать!
...         state = 0
...         print(label, state)
...     return nested
...
SyntaxError: no binding for nonlocal 'state' found
>>> def tester(start):
...     def nested(label):
...         global state    # Глобальные переменные могут отсутствовать
...         state = 0       # Создаст переменную в области видимости модуля
...         print(label, state)
...     return nested
...
>>> F = tester(0)
>>> F(‘abc’)
abc 0
>>> state
0
```

unlocal область видимости - без глобальной области модуля или built-in

```
>>> spam = 99
>>> def tester():
...     def nested():
...         nonlocal spam   # Переменная должна быть внутри def, а не в модуле!
...         print('Current=', spam)
...         spam += 1
...     return nested
...
SyntaxError: no binding for nonlocal 'spam' found
```

Интерпретатор определяет местоположение нелокальных имен в момент создания функции, а не в момент ее вызова.

Чем заменить unlocal?

Глобальная переменная

Минус - *один* (единственный) экземпляр переменной для хранения информации.

global нужно написать в обеих функциях.

```
>>> def tester(start):
...     global state        # Переместить в область видимости модуля
...     state = start       # global позволяет изменять переменные, находящиеся
...     def nested(label):  # в области видимости модуля
...         global state
...         print(label, state)
...         state += 1
...     return nested
...
>>> F = tester(0)
>>> F('spam')               # Каждый вызов будет изменять глобальную state
spam 0
>>> F('eggs')
eggs 1
>>> G = tester(42)          # Сбросит значение единственной копии state
>>> G('toast')              # в глобальной области видимости
toast 42
>>> G('bacon')
bacon 43
>>> F('ham')                # Ой – значение моего счетчика было затерто!
ham 44
```

Классы

Код лучше читается!

```
>>> class tester:                   # Альтернативное решение на основе классов
...     def __init__(self, start):  # Конструктор объекта,
...         self.state = start      # сохранение информации в новом объекте
...     def nested(self, label):
...         print(label, self.state) # Явное обращение к информации
...         self.state += 1          # Изменения всегда допустимы
...
>>> F = tester(0)           # Создаст экземпляр класса, вызовет __init__
>>> F.nested('spam')        # Ссылка на F будет передана в аргументе self
spam 0
>>> F.nested('ham')
ham 1
>>> G = tester(42)          # Каждый экземпляр получает свою копию информации
>>> G.nested('toast')       # Изменения в одном объекте не сказываются на других
toast 42
>>> G.nested('bacon')
bacon 43
>>> F.nested('eggs')        # В объекте F сохранилась прежняя информация
eggs 2
>>> F.state                 # Информация может быть получена за пределами класса
3
```

Избавимся от функции nested, чтобы мы могли писать прямо F('spam'), переопределив функцию __call__

```
>>> class tester:
...     def __init__(self, start):
...         self.state = start
...     def __call__(self, label):      # Вызывается при вызове экземпляра
...         print(label, self.state)    # Благодаря этому отпадает
...         self.state += 1             # необходимость в методе .nested()
...
>>> H = tester(99)
>>> H('juice')                          # Вызовет метод __call__
juice 99
>>> H('pancakes')
pancakes 100
```

Атрибуты функций

```
>>> def tester(start):
...     def nested(label):
...         print(label, nested.state)  # nested – объемлющая область видимости
...         nested.state += 1           # Изменит атрибут, а не значение имени nested
...         nested.state = start        # Инициализация после создания функции
...     return nested
...
>>> F = tester(0)
>>> F('spam')           # F – это функция 'nested'
spam 0                  # с присоединенным атрибутом state
>>> F('ham')
ham 1
>>> F.state             # Атрибут state доступен за пределами функции
2
>>>
>>> G = tester(42)      # G имеет собственный атрибут state,
>>> G('eggs')           # отличный от одноименного атрибута функции F
eggs 42
>>> F('ham')
ham 2
```

имя функции nested является локальной переменной в области видимости функции tester, включающей имя nested, – на это имя можно ссылаться и внутри функции nested. Кроме того, здесь используется то обстоятельство, что изменение самого объекта не является операцией присваивания, - операция увеличения значения nested.state изменяет часть объекта, на который ссылается имя nested, а не саму переменную с именем nested. Поскольку во вложенной функции не выполняется операция присваивания, необходимость в инструкции nonlocal отпадает сама собой.

Контрольные вопросы

1. Что выведет следующий фрагмент и почему?

```
>>> X = 'Spam'
>>> def func():
...     print(X)
...
>>> func()
```

2. Что выведет следующий фрагмент и почему?

```
>>> X = 'Spam'
>>> def func():
...     X ='NI!'
...
>>> func()
>>> print(X)
```

3. Что выведет следующий фрагмент и почему?

```
>>> X = 'Spam'
>>> def func():
...     X = 'NI'
...     print(X)
...
>>> func()
>>> print(X)
```

4. Что выведет следующий фрагмент и почему?

```
>>> X = 'Spam'
>>> def func():
...     global X
...     X = 'NI'
...
>>> func()
>>> print(X)
```

5. Что выведет следующий фрагмент и почему?

```
>>> X = 'Spam'
>>> def func():
...     X = 'NI'
...     def nested():
...         print(X)
...     nested()
...
>>> func()
>>> X
```

6. Что выведет следующий фрагмент и почему?

```
>>> def func():
...     X = 'NI'
...     def nested():
...         nonlocal X
...         X = 'Spam'
...     nested()
...     print(X)
...
>>> func()
```

7. Назовите три или более способов в языке Python сохранять информацию о состоянии в функциях.

Ответы

1. В данном случае будет выведена строка 'Spam', потому что функция обращается к глобальной переменной в объемлющем модуле (если внутри функции переменной не присваивается значение, она интерпретируется как глобальная).
2. В данном случае снова будет выведена строка 'Spam', потому что операция присваивания внутри функции создает локальную переменную и тем самым скрывает глобальную переменную с тем же именем. Инструкция print находит неизмененную переменную в глобальной области видимости.
3. Будет выведена последовательность символов 'Ni' в одной строке и 'Spam' - в другой, потому что внутри функции инструкция print найдет локальную переменную, а за ее пределами – глобальную.
4. На этот раз будет выведена строка 'Ni', потому что объявление global предписывает выполнять присваивание внутри функции переменной, находящейся в глобальной области видимости объемлющего модуля.
5. В этом случае снова будет выведена последовательность символов 'Ni' в одной строке и 'Spam' – в другой, потому что инструкция print во вложенной функции отыщет имя в локальной области видимости объемлющей функции, а инструкция print в конце фрагмента отыщет имя в глобальной области видимости.
6. Этот фрагмент выведет строку 'Spam', так как инструкция nonlocal (доступная в Python 3.0, но не в 2.6) означает, что операция присваивания внутри вложенной функции изменит переменную X в локальной области видимости объемлющей функции. Без этой инструкции операция присваивания классифицировала бы переменную X как локальную для вложенной функции и создала бы совершенно другую переменную - в этом случае приведенный фрагмент вывел бы строку 'NI'.
7. Поскольку значения локальных переменных исчезают, когда функция возвращает управление, то информацию о состоянии в языке Python можно сохранять в глобальных переменных, для вложенных функций - в области видимости

объемлющих функций, а также посредством аргументов со значениями по умолчанию. Иногда можно использовать прием, основанный на сохранении информации в атрибутах, присоединяемых к функциям, вместо использования области видимости объемлющей функции. Альтернативный способ заключается в использовании классов и приемов ООП, который обеспечивает лучшую поддержку возможности сохранения ин- формации о состоянии, чем любой из предыдущих приемов, основанных на использовании областей видимости, потому что этот способ делает сохранение явным, позволяя выполнять присваивание значений атрибутам.

Условные ветвления

if elif else

Синтаксис оператора ветвления:

```
if условие1:     # обязательная часть
    операторы1   # какие операторы относятся к if определяет отступ
elif условие2:   # не обязательная часть
    операторы2
elif условие3:   # не обязательная часть
    операторы3
...              # частей elif может быть 0 или больше
else:            # не обязательная часть
    операторы
```

Никаких () или {}. Составной (блочный) оператор определяется отступами.

Не забывайте : в конце условия и после else.

pass - ничего не делать.

Пример: посчитать модуль числа (без abs)

```
if x < 0:
    x = -x
    print('Change sign')
print('Absolute value is', x)
```

Пример: четное или нечетное

```
if x%2 == 0:
    print('even')
else:
    print('odd')
```

Пример: положительное, отрицательное, ноль

```
if x == 0:
    print('zero')
elif x < 0:
    print('negative')
else:
    print('positive')
```

if else в одну строку кода

Можно написать в одну строку

```
оператор1 if условие1 else оператор2
```

Если выражение *условие1* возвращает True, то результат всего выражения *оператор1*. Если False, то *оператор2*.

Допустим, в Windows переменная offset должна быть 10, а во всех других случаях 20.

```
offset = 20              # значение по умолчанию
if sys.platform.startswith("win"):
    offset = 10
```

или в одну строку:

```
offset = 10 if sys.platform.startswith("win") else 20
```

Не нужно писать : или ()

if else в выражениях

Пусть ширина 100, а если есть margine, то нужно прибавить еще 10.

```
width = 100 + 10 if margin else 0 # ОШИБКА!
```

Если margin *True*, то в width правильное значение 100+10. Но если *False*, в width будет записано число 0 (не то, что мы хотели). Нужно больше скобок! Код стал правильный и лучше читается.

```
width = 100 + (10 if margin else 0) # Теперь правильно
```

Пример: no files, 1 file, 2 files etc

```
print("{0} file{1}".format((count if count != 0 else "no"),
                           ("s" if count != 1 else "")))
```

Операторы сравнения

Присвоить используется чаще, поэтому присвоить =, а сравнить на равенство ==

Оператор	Значение
x < y	x меньше y
x <= y	x меньше или равно y
x > y	x больше y
x >= y	x больше или равно y
x != y	x НЕ равен y
x == y	x равен y
x is y	x и y ссылаются на один и тот же объект
x in y	x в коллекции y

```
>>> a = ["xyz", 3, None]  # один список
>>> b = ["xyz", 3, None]  # другой список с таким же содержимым
>>> a == b                # у списков одинаковое содержимое
True
>>> a is b                # но это разные списки
False
>>> b = a
>>> a is b
True
```

Питон любит математиков:

```
>>> x = 12
>>> 10 < x <= 15
True
>>> x = -3
>>> -10 < x < -2
True
```

Логические операторы

Оператор	Значение
not	логическое НЕ
and	логическое И
or	логическое ИЛИ

Для изменения приоритета выражений используйте скобки ().

Циклы

Существуют два оператора цикла: **while** и **for .. in**. Оба варианта могут быть с **else** частью.

while

```
while условие_продолжения_цикла:
    тело_цикла
```

или

```
while условие_продолжения_цикла:
    тело_цикла
else:
    выполняется_если_не_было_break
```

Тело цикла пишем с отступом

Найдем сумму чисел от 1 до n=10 (включительно) с помощью цикла while:

```
n = 10
i = 1
sum = 0

while i <= n:
    sum += i

print(sum)
```

Бесконечный цикл:

```
while True:
    print('hi')
```

for .. in - перебрать по одному элементы последовательности

Можно перебирать по 1 элементу последовательность, применяя for .. in. Переберем по символу строку 'Hello':

```
for x in 'Hello':
    print(x)
```

Напечатает:

```
H
e
l
l
o
```

Пример: индекс последнего вхождение символа в строке или -1

Найдем номер последнего вхождения буквы k = 'l' в строку 'Hello':

```
i = 0
ind = -1
for x in 'Hello':
    if x == k:
        ind = i
print(ind)
```

При k = 'l' напечатает 3 (нумерация с 0), а при k = 'z' напечатает -1. Получим номер последнего вхождения буквы в строку или -1, если буквы в строке нет.

continue - пропускаем итерацию цикла

Будем при поиске пропускать каждую вторую букву, т.е. буквы 'e' и 'l'.

continue - перейти к следующей итерации цикла.

```
i = 0
ind = -1
for x in 'Hello':
    if i % 2 != 0:  # пропускаем каждый второй символ
        continue
    if x == k:
        ind = i
print(ind)
```

break - прерываем цикл

Найдем индекс первого вхождения символа в строку. Как только его нашли, дальше цикл выполнять не надо. Прервем цикл с помощью оператора **break**.

```
i = 0
ind = -1
for x in 'Hello':
    if x == k:
        ind = i
        break

# сюда передаст управление оператор break
print(ind)
```

for .. else

Можно использовать расширенную форму оператора for:

```
for переменная in последовательность:
    операторы цикла
else:
    выполняется, если в цикле не выполнялось break
```

Напечатаем YES, если символ из переменной k в строке есть, и NO, если его нет:

```
for x in 'Hello':
    if x == k:
        print('YES')    # печатаем YES и выходим из цикла
        break
else:
    print('NO')         # если break не было, печатаем NO
```

Оператор while тоже имеет форму while .. else

range(от, до, шаг) - последовательность чисел

Напечатаем числа от 0 (включительно) до 10 (не включая 10). Последовательность чисел [0, 10) сделаем с помощью range.

```
for x in range(10):
    print(x)
```

- **range(10)** - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- **range(3, 10)** - 3, 4, 5, 6, 7, 8, 9
- **range(3, 10, 2)** - 3, 5, 7, 9

enumerate - индексы и значения

Можно перебирать не только по значениям, но получать на каждом шаге и индекс, и значение.

enumerate - дает последовательность пар индекс, значение

```
for i, x in enumerate('Hello'):
    print(i, x)
```

напечатает:

```
0 H
1 e
2 l
3 l
4 o
```

Примеры кода

Ввод с клавиатуры

Функция **input()** читает 1 строку с клавиатуры.

```
x = input()
```

Чтобы прочитать с клавиатуры 2 строки, нужно 2 раза вызвать input()

```
x = input()   # первую строку прочитали и записали в переменную x
y = input()   # вторую строку прочитали и записали в переменную y
```

Напишем программу, которая складывает 2 целых числа.

```
x = input()   # 3
y = input()   # 5
print(x+y)    # 35 ???
```

Почему 35?

```
x = input()        # 3
y = input()        # 5

print(x, type(x))  # 3 string
print(y, type(y))  # 5 string

z = x+y
print(z, type(z))  # 35 string (две строки написали рядом - конкатенация, concatenatio
n)
```

Потому что input() вернула строку. У нас есть строка "3" и строка "5", а не числа 3 и 5. Строки оператором + соединяются вместе в строку "35". Строки пишутся одна за другой.

Как исправить?

Мы знаем, что числа будут целые. Поэтому сразу изменим тип данных на int.

```
x = input()        # 3
y = input()        # 5

x = int(x)
y = int(y)

print(x, type(x))  # 3 int
print(y, type(y))  # 5 int

z = x+y
print(z, type(z))  # 8 int (работаю правила сложения целых чисел)
Можно написать короче. Сразу делаем прочитанные данные int

x = int(input())   # 3, прочитали строку, сделали из строки int
y = int(input()    # 5, прочитали строку, сделали из строки int

print(x, type(x))  # 3 int
print(y, type(y))  # 5 int

z = x+y
print(z, type(z))  # 8 int (работаю правила сложения целых чисел)
```

Два числа на одной строке

Введем числа не по 1 на строке, а на одной строке через пробел 3 5

```
x, y = map(int, input().split())   # 3 5
print(x+y)                         # 8
```

Строку, которую вернул input(), разбили по пробелам на "слова". К каждому "слову" с помощью функции map применили функцию int(). Результат записали в переменные x и y.

Задачи

1. Когда закончится K-тый урок

Уроки начинаются в 8:00. Урок длится 45 минут и 5 минут перемена. Во сколько закончится k-тый урок (k<15). Результат вывести в формате hh:mm

Чтобы напечатать часы h и минуты m с ведущими нулями пишем в старом формате:

```
print('%02d:%02d' %(h, m))
```

в новом формате:

```
print('{:02}:{:02}'.format(h, m))
```

2. Длина отрезка

Для отрезка на плоскости XY напишите функцию length(x1, y1, x2, y2), которая вычисляет расстояние между точками (x1, y1) и (x2, y2).

Программе на вход подаются координаты x1 y1 x2 y2 (на одной строке через пробел). Программа печатает расстояние между указанными точками.

3. Площадь треугольника (формула Герона)

На плоскости XY даны координаты вершин треугольника в формате
x1 y1 x2 y2 x3 y3
(на одной строке через пробел)
Найдите (и напечатайте) площадь этого прямоугольника по формуле Герона

$$s = \sqrt{p \cdot (p - a) \cdot (p - b) \cdot (p - c)}$$

.где a, b, c - длины сторон треугольника, а p = (a+b+c)/2 - его полупериметр.

Для этого напишите функцию s(x1, y1, x2, y2, x3, y3).

4. Часы

Чтобы прочитать часы и минуты в переменные h и m в формате hh:mm пишем

```
h, m = map(int, input().split(':'))   # делаем split по разделителю ':'
```

Напишите функции и решите задачи для вывода времени на электронные часы, которые показывают время в формате hh:mm.

4.1 time2min(h, m)

Реализуйте функцию time2min(h, m), которая переводит часы и минуты в минуты с начала суток (00:00).

Проверьте эту функцию.

4.2 min2time(mm)

Реализуйте функцию min2time(mm), которая минуты с начала суток переводит в часы и минуты (для показа на электронных часах).

4.3 Время прибытия электрички

Электричка отправляется в h1:m1 и едет h2:m2. Выведите время прибытия электрички на электронных часах в формате hh:mm.

Формат входных данных: на одной строке h1:m1, на другой h2:m2

4.4 Время в пути

Электричка отправляется в h1:m1 и прибывает в h2:m2. Выведите время в пути электрички в формате hh:mm.

5 Делится на 3 или 5, но не на 15

Дано натуральное число. Напечатайте YES, если число делится на 3 или 5, но не делится на 15. В противном случае напечатайте NO.

6 Високосный год

Дан год (натуральное число). Напечатайте YES, если год високосный. Иначе напечатайте NO.

Год високосный, если он делится на 4, но не делится на 100. Если год делится на 400, то он тоже считается високосным.

7 Наибольший общий делитель (НОД)

Даны 2 числа. Найдите их НОД по алгоритму Евклида. Найдем НОД для чисел 125 и 21.

125 % 21 = 18
21 % 18 = 3
18 % 3 = 1
НОД(125, 21) = 3

8 Наименьшее положительное

Даны целые числа. Напечатать наименьшее положительное из них. Если такого числа нет, то ничего не печатать.

Пример чтения нескольких чисел и их печати.

```
a = map(int, input().split())
for x in a:
    print(x)
```

Входные данные:

2 -7 66 1 0 -3

Выходные данные (печатаем все числа):

2
-7
66
1
0
-3

8.1 Наименьшее положительное (нет такого)

Если положительных чисел нет, печатать Nothing

9 Номера всех положительных чисел в последовательности

Дана последовательность целых чисел на 1 строке через пробел.

Напечатайте индексы всех положительных чисел в этой последовательности.

Если положительных чисел нет, ничего печатать не надо.

10 слияние

На двух строках даны *отсортированные* (неубывающие) последовательности целых чисел.

Напечатайте по неубыванию все числа.

При длине последовательностей n и m, сложность решения должна быть не более, чем O(n+m)

10 (Дополнительная) Ромашка

Математически угорелая девушка гадает на ромашках о любви. Она срывает ромашку, считает лепестки, вычисляет факториал (fn=fn-1* n, где f1=1) от количества лепестков. Затем подсчитывает сумму цифр полученного числа (fn).

Если сумма - простое число, значит ЛЮБИТ, если составное - НЕ ЛЮБИТ.

Девушке попалось поле с ромашками, у которых встречалось все количество лепестков от 1 до (1 ≤ N ≤ 1000) по одному разу.

Написать программу, которая по максимальному числу лепестков в ромашке вычисляет сколько раз встречается результат "ЛЮБИТ"

Input format: Целое число (1 ≤ N ≤ 1000) - максимальное число лепестков.

Output format: Целое число (сколько раз встречается результат "ЛЮБИТ")

Примеры:

Вход	Выход
1	0
3	1

Автор: Овсянникова Т.В.

Методические указания

- Формула Герона - обязательно именно такой формат функции и в функции использовать функцию length, которую написали в предыдущей задаче.
- Часы - использовать функции перевода в минуты и обратно
- Деление и високосный год - попробовать реализовать еще и в виде функции (без использования and и or).
- Наименьшее положительное - подумать о None и for .. else

Итераторы

Источники:

- Intermediate Python
- Лутц, главы 14, 20, 29. **(TODO: в следующий проход интегрировать сюда главы 20 и 29, кроме того, сейчас тут немного дублируется информация при переходе от итераторов к генераторам)**.

Для начала нам стоит познакомиться с итераторами. Как подсказывает Wiki, итератор — это интерфейс, предоставляющий доступ к элементам коллекции (массива или контейнера). Здесь важно отметить, что **итератор только предоставляет доступ, но не выполняет итерацию по ним**. Это может звучать довольно запутано, так что остановимся чуть подробнее. Тему итераторов можно разбить на три части:

- Итерируемый объект
- Итератор
- Итерация

Итерируемым объектом в Python называется любой объект, имеющий методы __iter__ или __getitem__, которые возвращают итераторы или могут принимать индексы. В итоге итерируемый объект это объект, который может предоставить нам итератор.

Итератором в Python называется объект, который имеет метод next (Python 2) или __next__. Вот и все. Это итератор.

Итерация - это процесс получения элементов из какого-нибудь источника, например списка. Итерация - это процесс перебора элементов объекта в цикле.

Теперь, когда у нас есть общее понимание основных принципов, перейдём к генераторам.

Генераторы

Генераторы это итераторы, по которым можно итерировать только один раз. Так происходит поскольку они не хранят все свои значения в памяти, а генерируют элементы "на лету". Генераторы можно использовать с циклом for или любой другой функцией или конструкцией, которые позволяют итерировать по объекту. В большинстве случаев генераторы создаются как функции. Тем не менее, они не возвращают значение также как функции (т.е. через **return**), в генераторах для этого используется ключевое слово **yield**.

Пример функции-генератора:

```
def generator_function():
    for i in range(10):
        yield i

for item in generator_function():
    print(item)

# Вывод: 0
# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
```

Чуть более полезный пример вычисления чисел Фибоначчи:

```
# generator version
def fibon(n):
    a = b = 1
    for i in range(n):
        yield a
        a, b = b, a + b

# использование:
for x in fibon(1000000):
    print(x)
```

В случае списка все бы числа хранились в памяти:

```
def fibon(n):
    a = b = 1
    result = []
    for i in range(n):
        result.append(a)
        a, b = b, a + b
    return result
```

Функция next()

Встроенная функция **next()** позволяет переходить к следующему элементу коллекции.

```
>>> def generator_function():
...     for i in range(3):
...         yield i
...
>>> gen = generator_function()
>>> print(next(gen))
0
>>> print(next(gen))
1
>>> print(next(gen))
2
>>> print(next(gen))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Когда по чему можно итерировать заканчивается, функция next() порождает исключение *StopIteration*

Цикл for автоматически перехватывает это исключение и перестает вызывать next().

Функция iter()

Попробуем проитерировать по строке с помощью next.

```
>>> str = 'Hello'
>>> next(str)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not an iterator
```

Да, строка - итерируемый объект, но не итератор. Чтобы получить из строки итератор, используем встроенную функцию **iter()**. Она возвращает *итератор* из *итерируемого объекта*.

Не все является итерируемым объектом. Например, объект типа int не является итерируемым.

```
int_var = 1779
iter(int_var)
# Вывод: Traceback (most recent call last):
#          File "<stdin>", line 1, in <module>
#        TypeError: 'int' object is not iterable
# int не итерируемый объект
```

Добудем из строки итератор и пройдемся по нему функцией next.

```
>>> str = 'Hello'
>>> my_iter = iter(str)
>>> next(my_iter)
'H'
```

Функции-генераторы

Источники:

- Саммерфилд, Глава 4 Управляющие структуры и функции / Собственные функции / Лямбда-функции
- Саммерфилд, Глава 8 Усовершенствованные приемы программирования / Выражения-генераторы и функции-генераторы

Зачем нужны генераторы?

Это средство отложенных вычислений. Значения вычисляются только тогда, когда они действительно необходимы.

Удобнее, чем вычислить за один раз огромный список и потом держать его в памяти.

Некоторые генераторы могут воспроизводить столько значений, сколько потребуется, без ограничения сверху. Например, последовательность квадратов 1, 4, 9, 16 и так далее.

Термины

Функция-генератор, или метод-генератор – это функция или метод, содержащая выражение **yield**. В результате обращения к функции-генератору возвращается *итератор*. Значения из итератора извлекаются по одному, с помощью его метода **next**(). При каждом вызове метода **next**() он возвращает результат вычисления выражения yield. (Если выражение отсутствует, возвращается значение None.) Когда функция-генератор завершается или выполняет инструкцию return, возбуждается исключение *StopIteration*. На практике очень редко приходится вызывать метод **next**() или обрабатывать исключение StopIteration. Обычно функция-генератор используется в качестве итерируемого объекта.

Список vs генератор

Создает и возвращает **список**:

```
def letter_range(a, z):
    res = []
    while ord(a) < ord(z):
        res.append(a)
        a = chr(ord(a)+1)
    return res
```

Использование:

```
for c in letter_range('m', 'v'):    # одинаково для списка и генератора
    print(c)

az = letter_range('m', 'v')         # az - список
```

Создает и возвращает **генератор**, т.е. возвращает каждое значение по требованию:

```
def letter_range(a, z):
    while ord(a) < ord(z):
        yield a
        a = chr(ord(a)+1)
```

Использование:

```
for c in letter_range('m', 'v'):    # одинаково для списка и генератора
    print(c)

az = letter_range('m', 'v')         # az - генератор
az_1 = list(az)                     # список
az_2 = tuple(az)                    # кортеж
```

Выражения-генераторы

Похожи на генераторы списков, но пишем круглые скобки, а не квадратные:

```
(expression for item in iterable)
(expression for item in iterable if condition)
```

Напишем функцию, которая для словаря возвращает генератор, который выдает пары ключ-значение в порядке, в котором отсортированы ключи. Вариант 1:

```
def items_in_key_order(d):
    for key in sorted(d):
        yield (key, d[key])
```

Вариант 2: (эквивалентно)

```
def items_in_key_order(d):
    return ((key, d[key]) for key in sorted(d))
```

Вариант 3: теперь через лямбда-фукнцию:

```
items_in_key_order = lambda d: ((key, d[key]) for key in sorted(d))
```

Использование:

```
>>> d1 = {12:21, 3:3, -6:6, 100:0}
>>> d1
{12: 21, 3: 3, -6: 6, 100: 0}
>>> for k, v in items_in_key_order(d1): print(k, v) # ключи отсортированы
...
-6 6
3 3
12 21
100 0
>>> for k, v in d1.items(): print(k, v)             # несортированный порядок
...
12 21
3 3
-6 6
100 0
```

Бесконечные последовательности

Напишем *генератор последовательности* без ограничения сверху. Например, следующий элемент на 0.25 больше предыдущего.

```
def quarters(next_quarter=0.0):
    while True:
        yield next_quarter
        next_quarter += 0.25
```

Получим с ее помощью *список* элементов от 0 до 1: [0.0, 0.25, 0.5, 0.75, 1.0]

```
result = []
for x in quarters():
    result.append(x)
    if x >= 1.0:
        break
```

Хотим, чтобы последовательность пропустила "плохие числа" (0.5 и далее) и перешла сразу к "хорошим" (1). Изменим генератор.

```
def quarters(next_quarter=0.0):
    while True:
        received = (yield next_quarter)   # было  yield next_quarter
        if received is None:              # вдруг фигню подсунут, проигнорируем
            next_quarter += 0.25
        else:                             # нефигню сделаем следующим значением
            next_quarter = received
```

Выражение yield поочередно возвращает каждое значение вызывающей программе. Кроме того, если будет вызван метод send() генератора, то переданное значение будет принято функцией-генератором в качестве результата выражения yield.
Использование:

```
result = []
generator = quarters()
while len(result) < 5:
    x = next(generator)
    if abs(x - 0.5) < sys.float_info.epsilon:
        x = generator.send(1.0)
    result.append(x)

print(result)    # [0.0, 0.25, 1.0, 1.25, 1.5].
```

Здесь создается переменная, хранящая ссылку на генератор, и вызывается встроенная функция next(), которая извлекает очередной элемент из указанного ей генератора. (Того же эффекта можно было бы достичь вызовом специального метода **next**() генератора, в данном случае следующим образом: x = generator.**next**().) Если значение равно 0.5, генератору передается значение 1.0 (которое немедленно возвращается обратно).

Лутц, стр 572: В версии Python 2.5 в протокол функций-генераторов был добавлен метод send. Метод send не только выполняет переход к следующему элементу в последовательности результатов, как это делает метод next, но еще и обеспечивает для вызывающей программы способ взаимодействия с генератором, влияя на его работу. С технической точки зрения yield в настоящее время является не инструкцией, а выражением, которое возвращает элемент, передаваемый методу send (несмотря на то, что его можно использовать любым из двух способов – как yield X или как A = (yield X)). Когда выражение yield помещается справа от оператора присваивания, оно должно заключаться в круглые скобки, за исключением случая, когда оно не является составной частью более крупного выражения. Например, правильно будет написать X = yield Y, а также X = (yield Y) + 42. При использовании расширенного протокола

значения передаются генератору G вызовом метода G.send(value). После этого программный код генератора возобновляет работу, и выражение yield возвращает значение, полученное от метода send. Когда вызывается обычный метод G.**next**() (или выполняется эквивалентный вызов next(G)), выражение yield возвращает None.

Функциональное программирование

Функциональное программирование — раздел дискретной математики и парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании).wiki

Функция высшего порядка — в программировании функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата.

map() - применить функцию ко всем элементам списка

map(*function_to_apply*, *iterable*, *...) - применяет функцию _function_to_apply* ко всем элементам последовательности *iterable*. Если заданы дополнительные аргументы (последовательности), то функция *function_to_apply* должна принимать столько аргументов, сколько последовательностей переданно далее в map.

```
a = map(int, input().split())  # 3 14 27 -1
```

Далее по полученному объекту типа map можно итерироваться.

```
for x in a:
    print(x)
# 3
# 14
# 27
# -1
```

Еще о различиях map и list:

```
>>> a = map(int, input().split())
3 14 27 -1                              # ввели эти числа
>>> a                                   # map - это не список
<map object at 0xffd87170>
>>> print(a)
<map object at 0xffd87170>
>>> b = list(a)                         # но из map можно получить list
>>> b
[3, 14, 27, -1]
>>> c = list(a)
>>> c                                   # итерироваться можно только ОДИН раз!!!
[]
```

map - пример: квадраты чисел

Посчитаем квадраты чисел, заданных в списке.

Обычная функция:

```
items = [1, 2, 3, 4, 5]
squared = []
for i in items:
    squared.append(i**2)
```

map:

```
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, items))
```

map - пример: задаем числа и их степени

Возведем числа из списка в степени, которые тоже заданы списком

Обычная функция:

```
items = [10, 2, 3, 4]
n     = [ 3, 1, 2, 0]
res = []
for i, x in enumerate(items):
    res.append(x**n[i]))
```

map:

```
items = [10, 2, 3, 4]
n     = [ 3, 1, 2, 0]
res = list(map((lambda x, i: x**i), items, n))
```

map - список входных данных может быть списком функций

```
def multiply(x):
    return (x*x)
def add(x):
    return (x+x)

funcs = [multiply, add]
for i in range(5):
    value = list(map(lambda x: x(i), funcs))
    print(value)

# Вывод:
# [0, 0]
# [1, 2]
# [4, 4]
# [9, 6]
# [16, 8]
```

filter() - оставить те элементы, для которых True фильтрующая фукнция

filter(*filter_function*, *list_of_inputs*) - оставить только те элементы списка *list_of_inputs*, у которых применение функции *filter_function* вернуло *True*.

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
print(less_than_zero)

# Вывод: [-5, -4, -3, -2, -1]
```

filer похож на цикл, но он является встроенной функцией и работает быстрее.

reduce() - свертка списка с помощью функции

В Python 3 встроенной функции reduce() нет, но её можно найти в модуле *functools*.

reduce(*function_to_apply*, *list_of_inputs*, *init_value*) - сворачивает элементы списка *list_of_inputs* в один объект, применяя *function_to_apply* по очереди к последовательным парам элементов. Предполагая для первой пары *init_value* - необязательный параметр.

Найдем произведение чисел из списка. Классический вариант:

```
product = 1
list = [1, 2, 3, 4]
for num in list:
    product = product * num

# product = 24
```

С reduce:

```
from functools import reduce
product = reduce((lambda res, x: res * x), [1, 2, 3, 4])    # 24
product = reduce((lambda res, x: res * x), [1, 2, 3, 4], 1) # эквивалентно
```

Порядок вычислений:

```
(((2*3)*4)*5)*6
```

Цепочка вызовов связывается с помощью промежуточного результата (res). Если список пустой, просто используется третий параметр (в случае произведения нуля множителей это 1):

```
reduce(lambda res, x: res*x, [], 1)    # 1
```

Реверс списка (если забыли про функцию *reversed*):

```
>>> reduce(lambda res, x: [x]+res, [1, 2, 3, 4], [])
[4, 3, 2, 1]
```

sum - сумма элементов списка

all

all(iterable) - возвращает True если все элементы в iterable истины (или этот iterable пустой). То же самое, что:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

any

any(iterable) - возвращает True если хоть один элемент в iterable истин (если этот iterable пустой, возвращается False). То же самое, что:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

Задачи

Факториал-лямбда

Напишите вычисление факториала через лямбда-функцию.

Продемонстрируйте работоспособность этой функции.

Факториал-генератор

Напишите вычисление факториала через генератор.

Продемонстрируйте работоспособность этой функции.

Векторное сложение

Реализовать через функции высших порядков и через обычную функцию (с for). Как ведет себя код, если размер векторов разный?

Векторное умножение

Реализовать через функции высших порядков скалярное умножение векторов.

Все в дом

Придумайте или найдите задачу по теме занятия.

Обзор главы

В главе рассказывается о базовых вещах объектно-ориентированном программирования (ООП).

Раньше у нас был процедурный подход. Записывали алгоритм в виде набора процедур (функций). Сейчас будем записывать в виде набора объектов и писать как эти объекты могут взаимодействовать.

Класс - это новый тип данных, который вы можете написать.

Уже известные вам классы: int, str, list, dict и так далее.

Основные принципы ООП:

- *Инкапсуляция* - к атрибутам объекта доступаются не напрямую, а через методы. Для чего? Ограничиваем возможность сломать класс.
- *Наследование* - добавляем к существующему классу новые атрибуты и методы. Можно воспользоваться полями и методами родительских классов. Цепочки наследования. Множественное наследование.
- *Полиморфизм* - в классах-наследниках можно изменить методы родительского класса. По факту будет вызываться самый последний переопределенный метод в цепочке наследования.

Источники (рекомендую)

- Быстро и на пальцах:
  - tutorialspoint
- Медленно для начинающих:
  - Think Python by Downey
- Подробно:
  - Лутц. Изучаем Python
  - Саммерфилд. Программирование на Python 3
- Читать дальше патерны программирования:
  - Pyhon in Practice by Sammerfield
  - Гради Буч Объектно-ориентированный анализ и проектирование с примерами приложений

Объектно-ориентированное программирование

Процедурный и объектно-ориентированный подход

Процедурный подход хорош для небольших (до 500 строк) проектов. Объектно-ориентированно программировать можно и на ассемблере. Но трудно. Удобнее программировать на языке, где есть для этого возможности.

Посмотрим, что есть в питоне для объектно-ориентированного подхода.

Рассмотрим окружности на плоскости XY.

Для задания окружности нужно задать координаты центра окружности x, y и ее радиус r. Для этого можно использовать обычный кортеж.

```
circle = (1, 2, 15)
```

Проблемы:

- не очевидно, где тут радиус. Может быть (x, y, r), а может мы задавали (r, x, y)
- если есть функции `distance_from_origin(x, y)` (расстояние от начала координат до центра окружности) и `edge_distance_from_origin(x, y, radius)` (расстояние от начала координат до окружности), при обращении к ним нам нужно распаковывать кортеж (лишние операции):

  ```
  distance = distance_from_origin(*circle[:2])
  distance = edge_distance_from_origin(*circle)
  ```

  Решение: именованный кортеж (named tuple):

```
import collections
Circle = collections.namedtuple("Circle", "x y radius")
circle = Circle(13, 84, 9)
distance = distance_from_origin(circle.x, circle.y)
```

Проблемы:

- можно создать кортеж с отрицательным радиусом (нет проверки значений) (при

процедурном подходе эти проверки либо не делаются, либо требуют написания слишком много кода);
- tuple неизменяем, named tuple изменяем через метод collections.namedtuple_replace() (но код ужасает) `circle = circle._replace(radius=12)`

Решение: взять коллекцию с изменяемыми значениями, например, list или dict.

```
circle = [1, 2, 15]                  # list
circle = dict(x=36, y=77, radius=8)  # dict
```

- list
  - нет доступа к circle['x'],
  - можно сделать circle.sort()
- dict - решает проблемы из list, но
  - еще нет защиты от отрицательного радиуса,
  - еще можно передать в функцию, которая не работает с окружностями (а, например, ждет прямоугольник).

Решение: сделать свой новый тип данных, который будет представлять окружность на XY плоскости.

Новые типы данных

Новый тип данных в питоне называется **класс (class)**. Данные этого типа называются (как и раньше) **объектами** или **экземплярами класса (instance)**.

- **Класс** - это набор правил (способ создания и работы с однотипными объектами):
  - из каких переменных и данных состоит объект - **атрибуты** или **поля** - характеризуют состояние объекта;
  - что можно делать с экземпляром класса (поведение объекта) - **метод** или **функция** класса - могут менять состояние объекта.
  - **конструктор** - специальный метод, где описывается как создавать объект.

5 - это объект класса int. Для него определены арфиметические операции, его можно преобразовать в строку, его можно сделать из других объектов (строки, float). "Hello" - это объект класса str. Строки можно складывать, брать из них срезы, находить в ней подстроку, и так далее.

As a noun, "AT-trib-ute" is pronounced with emphasis on the first syllable, as opposed to "a-TRIB-ute", which is a verb. (Downey, p 144).

Создаем новый класс

- Для создания класса используют ключевое слово **class**
- поля и методы пишут с **отступом**
- имя класса принято писать **с большой буквы**

Класс записывается так (будем использовать сегодня):

```
class ИмяКласса:
    'Описание для чего нужен класс - строка документации (можно не писать)'
    поля и методы класса
```

или так (будем использовать, когда начнем говорить о наследовании):

```
class ИмяКласса(базовые классы через запятую):
    'Описание для чего нужен класс - строка документации (можно не писать)'
    поля и методы класса
```

*На самом деле, когда базовый класс не указывается явно, это класс **object***

Самый простой класс (мы использовали для создания своего исключения):

```
class MyOwnException(Exception):   # мой класс исключений наследуется от базового клас
са Exception
    pass                           # ничего дополнительного или особого он не делает,
просто существует
```

Пример класса, описывающего окружности:

```
from math import PI
class Circle:
    'Окружности на плоскости XY'
    # конструктор класса, вызывается когда создаем новый объект класса
    def __init__(self, x=0, y=0, r=1):
         self.x = x                    # все переменные ОБЪЕКТА указываются в констру
кторе
         self.y = y
         self.r = r

    # методы объекта:
    def area(self):                    # первый аргумент всегда self
         return PI * self.r * self.r    # доступ к аргументам - только через self.

    def perimetr(self):                # первый аргумент всегда self
         return 2*PI * self.r           # доступ к аргументам - только через self.

    def zoom(self, k):                 # увеличим окружность в k раз
         self.r *= k

    def is_crossed(self, c):           # пересекается или нет эта окружность с окружн
остью c?
        d2 = (self.x - c.x)**2 + (self.y - c.y)**2
        r2 = (self.r + c.r)**2
        return d2 <= r2

    def __str__(self):
        return 'Circle x={} y={} r={}, area={}'.format(self.x, self.y. self.r, self.ar
ea())
        # обратите внимание на вызов self.area() - вызываем другой метод этого объекта
 тоже через self

# тут можно уже создавать объекты класса и их использовать
```

- **__init__(self, другие аргументы через запятую)** - конструктор класса (не совсем так, там еще есть __new__, но об этом подробнее в наследовании и переопределении методов).
- **self** - ссылка на себя, через нее доступаемся к атрибутам (полям и методам)

self.метод - вызов другого метода класса

self - первый аргумент методов экземпляра класса.

Создание объекта (экземпляра класса)

```
c = Circle(1, 2, 3)
```

- Создается экземпляр класса Circle - окружность с центром в точке (1, 2) и радиусом 3
- ссылка на нее записывается в переменную `c`

Доступ к полям и методам

ссылка*на*объект.поле

ссылка*на*объект.метод

```
c = Circle()
c.x = 1
c.y = 2
c.r = 3
a = c.area()    # у объекта, на который ссылается c, вызвали метод area
```

Объекты и ссылки (повторение - мать учения)

```
c = Circle()            # создали единичную окружность с центром в (0,0), на нее ссыла
ется c
c.x = 1                 # эта окружность стала радиусом 3 с центром в (1, 2)
c.y = 2
c.r = 3
a = c.area()            # у объекта, на который ссылается c, вызвали метод area

d = Circle(5, 6, 2.5)   # создали окружность радиуса 2.5 с центром в (5, 6), на нее сс
ылается d
a = c.area() + d.area() # площадь окружности, на которую ссылается c и площадь окружно
сти, на которую ссылается d

c = d                   # теперь c ТОЖЕ ссылается на вторую окружность, ссылок на перв
ую окружность нет
```

c = d - это присвоение ссылок на объекты.

Что будет напечатано?

```
c = Circle()
d = Circle()
c.x = 1
d.x = 6
print('c.x = ', c.x)
print('d.x = ', d.x)

c = d
print('c.x = ', c.x)
print('d.x = ', d.x)
```

Функция возвращает объект

Method overloading

Можно ли создать в питоне методы класса с одинаковыми именами?

Нет. В python так не пишут.

Вы можете использовать значения по умолчанию, *args и **kargs - в питоне есть другие механизмы для того же результата.

Можно использовать декоратор `@overload`, но о декораторах расскажем позже.

Термины (заключение)

- **Модель** - упрощенное описание
- **Объект**:
  - Переменные (характеризуют состояние объекта)
  - Методы (могут состояния менять)
- **Класс** – способ задания однотипных объектов
  - прототип объекта (его переменные)
  - метод создания из прототипа конкретного объекта

Еще раз "на пальцах":

- Класс - **шаблон** объекта,
- Объект - **экземпляр** класса.

Задачи

Uno card game (на дом)

Добавить карты +2, skip, wild, wild+4

Инкапсуляция

В python НЕТ возможности полностью ограничить доступ к переменным объекта и класса

Если вы изучали другие ООП-языки, то можете считать, что все поля в питоне public, а все методы virtual.

Зачем нужна инкапсуляция

Пусть мы пишем класс `Circle` для хранения окружностей в плоскости XY.

Что будет, если кто-то установит отрицательный радиус?

Можно дописать функцию проверки корректности радиуса и добавить ее в код.

```
class Circle:
    # конструктор класса, вызывается когда создаем новый объект класса
    def __init__(self, x=0, y=0, r=1):
         self.x = x                    # все переменные ОБЪЕКТА указываются в констру
кторе
         self.y = y
         self.r = r

    def set_r(self, r):
        if r < 0:
            raise ValueError('radius is {}'.format(r))
        self.r = r

    # другие методы класса
```

Хочется заставить других программистов присваивать радиус только через функцию set_r.

_x - джентельменское соглашение (псевдочастные имена)

В питоне придерживаются неофициального соглашения, что переменные, начинающиеся с _ не надо изменять вне класса. Они для внутреннего использования в классе.

Многие IDE не делают автодополнения вне класса на эти поля.

Но это просто договоренность. Вы можете использовать и изменять такие переменные где угодно.

__x - искажение

Если имя внутри конструкции `class` начинается с двух подчеркиваний __ за счет имени того класса, в котором они определены. Например, в классе Circle переменная __r не доступна вне класса по этому имени, но доступна по имени _Circle__r

```
class Circle:
    # конструктор класса, вызывается когда создаем новый объект класса
    def __init__(self, x=0, y=0, r=1):
         self.x = x                    # полностью открытое имя переменной
         self._y = y                   # частично закрытое
         self.__r = r                  # "закрытое" имя

    def set_r(self, r):
        if r < 0:
            raise ValueError('radius is {}'.format(r))
        self.__r = r

    # другие методы класса

c = Circle(1, 2, 3)
print(c.x)          # никак не ограничено
print(c._y)         # интерпретатор не ограничивает, коллеги осуждают
print(c.__r)        # нельзя, AttributeError
print(c._Circle__r) # можно, 3
```

В разделе "Подробнее об ООП" мы вернемся к этому примеру и рассмотрим как можно запретить добавление новых атрибутов и запись в уже существующие.

Права доступа в стране розовых пони

Мы помним, что присваивание создавало переменные.

```
x = 4   # если x не было, создать его
```

Аналогично присвоением можно экземпляру класса добавить атрибуты.

В питоне синтаксически можно написать так (но не нужно!!!):

```
class A:    # в классе нет метода __init__ и атрибутов экземпляров класса.
    pass

a = A()     # a - ссылка на созданный объект класса A
a.x = 1     # добавили этому объекту поле x и присвоили ему 1

b = A()     # b - ссылка на другой созданный объект класса A
b.y = 2     # добавили этому объекту поле y и присвоили ему 2

print(a.x)  # 1
print(b.y)  # 2
print(a.y)  # AttributeError: 'A' object has no attribute 'y'

print(A.__dict__) # {'__module__': '__main__', '__dict__': <attribute '__dict__' of 'A
' objects>, '__weakref__': <attribute '__weakref__' of 'A' objects>, '__doc__': None}
print(a.__dict__) # {'x': 1}
print(b.__dict__) # {'y': 2}
```

Заметьте, атрибуты x и y принадлежат не экземплярам класса (всем), а x - одному экземпляру, y - другому экземпляру. Появление поля x в одном экземпляре класса не означает, что оно появится в другом.

Как такое создание атрибутов объекта может смутить программистов? Рассмотрим измененный пример кода с доступом к полю __r.

```
class Circle:
    # конструктор класса, вызывается когда создаем новый объект класса
    def __init__(self, x=0, y=0, r=1):
         self.x = x                    # полностью открытое имя переменной
         self._y = y                   # частично закрытое
         self.__r = r                  # "закрытое" имя

    def get_r(self):
        return self.__r

    # другие методы класса

c = Circle(1, 2, 3)
print(c.__r)        # нельзя, AttributeError
c.__r = 22          # МОЖНО??? Да, можно. Мы в одном объекте класса добавили атрибут
__r
print(c.__r)        # 22 (раньше получали AttributeError)
print(c._Circle__r) # можно, 3 (c._Circle__r правильное полное "внешнее" имя атрибута
self.__r)
print(c.get_r())    # 3
```

Как мы видим, у объекта, на который ссылается переменная `c`, появился новый атрибут `c.\_\_r` равный 22. При этом атрибут `c._Circle\_\_r` остался недоступным и все еще равен 3.

Еще раз: **К атрибутам экземпляра класса, начинающихся с двойного подчеркивания, нельзя обратиться вне класса по этому имени. Полное имя атрибута _имякласса__имяатрибута**

Копирование объектов

Зададим прямоугольник как координаты левой верхней точки, его ширину и высоту.

```
>>> class Point():
>>>     def __init__(self, x=0, y=0):
>>>         self.x = x
>>>         self.y = y
>>>
>>>     def __str__(self):
>>>         return '({}, {})'.format(self.x, self.y)
>>>
>>> class Rect():
>>>     def __init__(self, x=0, y=0, w=0, h=0):
>>>         self.lt = Point(x, y)
>>>         self.width = w
>>>         self.height = h

>>> import copy
>>> p1 = Point(1, 2)
>>> p2 = copy.copy(p1)
>>> print(p1)           # (1,2)
>>> print(p2)           # (1,2)
>>> p1 is p2            # это другой объект (копия)
False
>>> p1 == p2            # не переопределена операция __eq__ - по умолчанию == как is
False

>>> r1 = Rect(0, 0, 100, 200)
>>> r2 = copy.copy(r1)
>>> r1 is r2
False
>>> r1.lt is r2.lt
True

>>> r3 = copy.deepcopy(r1)
>>> r1 is r3
False
>>> r1.lt is r3.lt
False
```

TODO: Нарисовать диаграмму для r1 и r2 со ссылкой на общую точку, как в Downey, p 148.

https://docs.python.org/3.6/library/copy.html

- A **shallow copy** constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
- A **deep copy** constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

Two problems often exist with deep copy operations that don't exist with shallow copy operations:

- Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.
- Because deep copy copies everything it may copy too much, such as data which is intended to be shared between copies.

Методы класса и статические методы

Когда интерпретатор достигает инструкции class (а не тогда, когда происходит вызов класса), он выполняет все инструкции в ее теле от начала и до конца. Все присваивания, которые производятся в ходе этого процесса, создают имена в локальной области видимости класса, которые становятся атрибутами объекта класса.

Благодаря этому классы напоминают модули и функции:

- Подобно функциям, инструкции class являются локальными областями видимости, где располагаются имена, созданные вложенными операциями присваивания.
- Подобно именам в модуле, имена, созданные внутри инструкции class, становятся атрибутами объекта класса.

Для работы с классом питон создает отдельный объект, описывающий весь класс целиком как набор правил, а не отдельный экземпляр класса. (class object).

У этого объекта тоже могут быть свои поля и методы.

Они нужны для атрибутов и методов, которые относятся не к конкретному экземпляру, а ко всему классу целиком.

Например, для класса, описывающих дату, это может быть список названий месяцев.

№ Переменные класса

```
class Date():
    month = ['январь', 'февраль', ..]
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year
        self.month_name = Date.month[month-1]
```

Попробуем посчитать, сколько экземпляров класса Circle было создано за время работы программы.

```
class Circle():
    counter = 0   # сколько экземпляров класса было создано
    def __init__(self, x=0, y=0, r=1):
        Circle.counter += 1

c = Circle()
d = Circle()
print(
```

методы класса и статические методы

@classmethod можно переопределить в наследнике класса

У метода класса есть **cls**, но нет **self**

@staticmethod нельзя переопределить при наследовании классов

```
class Date(object):

    def __init__(self, day=0, month=0, year=0):
        self.day = day
        self.month = month
        self.year = year

    @classmethod
    def from_string(cls, date_as_string):
        day, month, year = map(int, date_as_string.split('-'))
        date1 = cls(day, month, year)
        return date1

    @staticmethod
    def is_date_valid(date_as_string):
        day, month, year = map(int, date_as_string.split('-'))
        return day <= 31 and month <= 12 and year <= 3999

date2 = Date.from_string('11-09-2012')
is_date = Date.is_date_valid('11-09-2012')
```

Пример static factory

Допустим, у нас должно быть не более 1 экземпляра данного класса.

Когда какой метод делаем?

Метод:

- **экземпляра класса** (self) - функция обращается к атрибутам экземпляра;
- **класса** - функция не обращается к атрибутам *экземпляра* класса, но обращается к атрибутам класса;
- **static** - функция не обращается ни к каким атрибутам класса или объекта.

Источники (рекомендую)

- Быстро и на пальцах:
  - tutorialspoint
- Медленно для начинающих:
  - Think Python by Downey
- Подробно:
  - Лутц. Изучаем Python
  - Саммерфилд. Программирование на Python 3
- Читать дальше патерны программирования:
  - Pyhon in Practice by Sammerfield
  - Гради Буч Объектно-ориентированный анализ и проектирование с примерами приложений

Наследование

Мы разобрали принципы A *состоит из* B, C, D (композиция).

Разберем наследование - изменение свойств класса под изменяющиеся требования задачи.

Создадим класс, в котором будем записывать для каждого человека его имя и зарплату. Это может быть студент в институте, может быть преподаватель.

Внимание: последующие цифры зарплат и алгоритмы рассчета не имеют ничего общего с реальными зарплатами и расценками за работу. Любое совпадение с реальными данными считайте случайным.

Как делаем классы:

1. Создаем экземпляр класса
   - конструктор
   - тестируем
2. Методы, которые определяют поведение
3. Перегрузка операторов
4. Особое поведение - наследуем класс
   - создаем класс-наследник;

    - расширяем методы
5. Изменяем конструкторы
6. Инструменты интроспекции (как получать информацию о классе во время отладки)
7. Сохраняем объекты в базе данных.

Шаг 1. Создаем экземпляр класса

Пишем класс в файле person.py

Нужно описать класс и его конструктор.

Подумаем что нужно хранить о каждом работнике.

Его имя (полное), кем работет, его зарплату.

Человек может еще только приниматься на работу или его уже уволили. Тогда работы нет и зарплата 0. Запишем это в конструкторе.

```
class Person(object):
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
```

Сразу начинаем тестировать класс: создаем экземпляры класса, печатаем значение их полей

```
class Person(object):
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

# Конец класса Person

# Тестируем класс:
bob = Person('Boris Ivanov')
mike = Person('Mike Kuznetsov', job='student', pay=5000)

print(bob.name, bob.pay)        # Boris Ivanov 0
print(mike.name, mike.pay)      # Mike Kuznetsov 5000
```

bob и mike определяют каждое свое пространство имен. Т.е. поля name и pay в объекте bob не совпадают с полями name и pay в объекте mike, так как каждый экземпляр класса имеет свой набор атрибутов (name, job, pay).

Тестирование и выполнение

Мы написали тесты, но они всегда выполняются и при import этого модуля. Это не нужно.

Можно тесты положить в отдельные файлы (и это хорошо!). Можно написать тесты с использованием библиотек docstring, unittest, pytest и так далее.

Пока будем писать тесты в том же файле, но запускать их только тогда, когда мы запускаем непосредственно файл, а не импортируем его в другие файлы.

```
python person.py
```

Для этого будем проверять, как запускается файл, используя атрибут __name__ модуля:

```
class Person(object):
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

# Конец класса Person

if __name__ == '__main__':
    # Тестируем класс, только если запускаем файл
    bob = Person('Boris Alexeevich Ivanov')
    mike = Person('Mikhail Vladimirovich Kuznetsov', job='student', pay=5000)

    print(bob.name, bob.pay)       # Boris Alexeevich Ivanov 0
    print(mike.name, mike.pay)     # Mikhail Vladimirovich Kuznetsov 5000
```

Шаг 2. Добавляем методы, которые определяют поведение

У человека можно узнать его фамилию (а не полное имя), фамилию можно печатать с инициалами (первые буквы имени и отчества - first name, parent name).

Сотрудник может работать не на целую ставку, а меньше (например, работать половину времени и получать 0.5 зарплаты от целой ставки).

Если мы будем прямо в коде везде писать

```
bob.name.split()[2]         # Ivanov
mike.pay = mike.pay * 0.5   # 0.5 ставки
```

то потом, когда нужно будет поправить этот код, он будет в разных местах программы и использоваться разными людьми.

Например, у нас полное имя может состоять из 2 слов или 4 и более слов. Тогда нужно будет фамилию извлекать как "последнее слово полного имени", а не "слово с индексом 2". Поэтому получение фамилии нужно сделать функцией.

О зарплате: назовем лучше размер 1 ставки base_pay, добавим атрибут part_time, а метод pay

```
class Person(object):
    def __init__(self, name, job=None, pay=0, part_time=1):
        self.name = name
        self.job = job
        self.base_pay = pay
        self.part_time = 1

    def last_name(self):
        return self.name.split()[2]

    def pay(self):
        return int(self.base_pay * self.part_time)

# Конец класса Person

if __name__ == '__main__':
    # Тестируем класс, только если запускаем файл
    bob = Person('Boris Alexeevich Ivanov')
    mike = Person('Mikhail Vladimirovich Kuznetsov', job='student', pay=5000, part_tim
e=0.5)

    print(bob.name, bob.base_pay)       # Boris Alexeevich Ivanov 0
    print(mike.name, mike.base_pay)     # Mikhail Vladimirovich Kuznetsov 5000

    # добавили код - добавим тесты
    print(bob.last_name())          # Ivanov
    print(mike.last_name())         # Kuznetsov

    print(mike.pay())               # 2500
```

Если нужно будет изменить округление при подсчете зарплаты, то мы легко делаем это в *одном месте* - методе класса.

Шаг 3. Перегрузка операторов

Неудобно при тестировании печатать каждый атрибут отдельно. Хочется легко печатать всю информацию об объекте. Но `print(bob)` печатает что-то вроде `<__main__.Person object at 0x02614430>`.

Чтобы напечатать информацию об объекте типа Person, нужно этот объект представить в виде строки, то есть вызвать `str(bob)` (вызывается автоматически). Эта функция автоматически вызывает `a.__str__()`.

Внешняя информация о сотруднике - сколько он получает. Внутренняя - из чего складывается эта зарплата.

Чтобы наш объект удобно печатался надо переопределить функцию **__str__**. И изменим тесты, вызывая `print(bob)` и `print(mike)`.

```
class Person(object):
    def __init__(self, name, job=None, pay=0, part_time=1):
        self.name = name
        self.job = job
        self.base_pay = pay
        self.part_time = part_time

    def __str__(self):
        return '[Person: {}, {}]'.format(self.name, self.pay())

    def last_name(self):
        return self.name.split()[2]

    def pay(self):
        return int(self.base_pay * self.part_time)

# Конец класса Person

class Teacher(Person):
    pass

if __name__ == '__main__':
    # Тестируем класс, только если запускаем файл
    bob = Person('Boris Alexeevich Ivanov')
    mike = Person('Mikhail Vladimirovich Kuznetsov', job='student', pay=5000, part_tim
e=0.5)

    print(bob)                      # [Person: Boris Alexeevich Ivanov, 0]
    print(mike)                     # [Person: Mikhail Vladimirovich Kuznetsov, 2500]

    # добавили код - добавим тесты
    print(bob.last_name())          # Ivanov
    print(mike.last_name())         # Kuznetsov
```

Замечание об __str__ и __repr__: оба этих метода преобразуют объект к строке. Но __str__ обычно используют для представления данных в удобном для чтения пользователем виде (и именно его вызовет метод print), а __repr__ чаще пишут так, чтобы было удобно читать отладочную информацию или выполнять полученную строку как код.

Интерпретатор вызывает __repr__.

Если функции __str__ нет, то вызывается автоматически __repr__.

Шаг 4. Дополнительное поведение в подклассах

Часть сотрудников у нас проводит занятия. Занятия оплачиваются по часам. Допустим, что каждый 1 час стоит 200 рублей. Это расширенная возможность. Значит, расширим наш класс Person так, чтобы у преподавателей была возможность получать кроме базовой части зарплаты еще и почасовую оплату.

Чтобы расширить класс Person (а не добавлять каждому студенту возможность почасовой оплаты стипендии за каждое занятие), создадим новый класс Teacher на основе Person.

```
class Teacher(Person):
    тут опишем что добавили к базовому классу Person, чтобы получился Teacher
```

Как можно написать класс Teacher? Неправильно, но просто - скопировать нужный метод и изменить его.

НЕПРАВИЛЬНО (но будет работать):

```
class Teacher(Person):
    def __init__(self, name, job=None, pay=0, part_time=1, hours=0):
        self.name = name
        self.job = job
        self.base_pay = pay
        self.part_time = part_time
        self.hours = hours

   def pay(self):
        return int(self.base_pay * self.part_time + self.hours * 200)
```

Почему это неправильно? Потому что копируя код вы делаете сложным поддержку этого кода. Теперь если нужно бдует исправлять формулу подсчета `self.base_pay * self.part_time` , ее нужно будет исправить в 2 местах.

Как писать правильно? Нужно *использовать* уже написанный код.

Правильно:

```
class Teacher(Person):
    def __init__(self, name, pay=0, part_time=1, hours=0):
        super().__init__(name, 'teacher', pay, part_time)
        self.hours = hours

    def pay(self):
        return super().pay() + self.hours * 200
```

Почему правильно? Нужно только *дополнить* существующий метод, а не заменить его. Поэтому нужно вызывать метод базового класса при вызове метода наследника и потом дополнять результат.

Разберем как можно вызывать методы базового класса из наследника.

Вызов

```
instance.method(args...)
```

автоматически заменяется на вызов

```
class.method(instance, args...)
```

`self.pay()` внутри метода `pay()` вызывать нельзя, получится рекурсия.

Можно вызвать непосредственно метод базового класса как `Person.pay(self)`. Это отменит поиск в дереве наследования, так как мы сразу указываем класс. Плюс: быстрее. Минус: если потом будем писать класс между Person и Teacher, то придется проверять весь код класса Teacher и заменять вызов "метода родителя" на другого родителя. Мы заботимся о возможных изменениях.

```
super().method(args...)     # вызвать этот метод у базового класса
```

Напишем именно так.

Добавим тестов:

```
    tanya = Teacher(name='Tatyana Vladimirovna Ovsyannikova', job='teacher', pay=10000
, hours=6*4)
    print(tanya.pay())              # вызов измененной версии pay класса Teacher
    print(tanya.last_name())        # вызов унаследованного метода класса Person
    print(tanya)                    # вызов унаследованного метода класса Person
```

Полиморфизм

Добавим еще тестов:

```
    for p in (bob, mike, tanya):
        print(p.pay())
        print(p)
```

Этот код выведет:

```
0
[Person: Boris Alexeevich Ivanov, 0]
2500
[Person: Mikhail Vladimirovich Kuznetsov, 2500]
14800
[Person: Tatyana Vladimirovna Ovsyannikova, 14800]
```

`p` может быть как объектом класса Person, так и объектом класса Teacher.

`p.pay()` - вызывается (в зависимости от того, какой *реально* тип у объекта `p`) метод либо класса Person, либо класса Teacher.

`print(p)` - вызвается метод `p.__str__()` класса Person (так как у Teacher этого метода нет). Обратите внимание, этот метод вызывает `self.pay()` . И вызывается в зависимости от того, какой это *реальный* объект, либо `Person.pay(self)` , либо `Teacher.pay(self)` .

Вызов метода того класса, к которому принадлежит реальный объект, называют полиморфизмом.

Что мы можем сделать для нового поведения?

```
class Person(object):
    def last_name(self): ...
    def pay(self): ...
    def __str__(self): ...

class Teacher(Person):      # наследование
    def pay(self): ...      # адаптация (изменение)
    def for_books(self):... # расширение (дополнительные методы)

dima = Teacher()
dima.last_name()            # унаследованный метод
dima.pay()                  # адаптированная (измененная) версия
dima.for_books()            # дополнительный метод
print(dima)                 # унаследованный перегруженный метод
```

Шаг 5. Изменим конструкторы

Заметьте, что в классе Teacher не только pay() вызывает метод базового класса.

Нам понадобилось изменить конструктор. В него добавлось поле `self.hours` , был вызван конструктор базового класса и мы явно указали при его вызове, что `job='teacher'`

```
    def __init__(self, name, pay=0, part_time=1, hours=0):
        super().__init__(name, 'teacher', pay, part_time)
        self.hours = hours
```

Мы передаем конструктору суперкласса только необходимые аргументы.

Можно вообще не вызывать конструктор базового класса, а полностью его переписать в конструкторе наследника.

Альтернатива наследованию - композиция и getattr

Существует альтернативный шаблон проектирования - делегирование. Когда мы пишем обертку вокруг вложенного объекта, эта обертка управляет вложенным объектом и перенаправляет ему вызовы методов.

Можно вместо наследования Teacher от Person, сделать Person одним из атрибутов Teacher (полный текст примера смотри в файле person2.py примеров к разделу):

```
class Teacher(object):      # Teacher НЕ наследует от Person
    HOUR_RATE = 200
    def __init__(self, name, pay=0, part_time=1, hours=0):
        self.person = Person(name, 'teacher', pay, part_time)   # вложенный объект
        self.hours = hours

    def pay(self):
        # перехватывает обращение и делегирует его к другим методам
        return self.person.pay() + self.hours * self.HOUR_RATE

    def __getattr__(self, attr):
        # делегирует обращения ко всем остальным атрибутам
        return getattr(self.person, attr)

    def __str__(self):
        # тоже требуется перегрузить
        return str(self.person)

if __name__ == '__main__':
    tanya = Teacher(name='Tatyana Vladimirovna Ovsyannikova', pay=10000, hours=6*4)
    print(tanya.pay())              # 14800
    print(tanya.last_name())        # Ovsyannikova
    print(tanya)                    # [Person: Tatyana Vladimirovna Ovsyannikova, 1000
0]
```

Заметьте, `tanya.pay()` посчитало правильную зарплату с часами, а `str(tanya)` показывает зарплату БЕЗ часов. 10000 вместо 14800.

Доступ к полям и методам разберем позже.

Без переопределения __str__ не вызывается из Person, хотя в Person определен метод __str__.

Лутц, стр 750: встроенные операции, например вывод и обращение к элементу по индексу, неявно вызывают методы перегрузки операторов, такие как __str__ и __getitem__.

В версии 3.0 встроенные операции, подобные этим, не используют менеджеры атрибутов для неявного получения ссылок на атрибуты: они не используют ни метод __getattr__ (вызывается при попытке обращения к неопределенным атрибутам), ни родственный ему метод __getattribute__ (вызывается при обращении к любым атрибутам). Именно по этой причине нам потребовалось переопределить метод __str__ в альтернативной реализации класса Teacher, чтобы обеспечить вызов метода встроенного объекта Person при запуске сценария под управлением Python 3.0.

Технически это обусловлено тем, что при работе со старыми классами интерпретатор пытается искать методы перегрузки операторов в экземплярах, а при работе с классами нового стиля - нет. Он вообще пропускает экземпляр и пытается отыскать требуемый метод в классе.

В версии 2.6 встроенные операции, при применении к экземплярам классических классов, выполняют поиск атрибутов обычным способом. Например, операция вывода пытается отыскать метод __str__ с помощью метода __getattr__. Однако в версии 3.0 классы нового стиля наследуют метод __str__ по умолчанию, что мешает работе метода __getattr__, а метод __getattribute__ вообще не перехватывает обращения к подобным именам.

Это проблема, но вполне преодолимая, – классы, опирающиеся на прием делегирования, в версии 3.0 в общем случае могут переопределять методы перегрузки операторов, чтобы делегировать вызовы вложенным объектам, либо вручную, либо с помощью других инструментов или суперклассов.

```
class Person:               # старый класс
class Person(object): ...   # новый класс
```

Шаг 6. Как показывать информацию об объектах

При печати `print(tanya)` плохо: `[Person: Tatyana Vladimirovna Ovsyannikova, 14800]`

- Пишется 'Person', хотя фактически класс Teacher. (Хотя объект Teacher - это измененный Person, но не хотелось бы печатать имя реального класса.
- Выводятся только те атрибуты, что мы руками указали в __str__. О поле hours ничего не печатается. Это значит, что придется делать много лишней работы, выписывая руками каждый добавленный атрибут.

Как избежать лишней работы? (Больше кода - больше шанс сделать ошибку).

Что уже есть?

- instance.**__class__** - из экземпляра *instance* ссылка на класс (класс - это тоже объект).
- у класса есть атрибут **__name__** - имя (как у модуля), хранит имя класса (у нас 'Person', 'Teacher')
- у класса есть **__bases__** - последовательность ссылок на базовые классы.
- у объекта (экземпляр или сам класс) есть атрибут **__dict__** - список всех полей и методов класса в виде пар ключ (название атрибута) и значение (ссылка на значение).
- **dir(obj)** - включаем еще унаследованные атрибуты и методы (использование в коде: `list(dir(bob))`)

Посмотрим, как они работают:

```
>>> from person import Person
>>> bob = Person('Bob Smith')
>>> print(bob)                  # Вызов метод __str__ объекта bob
[Person: Bob Smith, 0]
>>> bob.__class__               # Выведет класс объекта bob и его имя
<class 'person.Person'>
>>> bob.__class__.__name__
'Person'
>>> list(bob.__dict__.keys())   # Атрибуты – это действительно ключи словаря
['base_pay', 'job', 'name']     # Функция list используется для получения
                                # полного списка в версии 3.0
>>> for key in bob.__dict__:
        print(key, '=>', bob.__dict__[key]) # Обращение по индексам
pay => 0
job => None
name => Bob Smith
>>> for key in bob.__dict__:
        print(key, '=>', getattr(bob, key)) # Аналогично выражению obj.attr,
                                            # где attr - переменная
pay => 0
job => None
name => Bob Smith
```

Для того, чтобы печатать правильное имя класса, используйте `self.__class__.__name__`

Чтобы распечатать все атрибуты, вызываем

```
def str_attrs(self):
    attr = []
    for k in sorted(self.__dict__):
        attr.append('{}={}'.format(k, getattr(self, k))
    return ' '.join(attr)
```

Шаг 7. Сохраним объекты в базе данных

Используйте для этого модули:

- *pickle* - Преобразует произвольные объекты на языке Python в строку байтов и обратно.
- *dbm* - Реализует сохранение строк в файлах, обеспечивающих возможность обращения по ключу.
- *shelve* - Использует первые два модуля, позволяя сохранять объекты в файлах-хранилищах, обеспечивающих возможность обращения по ключу.

Лутц стр 757 и далее.

Права доступа и область видимости

Лутц, стр 773

Имена и области видимости

Лутц p.783

mynames.py:

```
X = 11 # Глобальное (в модуле) имя/атрибут (X, или manynames.X)

def f():
    print(X) # Обращение к глобальному имени X (11)

def g():
    X = 22 # Локальная (в функции) переменная (X, скрывает имя X в модуле)
    print(X)

class C:
    X = 33 # Атрибут класса (C.X)
    def m(self):
        X = 44 # Локальная переменная в методе (X)
        self.X = 55 # Атрибут экземпляра (instance.X)
```

Встроенные атрибуты класса

https://www.tutorialspoint.com/python/python_classes_objects.htm

- __dict__ - Dictionary containing the class's namespace.
- __doc__ - Class documentation string or none, if undefined.
- __name__ - Class name.
- __module__ - Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- __bases__ - A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

```
#!/usr/bin/python

class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

Получаем:

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

Источники (рекомендую)

- Быстро и на пальцах:
  - tutorialspoint
- Медленно для начинающих:
  - Think Python by Downey
- Подробно:
  - Лутц. Изучаем Python
  - Саммерфилд. Программирование на Python 3
- Читать дальше патерны программирования:
  - Pyhon in Practice by Sammerfield
  - Гради Буч Объектно-ориентированный анализ и проектирование с примерами приложений

Задачи

Вспомните про unittest и по возможности оформите тесты как юниттесты.

Реализуйте класс Matrix.

Вся матрица должна хранится в виде 1 (ОДНОГО ОДНОМЕРНОГО) списка.

Он должен содержать:

Конструктор

Конструктор от списка списков. Гарантируется, что списки состоят из чисел, не пусты и все имеют одинаковый размер. Конструктор должен копировать содержимое списка списков, т.е. при изменении списков, от которых была сконструирована матрица, содержимое матрицы изменяться не должно.

__str__

Метод __str__ переводящий матрицу в строку. При этом элементы внутри одной строки должны быть разделены знаками табуляции, а строки — переносами строк. При этом после каждой строки не должно быть символа табуляции и в конце не должно быть переноса строки.

size

Метод size без аргументов, возвращающий кортеж вида (число строк, число столбцов)

reshape

Устанавливает новые параметры количества строк и столбцов. С проверкой на корректность.

```
Тест 1
Входные данные:
# Task 1 check 1
m = Matrix([[1, 0], [0, 1]])
print(m)
m = Matrix([[2, 0, 0], [0, 1, 10000]])
print(m)
m = Matrix([[-10, 20, 50, 2443], [-5235, 12, 4324, 4234]])
print(m)

Вывод программы:
1    0
0    1
2    0    0
0    1    10000
-10    20    50    2443
-5235    12    4324    4234

Тест 2
Входные данные:
# Task 1 check 2
m1 = Matrix([[1, 0, 0], [1, 1, 1], [0, 0, 0]])
m2 = Matrix([[1, 0, 0], [1, 1, 1], [0, 0, 0]])
print(str(m1) == str(m2))
Вывод программы:
True

Тест 3
Входные данные:
# Task 1 check 3
m = Matrix([[1, 1, 1], [0, 100, 10]])
print(str(m) == '1\t1\t1\n0\t100\t10')
Вывод программы:
True
```

сложение и умножение

- __add__ принимающий вторую матрицу того же размера и возвращающий сумму матриц
- __mul__ принимающий число типа int или float и возвращающий матрицу, умноженную на скаляр
- __rmul__ делающий то же самое, что и __mul__. Этот метод будет вызван в том случае, аргументнаходится справа. Можно написать __rmul__ = __mul__

В этом случае вызовется __mul__: Matrix([[0, 1], [1, 0]) *10 В этом случае вызовется __rmul__ (так как у int не определен __mul__ для матрицы справа): 10* Matrix([[0, 1], [1, 0])

Разумеется, данные методы не должны менять содержимое матрицы.

```
Тест 1
Входные данные:
# Task 2 check 1
m = Matrix([[10, 10], [0, 0], [1, 1]])
print(m.size())
Вывод программы:
(3, 2)



Тест 2
Входные данные:
# Task 2 check 2
m1 = Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
m2 = Matrix([[0, 1, 0], [20, 0, -1], [-1, -2, 0]])
print(m1 + m2)
Вывод программы:
1    1    0
20    1    -1
-1    -2    1



Тест 3
Входные данные:
# Task 2 check 3
m = Matrix([[1, 1, 0], [0, 2, 10], [10, 15, 30]])
alpha = 15
print(m * alpha)
print(alpha * m)
Вывод программы:
15    15    0
0    30    150
150    225    450
15    15    0
0    30    150
150    225    450
```

__add__ - проверка корректности

Добавьте в программу из предыдущей задачи класс MatrixError, содержащий внутри self поля matrix1 и matrix2 (ссылки на матрицы).

Добавьте в метод **add** проверку на ошибки в размере входных данных, чтобы при попытке сложить матрицы разных размеров было выброшено исключение MatrixError таким образом, чтобы matrix1 поле MatrixError стало первым аргументом **add** (просто self), а matrix2 — вторым (второй операнд для сложения).

транспонирование

- Реализуйте метод transpose, транспонирующий матрицу и возвращающую результат (данный метод модифицирует экземпляр класса Matrix)
- Реализуйте статический метод transposed, принимающий Matrix и возвращающий транспонированную матрицу.

Подумайте, надо ли делать транспонирование методом класса или статическим методом?

```
Тест 1
Входные данные:
# Task 3 check 1
# Check exception to add method
m1 = Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
m2 = Matrix([[0, 1, 0], [20, 0, -1], [-1, -2, 0]])
print(m1 + m2)

m2 = Matrix([[0, 1, 0], [20, 0, -1]])
try:
    m = m1 + m2
    print('WA\n' + str(m))
except MatrixError as e:
    print(e.matrix1)
    print(e.matrix2)
Вывод программы:
1    1    0
20    1    -1
-1    -2    1
1    0    0
0    1    0
0    0    1
0    1    0
20    0    -1



Тест 2
Входные данные:
# Task 3 check 2
m = Matrix([[10, 10], [0, 0], [1, 1]])
print(m)
m1 = m.transpose()
print(m)
print(m1)
Вывод программы:
10    10
0    0
1    1
10    0    1
10    0    1
```

```
10    0    1
10    0    1



Тест 3
Входные данные:
# Task 3 check 3
m = Matrix([[10, 10], [0, 0], [1, 1]])
print(m)
print(Matrix.transposed(m))
print(m)
Вывод программы:
10    10
0    0
1    1
10    0    1
10    0    1
10    10
0    0
1    1
```

solve

Пусть экземпляр класса Matrix задаёт систему линейных алгебраических уравнений.

Тогда добавьте в класс метод solve, принимающий вектор-строку свободных членови возвращающий строку-список, состоящую из float — решение системы, если оно единственно. Если решений нет или оно не единственно — выдайте какую-нибудь ошибку.

```
Тест 1
Входные данные:
# Task 5 check 1
m = Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
print(m.solve([1,1,1]))
Вывод программы:
[1.0, 1.0, 1.0]



Тест 2
Входные данные:
# Task 5 check 2
m = Matrix([[1, 1, 1], [0, 2, 0], [0, 0, 4]])
print(m.solve([1,1,1]))
Вывод программы:
[0.25, 0.5, 0.25]



Тест 3
Входные данные:
# Task 5 check 3
m = Matrix([[1, 1, 1], [0, 1, 2], [0.5, 1, 1.5]])
try:
    s = m.solve([1,1,1])
    print('WA No solution')
except Exception as e:
    print('OK')
Вывод программы:
OK
```

возведение в степень

К программе в предыдущей задаче добавьте класс SquareMatrix — наследник Matrix с операцией возведенияв степень __pow__, принимающей натуральную (с нулём) степень, в которую нужно возвести матрицу. Используйте быстрое возведение в степень.

```
Тест 1
Входные данные:
# Task 6 check 1
m = SquareMatrix([[1, 0], [0, 1]])
print(isinstance(m, Matrix))
Вывод программы:
True
```

```
Тест 2
Входные данные:
# Task 6 check 2
m = SquareMatrix([[1, 0], [0, 1]])
print(m ** 0)
Вывод программы:
1    0
0    1



Тест 3
Входные данные:
# Task 6 check 3
m = SquareMatrix([[1, 1, 0, 0, 0, 0],
                  [0, 1, 1, 0, 0, 0],
                  [0, 0, 1, 1, 0, 0],
                  [0, 0, 0, 1, 1, 0],
                  [0, 0, 0, 0, 1, 1],
                  [0, 0, 0, 0, 0, 1]]
                )
print(m)
print('----------')
print(m ** 1)
print('----------')
print(m ** 2)
print('----------')
print(m ** 3)
print('----------')
print(m ** 4)
print('----------')
print(m ** 5)
Вывод программы:
1    1    0    0    0    0
0    1    1    0    0    0
0    0    1    1    0    0
0    0    0    1    1    0
0    0    0    0    1    1
0    0    0    0    0    1
----------
1    1    0    0    0    0
0    1    1    0    0    0
0    0    1    1    0    0
0    0    0    1    1    0
0    0    0    0    1    1
0    0    0    0    0    1
----------
1    2    1    0    0    0
0    1    2    1    0    0
0    0    1    2    1    0
0    0    0    1    2    1
0    0    0    0    1    2
0    0    0    0    0    1
```

```
----------
1    3    3    1    0    0
0    1    3    3    1    0
0    0    1    3    3    1
0    0    0    1    3    3
0    0    0    0    1    3
0    0    0    0    0    1
----------
1    4    6    4    1    0
0    1    4    6    4    1
0    0    1    4    6    4
0    0    0    1    4    6
0    0    0    0    1    4
0    0    0    0    0    1
----------
1    5    10    10    5    1
0    1    5    10    10    5
0    0    1    5    10    10
0    0    0    1    5    10
0    0    0    0    1    5
0    0    0    0    0    1
```

Uno card game (на дом)

Добавить карты +2, skip, wild, wild+4

Обзор главы

В главе рассказывается об объектно-ориентированном программировании (ООП).

Раньше у нас был процедурный подход. Записывали алгоритм в виде набора процедур (функций). Сейчас будем записывать в виде набора объектов и писать как эти объекты могут взаимодействовать.

Класс - это новый тип данных, который вы можете написать.

Уже известные вам классы: int, str, list, dict и так далее.

Основные принципы ООП:

- *Инкапсуляция* - к атрибутам объекта доступаются не напрямую, а через методы. Для чего? Ограничиваем возможность сломать класс.
- *Наследование* - добавляем к существующему классу новые атрибуты и методы. Можно воспользоваться полями и методами родительских классов. Цепочки наследования. Множественное наследование.
- *Полиморфизм* - в классах-наследниках можно изменить методы родительского класса. По факту будет вызываться самый последний переопределенный метод в цепочке наследования.

Источники (рекомендую)

- Быстро и на пальцах:
  - tutorialspoint
- Медленно для начинающих:
  - Think Python by Downey
- Подробно:
  - Лутц. Изучаем Python
  - Саммерфилд. Программирование на Python 3
- Читать дальше патерны программирования:
  - Pyhon in Practice by Sammerfield
  - Гради Буч Объектно-ориентированный анализ и проектирование с примерами приложений

Инкапсуляция

В python НЕТ возможности полностью ограничить доступ к переменным объекта и класса

Если вы изучали другие ООП-языки, то можете считать, что все поля в питоне public, а все методы virtual.

@property - определяем get, set, del функции

Хочется read-only атрибуты.

Хочется, чтобы set методы для атрибута обязательно вызывались, запретить прямое присваивание obj.x = 7.

```
property([fget[, fset[, fdel[, doc]]]]) -> property
```

- *fget* : Функция, реализующая возврат значения свойства.
- *fset* : Функция, реализующая установку значения свойства.
- *fdel* : Функция, реализующая удаление значения свойства.
- *doc* : Строка документации для создаваемого свойства. Если не задано , будет использовано описание от fget (если оно существует).

Позволяет использовать методы в качестве свойств объектов — порождает дескриптор, позволяющий создавать «вычисляемые» свойства (тип property).

Пример использования в классическом виде:

```
class Mine(object):

    def __init__(self):
        self._x = None

    def get_x(self):
        return self._x

    def set_x(self, value):
        self._x = value

    def del_x(self):
        self._x = 'No more'

    x = property(get_x, set_x, del_x, 'Это свойство x.')

type(Mine.x)  # property
mine = Mine()
mine.x        # None
mine.x = 3
mine.x        # 3
del mine.x
mine.x        # No more
```

Используя функцию в качестве декоратора можно легко создавать вычисляемые свойства только для чтения:

```
class Mine(object):

    def __init__(self):
        self._x = 'some value'

    @property
    def prop(self):
        return self._x

mine = Mine()
mine.prop                  # some value
mine.prop = 'other value'  # AttributeError

del mine.prop              # AttributeError
```

Объект свойства также предоставляет методы getter, setter, deleter, которые можно использовать в качестве декораторов для указания функций реализующих получение, установку и удаление свойства соответственно. Следующий код эквивалентен коду из первого примера:

```
class Mine(object):

    def __init__(self):
        self._x = None

    x = property()

    @x.getter
    def x(self):
        """Это свойство x."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        self._x = 'No more'
```

__slots__

Можно запретить создание атрибута объекта используя __slots__

```
class Robot():
  __slots__ = ['a', '_b', '__c']
  def __init__(self):
    self.a = 123
    self._b = 123
    self.__c = 123

obj = Robot()
print(obj._Robot__c)    # 123 - все еще можем доступиться до атрибута по полному имени
obj.__c = 77            # УРА! AttributeError: 'Robot' object has no attribute '__c'
print(obj.a)
print(obj._b)
print(obj.__c)
```

Нельзя создать атрибут объекта, не перечисленный в __slots__

Переопределение __setattr__

```
class Robot(object):
   def __init__(self):
      self.a = 123
      self._b = 123

   def __setattr__(self, name, val):
      if name not in ('a', '_b'):
            raise AttributeError(name)
      super().__setattr__(name, val)

obj = Robot()
obj.a = 5
print(obj.a)
obj.__c = 77       # AttributeError
print(obj.__c)     # AttributeError
```

Аналогично можно переопределить другие функции доступа к атрибутам.