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 Community Edition.

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()**

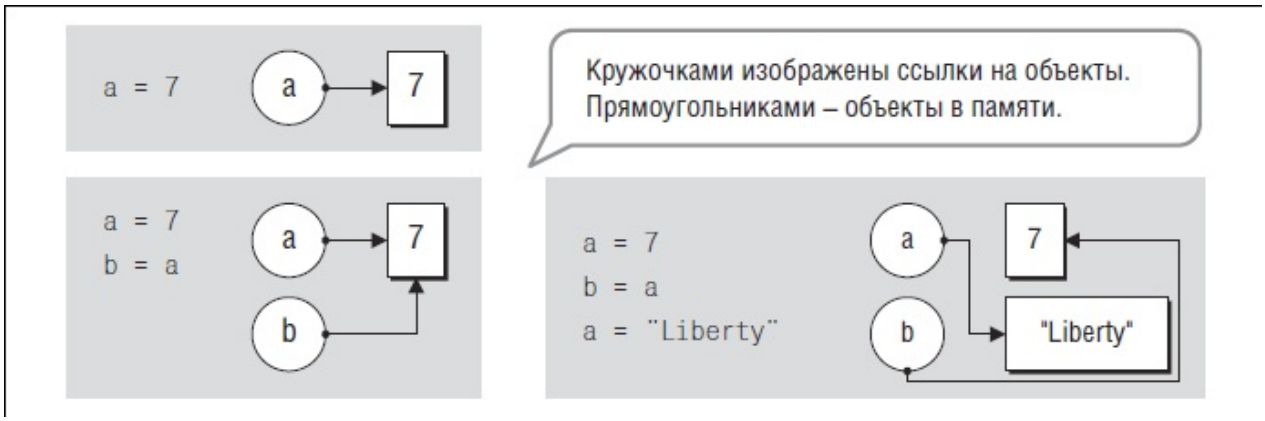
```
x = 5
print(x)          # 5
print(type(x))    # <class 'int'>

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

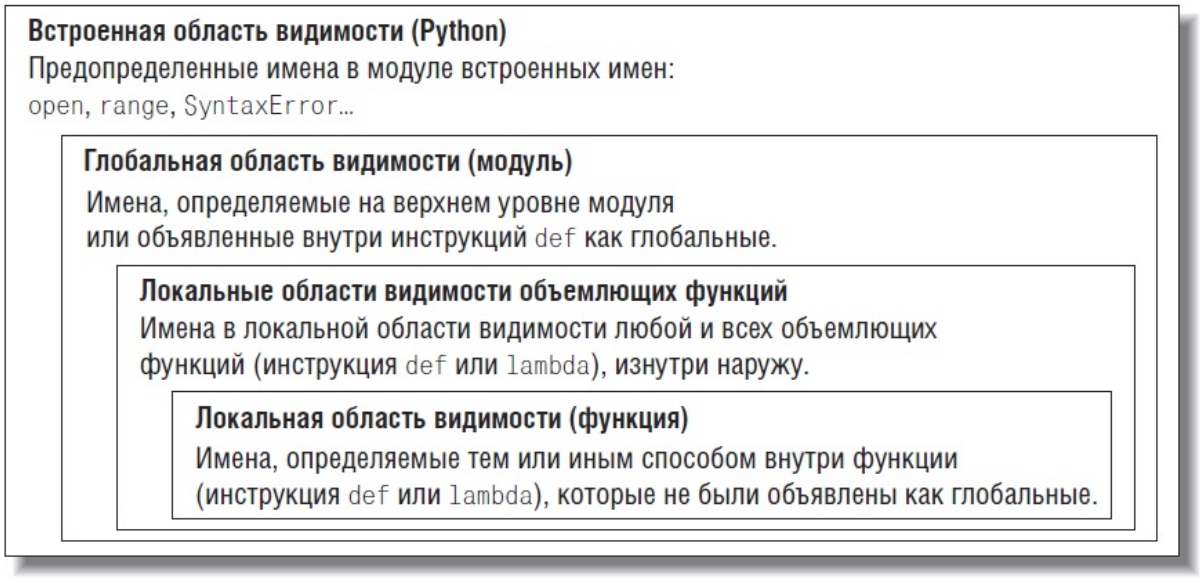
Для инструкции def:

- 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.

Функции

Лямбда-функции

Источники:

- Саммерфилд, Глава 4 Управляющие структуры и функции / Собственные функции / Лямбда-функции

Лямбда-функции - это функции, для создания которых используется следующий синтаксис:

```
lambda parametes: expression
```

parameters - не обязательная часть. Обычно позиционые переменные через запятую. Можно использовать полный синтаксис, который допустим в def.

expression :

- нельзя условные операторы или циклы (можно условные выражения);
- нельзя **return** или **yield**

Результатом лямбда-выражения является анонимная функция. При ее вызове вычисляется значение *expression* при указанных значениях параметров.

Если выражение *expression* - кортеж, то он должен быть заключен в круглые скобки ()

Пример: лямбда-функция, которая добавляет 's' если аргумент не 1 (окончание множественного числа).

```
s = lambda x: "" if x == 1 else "s"                    # Анонимная функция присваевае
тся переменной s.
print("{0} file{1} processed".format(count, s(count))   # использование этой функции
```

Точно не нужно return?

Не нужно. Эти две функции area (площадь треугольника по основаниию b и высоте h) вызываются одинаково:

Обычная функция:

```
def area(b, h):
    return 0.5 * b * h
```

Лямбда-функция (заметьте, return нет):

```
area = lambda b, h: 0.5 * b * h
```

Вызываем одинаково:

```
area(6, 5)
```

PEP-8 и лямбда-функции

Если код нужно использовать повторно (вызывать по имени), и функция пишется в одну строку, то лучше написать обычную функцию через def, а не использовать анонимную функцию (лямбду).

Плохо:

```
area = lambda b, h: 0.5 * b * h
```

Хорошо:

```
def area(b, h): return 0.5 * b * h
```

Пример лямбда-функции - сортировка по ключу

Пусть у нас есть список кортежей, которые описывают химические элементы (номер группы, порядковый номер, название).

```
>>> elements = [(2, 12, "Mg"), (1, 11, "Na"), (1, 3, "Li"), (2, 4, "Be")]
```

При сортировке по умолчанию получим:

```
>>> sorted(elements)
[(1, 3, 'Li'), (1, 11, 'Na'), (2, 4, 'Be'), (2, 12, 'Mg')]
```

Если мы не хотим учитывать при сортировке первый элемент кортежа (группу), то можно написать обычную функцию, которая вернет ключ. Но мы не хотим плодить много маленьких функций. Напишем лямбда-функцию при вызове sorted.

```
elements.sort(key=lambda e: (e[1], e[2]))   # не забываем про () если возвращаем кортеж

elements.sort(key=lambda e: e[1:3])         # или так
```

Отсортируем сначала по названию (без учета регистра), а потом по порядковому номеру:

```
elements.sort(key=lambda e: (e[2].lower(), e[1]))
```

Пример лямбда-функции - словари со значением по умолчанию

При обращении к несуществующему ключу будет создано значение по умолчанию. Лямбда-функция задает это значение.

```
minus_one_dict = collections.defaultdict(lambda: -1)
point_zero_dict = collections.defaultdict(lambda: (0, 0))
message_dict = collections.defaultdict(lambda: "No message available")
```

assert

TODO

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 123 и 21.

123 % 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

Последовательности

Последовательность - это такой тип данных, в котором есть:

- **in** - проверка входит элемент в последовательность или нет;
- **[ ]** - срезы
- **for** - перебор всех элементов последовательности.

Встроенные типы последовательностей в python:

- Строки
  - bytearray
  - bytes (НЕизменяемое)
  - str (НЕизменяемое)
- list (изменяемое)
- tuple (НЕизменяемое)

Методы неизменяемых последовательностей

Операция	Результат
x in s	True если хоть один элемент s равен x, иначе False
x not in s	False если хоть один элемент s равен x, иначе True
s + t	конкатенация s и t
s * n n * s	сложить s n раз
s[i]	элемент номер i
s[i:j]	срез от i до j [i,j)
s[i:j:k]	срез от i до j [i,j) с шагом k
len(s)	длина s
min(s)	минимальный элемент s
max(s)	максимальный элемент s
s.index(x) s.index(x,i) s.index(x,i,j)	номер первого вхождения x с индексом от i до j [i,j)
s.count(x)	сколько раз x встречается в s

Методы изменяемых последовательностей

Операция	Результат
s[i] = x	Элемент с номером i последовательности s заменить на x
s[i:j] = t	срез s от i до j заменить содержимым итерабельного (можно перебирать элементы) объекта t
del s[i:j]	s[i:j] = []
s[i:j:k] = t	срез s[i:j:k] заменить элементами из t
del s[i:j:k]	удалить элементы среза s[i:j:k] из последовательности
s.append(x)	добавить x в конец последовательности (s[len(s):len(s)] = [x])
s.clear()	удалить все элементы из s (del s[:])
s.copy()	создать shallow copy посделовательности s (s[:])
s.extend(t) s += t	добавить к последовательности s содержимое последовательности t (s[len(s):len(s)] = t)
s *= n	добавить в s его содержимое n раз
s.insert(i, x)	вставить x в s на место с индексом i (s[i:i] = [x])
s.pop() s.pop(i)	вернуть последний элемент (с номером i) и удалить его из последовательности
s.remove(x)	удалить первое вхождение x в s (если нет, ValueError exception)
s.reverse()	последовательность в обратном порядке

Перебираем элементы посделовательности

Перебираем только элементы:

```
for x in s:
    print(x)
```

Перебираем и элементы, и их номера:

```
for i, x in enumerate(s):
    print(i, x)
```

Строки

Строковые типы

- str (НЕизменяемое) - строки юникода
- bytes (НЕизменяемое) - для представления двоичных данных (в том числе кодированный текст)
- bytearray (изменяемый bytes) - последовательности 8-битных целых чисел. Однако они поддерживают большую часть строковых операций и при отображении выводятся как строки символов ASCII. Объекты этого типа обеспечивают возможность хранения больших объемов текста, который требуется изменять достаточно часто.

Как задаются строки

- 'в одинарных кавычках', "двойных кавычках", 'фильм "Титаник"' или "фильм 'Титаник'"
- тройные кавычки из ' или ":
  - многострочная строка; (html)
  - docstring
  - комментарий (лучше так не делать)
- 'C:\\new\\text.dat' - escape-последовательности \n, \t, \r
- r'C:\new\text.dat' - экранирует escape-последовательности (сделано, чтобы удобно было писать regexp)
- u'Unicode string' - строка юникода
- str(...) - строка

Методы строки

- сложение **'abc'+'xyzb**

```
>>> 'abc'+'xyz'
'abcxyz'
```

- умножение **'hi'*3**

```
>>> 'hi' * 3
'hihihi'
```

- <, >, <=, >=, ==, != - *побайтовое* сравнение строк
    - символы юникода могут быть представлены по-разному, например, символ 0x00C5 может быть представлен как [0xE2, 0x84, 0xAB], [0xC3, 0x85] и [0x41, 0xCC, 0x8A].

Используем **unicodedata.normalize(form='NFKD')** "Normalization Form Compatibility Decomposition" – нормализация в форме совместимой декомпозиции), то, передав ей строку, содержащую символ 0x00C5, представленный любой из допустимых последовательностей байтов, мы получим строку с символами в кодировке UTF-8, где интересующий нас символ всегда будет представлен последовательностью [0x41, 0xCC, 0x8A].

- порядок сортировки некоторых символов зависит от языка, в шведском a с двумя точками будет идти после z, а немецком он будет сортироваться, как если бы это была строка 'ae'
- подробнее о проблемах сортировки юникода см unicode.org/reports/tr10
- Срезы

```
>>>'Hello'[1]
e
>>>'Hello'[1:3]
el
>>>'Hello, world!'[1:10:2]
'el,wr'
```

- Проверка, что **строка пустая**

```
if s:
  print('Строка пустая')
else:
  print('Строка НЕ пустая')
```

Потому что:

```
>>> bool(s)
False
>>> bool('Hello')
True
```

- **in** - проверка, что в строке содержится подстрока

```
>>> 'el' in 'Hello'
True
```

- Проверки (строка непустая и *все* символы удовлетворяют критерию):
  - str.**isalnum**()
  - str.**isalpha**()
  - str.**isdecimal**()
  - str.**isdigit**()
  - str.**isidentifier**()
  - str.**islower**()
  - str.**isnumeric**()
  - str.**isprintable**()
  - str.**isspace**()
  - str.**istitle**()
  - str.**isupper**()
- Большие и маленькие буквы:
  - str.**capitalize**()
  - str.**casefold**()
  - str.**swapcase**()
  - str.**title**()
  - str.**upper**()
  - str.**lower**()
- Выравнивание:
  - str.**center**()
  - str.**ljust**()
  - str.**rjust**()
- Убрать символы (пробелы)
  - str.**lstrip**()
  - str.**rstrip**()
  - str.**strip**()

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
>>> comment_string = '#....... Section 3.2.1 Issue #32 .......'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

- Из табуляций в пробелы
  - str.**expandtabs**()

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
```

- Делаем строку
  - str.**zfill**() - заполняем строку нулями.

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

- Кодировка:
  - str.**encode**(encoding="utf-8")
- Шифрование:
  - str.**maketrans**()
  - str.**translate**(table)
- Форматирование:
  - str.**format**()
  - str.**format_map**()
- Проверки (подстрока в строке)
  - str.**count**(sub) - сколько раз входит (без пересечений)
  - **in** - поиск подстроки в строке
  - str.**endswith**(sub) - str оканчивается на sub
  - str.**startswith**(sub) -
- **Индекс** начала подстроки в строке, (иначе проверяем как 'el' in 'Hello')
  - str.**find**(sub[, start[, end]]) - возвращает -1, если подстроки нет
  - str.**index**(sub[, start[, end]]) - кидает ValueError, если подстроки нет
  - str.**rfind**(sub[, start[, end]])
  - str.**rindex**(sub[, start[, end]])
- Разделение и склейка
  - str.**partition**() - разделить на 3 части - до, разделитель, после
  - str.**rpartition**()
  - str.**rsplit**()
  - str.**split**(sep=None, maxsplit=-1) - разделить по sep на много частей

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
```

- str.**join**(iterable)

```
>>> a = ['hi', 'ha', 'ho']
>>> '-'.join(a)
'hi-ha-ho'
```

- str.**splitlines**()

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

- Поиск и замена:
  - str.**replace**(old, new, [count])

```
>>>'AAAAAA'.replace('AA', 'A')
'AAA'
>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> S.replace('SPAM', 'EGGS')    # Заменить все найденные подстроки
‘xxxxEGGSxxxxEGGSxxxx’
>>> S.replace('SPAM', 'EGGS', 1) # Заменить одну подстроку
'xxxxEGGSxxxxSPAMxxxx'
```

- Коды символов
  - **ord**(символ) - ASCII code символа
  - **chr**(ascii_code) - получить символ

```
>>> ord('a')
97
>>> chr(97)
'a'
```

Список (list)

Списки в Python - упорядоченные изменяемые коллекции объектов произвольных типов (почти как массив в других языках, но типы элементов могут отличаться).

Все методы неизменяемых и изменяемых последовательностей + метод **sort**.

Все методы списка (в ipython или тетради): **dir**(list) или **help**(list)

Help по 1 методу (например, append): list.append**?**

Создание списка

Пустой список:

```
a1 = []
a2 = list()
```

Список с данными:

```
a = ['apple', 'banana', 'wildberry']
b = [12, 34, -5, 16]
c = [12, 'apple', [3.14, 9.81], 'orange' ]
d = list('hello')
```

Индексы

Индексы начинаются с 0.

-1 - последний элемент.

```
   +---+---+---+---+---+---+
x  | P | y | t | h | o | n |        значение (число), value
   +---+---+---+---+---+---+
i    0   1   2   3   4   5   6=len  номер (index)
    -6  -5  -4  -3  -2  -1
```

```
a = list('python')
for i, x in enumerate(a):
    print(i, x)

print('length =', len(a))

# Output:
# Output:
# 0 p
# 1 y
# 2 t
# 3 h
# 4 o
# 5 n
# length = 6
```

Срез (slice) - часть списка

Можно обратиться к части массива от номера i (включая) до номера j (не включая). Математики запишут это как [i, j) Программисты запишут **срез** (slice) **a[i:j]** и сделают новый список.

```
>>> a = [3, 5, -2, 10, 8, 1, 17]
>>> a[3]
10
>>> a[1:4]
[5, -2, 10]
>>> a[-5:-2]
[-2, 10, 8]
```

У срезов есть (как у range) третий аргумент - шаг.

a[i:j:k] - взять срез списка a от i (включая) до j (НЕ включая) с шагом (прибавить) k.

```
>>> a = [3, 5, -2, 10, 8, 1, 17]
>>> a[1:6:2]
[5, 10, 1]
>>> a[-2:-5:-1]
[1, 8, 10]
```

Можно не писать номер начала или номер конца среза. Тогда это будет "начало" и "конец" списка.

```
>>> a = [3, 5, -2, 10, 8, 1, 17]
>>> a[:4]
[3, 5, -2, 10]
>>> a[3:]
[10, 8, 1, 17]
>>> a[:4:2]
[3, -2]
>>> a[3::2]
[10, 1]
>>> a[::2]
[3, -2, 8, 17]
>>> a[::-1]
[17, 1, 8, 10, -2, 5, 3]
```

Ссылки

В списке хранятся только ссылки на объекты.

```
m = [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
print(m)       # [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
m[0][0] = 10
print(m)       # [[10, 2, 3], [1, 2, 3], [1, 2, 3]]

a = [1, 2, 3]
m = [a, a, a]  # m содержит 3 ссылки на один и тот же список a
print(m)       # [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
m[0][0] = 10
print(m)       # [[10, 2, 3], [10, 2, 3], [10, 2, 3]]
```

Как изменить код последнего примера, чтобы для создания списка использовался все тот же список a, но изменение элемента m[0][0]=10 не изменяло другие элементы матрицы?

Копирование списка

Копировать список a можно:

- **list**(a)
- a**[:]**
- a.**copy()**

Shell copy, deep copy

- shell copy - копируем ссылки
- deep copy - идем по ссылкам рекурсивно и копируем данные (проблема с циклическими ссылками)

```
>>> a = [1, 2, 3]
>>>
>>> m1 = [a.copy(), a.copy(), a.copy()]
>>> m1
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
>>> m1[0][0] = 10
>>> m1
[[10, 2, 3], [1, 2, 3], [1, 2, 3]]
>>> m2 = m1.copy()                  # m2 содержит копии ссылок из m1, а не копии списк
ов
>>> m2
[[10, 2, 3], [1, 2, 3], [1, 2, 3]]
>>> m2[1][0] = 77                   # меняем список из m2
>>> m2
[[10, 2, 3], [77, 2, 3], [1, 2, 3]]
>>> m1                              # видим изменения в m1
[[10, 2, 3], [77, 2, 3], [1, 2, 3]]
>>> import copy
>>> m3 = copy.deepcopy(m1)          # копируем данные
>>> m3
[[10, 2, 3], [77, 2, 3], [1, 2, 3]]
>>> m3[2][0] = 666                  # меняем список в m3
>>> m3
[[10, 2, 3], [77, 2, 3], [666, 2, 3]]
>>> m2                              # изменения не затронули m2
[[10, 2, 3], [77, 2, 3], [1, 2, 3]]
>>> m1                              # изменения не затронули m1
[[10, 2, 3], [77, 2, 3], [1, 2, 3]]
```

Изменяем список

```
>>> [1, 2, 3] + [4, 5]
[1, 2, 3, 4, 5]
>>> [1, 2] * 3
[1, 2, 1, 2, 1, 2]
```

Оба примера сделают новый список.

Разница между append и extend

```
a = [1, 2, 3]
a.append([4, 5])
print(a)           # [1, 2, 3, [4, 5]] в списке 4 элемента, последний элемент - список
 [4, 5]

b = [1, 2, 3]
b.extend([4, 5])
print(b)           # [1, 2, 3, 4, 5] в списке 5 элементов

b2 = [1, 2, 3]
b2 += [4, 5]
print(b)           # [1, 2, 3, 4, 5] в списке 5 элементов
```

Все эти методы изменяют уже существующий список, а не создают новый.

Разница между append и +

Конкатенация (+) создает новый объект, а метод append нет. Поэтому append работает быстрее.

a.append(x) в конец работает как a[len(a):] = [x]

a[:0] = [x] - добавить в начало списка. Работают так же быстро, как append.

Изменение списка с помощью slice (удаление)

- **del** элемент
- **del** часть списка
- **del** переменная

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
print(a)        # [1, 66.25, 333, 333, 1234.5]

del a[2:4]
print(a)        # [1, 66.25, 1234.5]

del a[:]        # удалим ВСЕ ЭЛЕМЕНТЫ из списка a, (можно было так: a.clear() или a =
[])
print(a)        # []
```

Можно удалить переменную:

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a
print(a)    # ошибка! переменной a больше нет!
```

Изменение списка с помощью slice (удаление и вставка)

Удаление всего куска слева и вставка в это место куска справа:

```
>>> a = [1, 2, 3, 4, 5]
>>> a[2:4] = [10, 11, 12, 13]
>>> a
[1, 2, 10, 11, 12, 13, 5]
```

Заметьте, размеры удаленной и вставляемой части не обязаны совпадать.

Проверка принадлежности и сравнение

- **is** - сравнение ссылок на объекты
- **in** - элемент принадлежит списку
- **==**, **!=**, **<**, **>**, **<=**, **>=** - поэлементное сравнение списков (возможно рекурсивное). Подробнее в сортировках

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
>>> a == b
True
>>> 3 in a
True
>>> 88 in a
False
```

Проверить, что список пустой

`bool(a)` возвращает False, если a пустой список. Иначе он возвращает True.

```
a = []
if a:
    print('NON-empty list')
else:
    print('Empty list')
```

```
if not a:   # поверка, что лист ПУСТОЙ
```

split и join

См. строки

List comprehensions

Общий вид как сделать список:

```
[выражение for переменная in последовательность]
или
[выражение for переменная in последовательность if условие]
```

Как сделать список, написав меньше кода?

Привычный вариант:

```
a = []
for x in range(5):
     a.append(x**2)
```

Если функция достаточно сложная, можно ее написать отдельно:

```
def sqr(x):
    return x**2
a = list(map(sqr, range(5)))
```

А если простая, то записать через lambda:

```
a = list(map(lambda x: x**2, range(10)))
```

Или, как большинство программистов на питоне, использовать list comprehensions

```
a = [x**2 for x in range(5)]       # [0, 1, 4, 9, 16]
```

Примеры:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

то же самое, что:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Обратите внимание на порядок for и if.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> [x*2 for x in vec]                # create a new list with the values doubled
[-8, -4, 0, 4, 8]

>>> [x for x in vec if x >= 0]        # filter the list to exclude negative numbers
[0, 2, 4]

>>> [abs(x) for x in vec]             # apply a function to all the elements
[4, 2, 0, 2, 4]
                                      # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']

>>> [(x, x**2) for x in range(6)]     # create a list of 2-tuples like (number, square)

[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax

>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]   # flatten a list using a listcomp with two
 'for'
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Nested List Comprehensions (генерация вложенных списков)

Дана матрица

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

Надо ее транспонировать.

```
t = [[row[i] for row in m] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

чуть подробнее:

```
transposed = []
for i in range(4):
    transposed.append([row[i] for row in matrix])
```

еще подробнее:

```
transposed = []
for i in range(4):
    # the following 3 lines implement the nested listcomp
    transposed_row = []
    for row in matrix:
        transposed_row.append(row[i])
    transposed.append(transposed_row)
```

или через функцию zip()

```
list(zip(*matrix))
```

Не пилите сук, на котором сидите

Не изменяйте последовательность, которую перебираете в цикле.

Как напечатать список

Дан список `a = [3, 5, -2, 10]`. Надо его напечатать.

Как пишут в программе:

```
print(a)        # [3, 5, -2, 10]
```

В столбик:

```
for x in a:
    print(x)
# Output:
# 3
# 5
# -2
# 10
```

В строку, используем аргумент **end** функции print:

```
for x in a:
    print(x, end=' ')  # печатаем через пробел
print()                # в конце 1 раз - новая строка
```

В строку, используя оператор * (развернуть список в его элементы):

```
a = [3, 5, -1, 10]
print(a)              # [3, 5, -1, 10]
print(*a)             # 3 5 -2 10
```

В строку через разделитель, используя join:

```
a = ['abc', 'qw', 'xyz']
s = ', '.join(a)            # abc, qw, xyz
```

В этом случае все элементы списка должны быть строками (требует join). Если это не так, сделаем их str.

```
a = [1, 3.4, 'abc', [12, 34]]
s = '-'.join(map(str, a))       # 1-3.4-abc-[12, 34]
```

Изменяемый объект в виде значения по умолчанию

Мы хотим в функцию передать список. Если список не задали, создать пустой список.

```
def func(x = 0, a = []):
    a.append(x)
    print(a)
```

Не пишите изменяемые объекты в значения по умолчанию.

Почему x = 0 - хорошо, a = [] - плохо?

Потому что эти объекты создаются 1 раз за все время работы программы перед первым вызовом функции.

(Для знающих языки C, C++ или Java - это static объекты).

Каждый новый вызов функции может изменить этот объект.

```
>>> def foo(x = 0, a = []):
...     a.append(x)
...     print(a)
...
>>> foo(3, [5, 7, 11])  # ок, пока не используем значение по умолчанию
[5, 7, 11, 3]
>>> foo(-4, [1, 2])
[1, 2, -4]

>>> foo(1)              # в список по умолчанию добавили 1 - ок
[1]
>>> foo(5)              # в ТОТ ЖЕ список по умолчанию (он уже [1], а не []), добавили
 5
[1, 5]
```

Что делать? Как исправить?

Значениями по умолчанию делайте только неизменяемые объекты.

```
>>> def foo2(x = 0, a = None):
...     b = a or []
...     b.append(x)
...     print(b)
...
>>> foo2(1)
[1]
>>> foo2(5)
[5]
```

`None` - неизменяемый объект.

`a or []` работает так:

- если a - непустой список, то он в выражении будет True и выражение с or дальше вычисляться не будет, значение `a or []` равно `a`.
- если `a` это `None`, то `None` в логическом выражении будет False и выражение будет выполняться дальше, значение `a or []` равно `[]`

Кортеж (tuple)

Кортеж - неизменяемый список.

Создание кортежа

Пустой кортеж:

```
a1 = ()
a2 = tuple()
```

Непустой кортеж:

```
a = (7, 3.4, 'abc', [1, 2, 3])
b = 7, 3.4, 'abc', [1, 2, 3]    # скобки не обязательны
```

Распаковка кортежа

кортеж = последовательность происходит распаковка элементов последовательности в кортеж

```
x, y = map(int, input().split())
```

или

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

Множество (set)

- **set** - множество. В нем нет одинаковых элементов. Его можно изменять.
- **frozenset** - множество. В нем нет одинаковых элементов. Его НЕЛЬЗЯ изменять.

Порядок перебора элементов множества не определен.

То есть в множестве определены операции **in**, **len**, возможен перебор **for .. in**, но нельзя взять i-тый элемент.

Можно взять `enumerate(a)`, но порядок перебора может быть разный.

Создание множества

Множество пишем в { }.

Пустое множество:

```
a = set()   # только так
a = {}      # это НЕ множество, это словарь
```

Создаем множество:

```
a = [1, 5, 17, 5, -22, 4, 4, 4]   # сделали list
print(a)                          # 1 5 17 5 -22 4 4 4
b = set(a)                        # сделали set из list
print(b)                          # 1 5 17 -22 4
c = {1, 5, 17, 5, -22, 4, 4, 4}   # сделали set
print(c)                          # 1 5 17 -22 4
```

Что может быть элементом множества?

Только хешируемый объект. Т.е. объект, для которого определена функция **__hash__()** - она возвращает все время одно и то же значение на протяжении всей жизни объекта. Эти объекты можно сравнить на равенство, используя **__eq__()**

- int, float, str, tuple, frozenset - хешируемые объекты, могут быть добавлены в множество;
- list, dict, set - нехэшируемые, так как значение хеша зависит от значений их

элементов (а они могут изменяться).

Хешируемый != неизменный. Объект класса Person может быть хешируемым, если в классе определена фукнция __hash__(). Она может возвращать, например, СНИЛС человека или любой другой его неизменяемый идентификатор. При этом данные о человеке могут меняться (например, смена фамилии).

Методы множества

```
a = {1, 3, 5, 11, 12, 13}
b = {5, 1, 3, 21, 22, 23}
c = {1, 3, 5}
d = {3, 1, 5}
w = {2, 8}
```

Операция	Значение	Результат
a.add(7)	добавить 7 в множество a	
a.remove(5)	удалить 5 из a (если нет, KeyError exception)	{1, 3, 11, 12, 13}
a.discard(5)	удалить 5 из a если элемент есть	{1, 3, 11, 12, 13}
a.pop()	удаляет случайный элемент из множества (или KeyError, если множество пустое)	
a.clear()	удалить все элементы из a	
a.copy()	копия множества	
len(a)	число элементов в множестве	6
x in a	элемент x в множестве a	
x not in a	элемент x НЕ в множестве a	
a.**isdisjoint**(w)	True, если у множеств нет общих элементов	
c == d	множества c и d равны	
c < a	c содержится в множестве a, но не равно ему	
c.issubset(a) c <= a	c содержится в множестве a	
a > c	a содержит в себе c	
a.issuperset(c)	a содержит в себе c, но не равно	

a >= c	ему	
a.union(b) a | b	объединение множеств (новое)	{1, 3, 5, 11, 12, 13, 21, 22, 23}
a.update(b) a |= b	объединение множеств (изменяется a)	{1, 3, 5, 11, 12, 13, 21, 22, 23}
a.intersection(b) a & b	пересечение множеств (новое)	{1, 3, 5}
a.intersection_update(b) a &= b	оставляет во множестве a пересечение множеств a и b	{1, 3, 5}
a.difference(c) a - c	вычитание (новое множество)	{11, 12, 13}
a.difference_update(c) a -= c	удаляет из множества a все элементы, которые присутствуют в c	a = {11, 12, 13}
a.symmetric_difference(b) a ^ b	XOR (новое множество)	{11, 12, 13, 21, 22, 23}
a.symmetric_difference_update(b) a ^= b	XOR (изменяем a)	{11, 12, 13, 21, 22, 23}

Применение множеств:

- убрать повторения;
- список уже обработанных имен и "уже обрабатывали?"

Если нет аргументов командной строки или указаны -h или --help:

```
if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}
```

Словарь (dict)

Зачем нужны словари?

Если есть список дней недели, то мы по *номеру* дня недели быстро получаем название дня недели.

Хотим решить обратную задачу - по строке названия дня недели получать его номер. Хотим решать быстрее, чем index.

Нужно получить пары "название дня недели" - "номер дня недели".

Термины

Словарь - неупорядоченная коллекция пар ключ-значение.

Неупорядоченная - значит порядок перебора не определен, нет понятие i-того элемента или среза.

Ключ - **хешируемый** объект. Ключи **Уникальные**.

Значение - любой объект.

Создание словаря

Пустой словарь:

```
d1 = dict()
d2 = {}     # это словарь, а не множество
```

Непустой словарь: **dict(d)** - shallow copy словаря d.

```
d1 = dict({"id": 1948, "name": "Washer", "size": 3})        # литерал словаря
d2 = dict(id=1948, name="Washer", size=3)                   # именованные аргументы
d3 = dict([("id", 1948), ("name", "Washer"), ("size", 3)])  # из последовательности
d4 = dict(zip(("id", "name", "size"), (1948, "Washer", 3))) # из последовательности
d5 = {"id": 1948, "name": "Washer", "size": 3}              # из литерала словаря
```

Словарь страна-столица:

```
capitals = {'Russia': 'Moscow', 'Ukraine': 'Kiev', 'USA': 'Washington', 'Myanmar':'Nay
pyidaw', 'Mongolia':'Ulaanbaatar', 'China':'Beijing'}
capitals = dict(Russia = 'Moscow', Ukraine = 'Kiev', USA = 'Washington', )
capitals = dict([("Russia", "Moscow"), ("Ukraine", "Kiev"), ("USA", "Washington")])
capitals = dict(zip(["Russia", "Ukraine", "USA"], ["Moscow", "Kiev", "Washington"]))
```

Пишем красиво:

```
capitals = {
    'Russia': 'Moscow',
    'Ukraine': 'Kiev',
    'USA': 'Washington',
    'Myanmar':'Naypyidaw',
    'Mongolia':'Ulaanbaatar',
    'China':'Beijing'
}
```

d.fromkeys(s, v) - Возвращает словарь типа dict, ключами которого являются элементы последовательности s, а значениями либо None, либо v, если аргумент v определен.

```
d = {}.fromkeys ("ABCD", 3) # d == {'A': 3, 'B': 3, 'C': 3, 'D': 3}
```

dict comprehensions

```
cities = ["Moscow", "Kiev", "Washington"]
states = ["Russia", "Ukraine", "USA"]
capitalsOfState = {state: city for city, state in zip(cties, states)}
```

Сделаем словарь квадратов натуральных чисел:

```
square1 = {x : x*x for x in range(10) }                              # dict comprehensi
ons
square2 = {0=0, 1=1, 2=4, 3=9, 4=16, 5=25, 6=36, 7=49, 8=64, 9=81}  # задаем явно пары
 ключ=значение
```

Выворачиваем словарь "наоборот":

```
>>> StateByCapital = {CapitalsOfState[state]: state for state in CapitalsOfState}
>>> stateByCapital
{'Kiev': 'Ukraine', 'Moscow': 'Russia', 'Washington': 'USA'}
```

и квадраты:

```
sqrts = {square1[x]:x for x in square1}
```

Этот код будет работать чуть-чуть быстрее:

```
sqrts2 = {v:k for k,v in square1.items()}
```

if тоже поддерживается:

```
file_sizes = {name: os.path.getsize(name) for name in os.listdir(".") if os.path.isfil
e(name)}
```

Методы словаря

Операция	Значение
value = A[key]	Получение элемента по ключу. Если элемента с заданным ключом в словаре нет, то возникает исключение KeyError
value = A.get(key)	Получение элемента по ключу. Если элемента в словаре нет, то get возвращает None.
value = A.get(key, *default_value*)	То же, но вместо None метод get возвращает default_value.
key in A	Проверить принадлежность ключа словарю.
key not in A	То же, что not key in A.
A[key] = value	Добавление нового элемента в словарь или изменяет старое значение на value
len(A)	Возвращает количество пар ключ-значение, хранящихся в словаре.
A.keys()	Возвращает список ключей
A.values()	Возвращает список значений (порядок в нем такой же, как для списка ключей)
A.items()	Возвращает список пар ключ, значение

```
d = {}.fromkeys ("ABCD", 3) # d == {'A': 3, 'B': 3, 'C': 3, 'D': 3}
s = set("ACX")              # s == {'A', 'C', 'X'}
matches = d.keys() & s      # matches == {'A', 'C'}
```

Методы **keys(), values(), items()** возвращают *представления* словарей.

Представление - это итерируемый объект +

- если представляемый объект изменяется, представление будет отражать эти изменения;
- поддерживает операции над множествами;

Пусть v - представление словаря, а x - множество или представление словаря. Определены операции:

```
v & x # Пересечение
v | x # Объединение
v - x # Разность
v ^ x # XOR
```

Операция	Значение
del A[key]	Удаление пары ключ-значение с ключом key. Возбуждает исключение KeyError, если такого ключа нет.
value = A.pop(key)	Удаление пары ключ-значение с ключом key и возврат значения удаляемого элемента. Если такого ключа нет, то возбуждается KeyError.
value = A.pop(key, default_value)	То же, но вместо генерации исключения возвращается default_value.
A.pop(key, None)	Это позволяет проще всего организовать безопасное удаление элемента из словаря.

Удаление ключа с проверкой

```
if key in A:
    del A[key]
```

Удаление ключа с перехватом исключения

```
try:
    del A[key]
except KeyError:
    pass
```

Перебор словаря

При переборе словаря порядок ключей и пар может быть любой. Порядок может измениться со временем (а может остаться прежним).

```
In [12]: capital = {'Russia': 'Moscow', 'Ukraine': 'Kiev', 'USA': 'Washington',
    ...:  'Myanmar':'Naypyidaw', 'Mongolia':'Ulaanbaatar', 'China':'Beijing'}

In [13]: capital
Out[13]:
{'China': 'Beijing',
 'Mongolia': 'Ulaanbaatar',
 'Myanmar': 'Naypyidaw',
 'Russia': 'Moscow',
 'USA': 'Washington',
 'Ukraine': 'Kiev'}
```

по ключам (перебор словаря идет неявно по ключам):

```
In [14]: for s in capital:
    ...:     print(s, capital[s])
    ...:
China Beijing
Mongolia Ulaanbaatar
Ukraine Kiev
Russia Moscow
USA Washington
Myanmar Naypyidaw
```

явно по ключам:

```
In [15]: for s in capital.keys():
    ...:     print(s, capital[s])
China Beijing
Mongolia Ulaanbaatar
Ukraine Kiev
Russia Moscow
USA Washington
Myanmar Naypyidaw
```

отсортировать ключи:

```
In [18]: for s in sorted(capital.keys()):
    ...:     print(s, capital[s])
China Beijing
Mongolia Ulaanbaatar
Myanmar Naypyidaw
Russia Moscow
USA Washington
Ukraine Kiev
```

только значения:

```
In [19]: for c in capital.values():
    ...:     print(c)
    ...:
Beijing
Ulaanbaatar
Kiev
Moscow
Washington
Naypyidaw
```

Значения тоже можно отсортировать.

Если нам в цикле будет нужен и ключ, и значение, лучше перебирать по парам:

```
In [20]: for s, c in capital.items():
    ...:     print(s, c)
China Beijing
Mongolia Ulaanbaatar
Ukraine Kiev
Russia Moscow
USA Washington
Myanmar Naypyidaw
```

Пары тоже можно отсортировать:

```
In [21]: for s, c in sorted(capital.items()):
    ...:     print(s, c)
China Beijing
Mongolia Ulaanbaatar
Myanmar Naypyidaw
Russia Moscow
USA Washington
Ukraine Kiev
```

В обратном порядке reverse=True:

```
In [22]: for s, c in sorted(capital.items(), reverse=True):
    ...:     print(s, c)
Ukraine Kiev
USA Washington
Russia Moscow
Myanmar Naypyidaw
Mongolia Ulaanbaatar
China Beijing
```

Сортировка

Источники

- Google for Education / Python Course / Sorting
- Sorting HowTo

Сравнение

Чтобы отсортировать объекты, для них нужно знать какой из них больше, какой меньше.

Числа сравнивают как числа:

```
3 > 1
-3.5 < 5.24
-1 < 1.5
```

Строки сравнивают, как слова в словаре. Какое слово в словаре идет раньше, то и меньше.

```
'Abc' < 'abc'
'ABC' < 'C' < 'Pascal' < 'Python'
```

То есть сравнивают по символам, начиная с первого.

Списки (list) и кортежи (tuple) сравнивают по элементам.

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Если сравнивать типы, которые не сравниваются, получается исключение `TypeError`.

Функции sort и sorted

Для сортировки используют стандартные функции sort (сортировка списка) и sorted (сортировка последовательности)

- **список.sort(*, key=None, reverse=False)** - стабильная сортировка самого списка
- **sorted(iterable, *, key=None, reverse=False)** - создается новый отсортированный список, старый остается без изменения

По умолчанию используются стандартные операторы сравнения и сортирует по возрастанию.

```
a = [3, 6, 8, 2, 78, 1, 23, 45, 9]

b = sorted(a)
print(a)       # [3, 6, 8, 2, 78, 1, 23, 45, 9]
print(b)       # [1, 2, 3, 6, 8, 9, 23, 45, 78]

a.sort()
print(a)       # [1, 2, 3, 6, 8, 9, 23, 45, 78]
```

reverse=True - в обратном порядке

```
>>> sorted(a, reverse=True)
[78, 45, 23, 9, 8, 6, 3, 2, 1]
>>> a.sort(reverse=True)
>>> a
[78, 45, 23, 9, 8, 6, 3, 2, 1]
```

key=функция - как сравнивать объекты

Отсортируем список по значению модуля каждого числа. Модуль берется функцией abs()

```
a = [3, 6, -8, 2, -78, 1, 23, -45, 9]

b = sorted(a, key=abs)
print(a)       # [3, 6, -8, 2, -78, 1, 23, -45, 9]
print(b)       # [1, 2, 3, 6, -8, 9, 23, -45, -78]

a.sort(key=abs)
print(a)       # [1, 2, 3, 6, -8, 9, 23, -45, -78]
```

Другой пример. Отсортируем строки по длине.

```
strs = ['ccc', 'aaaa', 'd', 'bb']
print sorted(strs, key=len)         # ['d', 'bb', 'ccc', 'aaaa']
```

Ключевая функция (имя которой передается в аргументе key) должна принимать 1 значение (value) и возвращать 1 значение (proxy value). По этому proxy value проходит сортировка.

Отсортируем список строк БЕЗ учета регистра. Для этого будем сортировать строки, которые приведены к нижнему регистру.

```
strs = ['aa', 'BB', 'zz', 'CC']
print sorted(strs)                  # ['BB', 'CC', 'aa', 'zz'] (case sensitive)
print sorted(strs, key=str.lower)   # ['aa', 'BB', 'CC', 'zz']
```

Как написать функцию для сравнения

Пишем функцию, которая для 1 объекта (аргумента) возвращает proxy value, по которым этот объект будут сравнивать в сортировке

```
# Say we have a list of strings we want to sort by the last letter of the string.
strs = ['xc', 'zb', 'yd' ,'wa']

# Write a little function that takes a string, and returns its last letter.
# This will be the key function (takes in 1 value, returns 1 value).
def MyFn(s):
    return s[-1]

# Now pass key=MyFn to sorted() to sort by the last letter:
print sorted(strs, key=MyFn)  ## ['wa', 'zb', 'xc', 'yd']
```

Еще пример: отсортируем разнотипные числа как числа.

```
sorted(["1.3", 7.5, "5", 4, "2.4", 1], key=float)
```

Как написать функцию для сравнения

Пишем функцию, которая для 1 объекта (аргумента) возвращает proxy value, по которым этот объект будут сравнивать в сортировке

Пример: сортируем строки по последним буквам

```
# Say we have a list of strings we want to sort by the last letter of the string.
strs = ['xc', 'zb', 'yd' ,'wa']

# Write a little function that takes a string, and returns its last letter.
# This will be the key function (takes in 1 value, returns 1 value).
def MyFn(s):
    return s[-1]

# Now pass key=MyFn to sorted() to sort by the last letter:
print sorted(strs, key=MyFn)  ## ['wa', 'zb', 'xc', 'yd']
```

Пример: Сортируем по росту и весу

Даны рост (см) и вес (кг) каждого человека. По одному человеку на строку. Отсортируем людей.

```
a = [(166, 55.2), (157, 55.2), (170, 55.2), (175, 90), (166, 73), (180, 73)]
print(a)

b = sorted(a)
print(b)
# [(157, 55.2), (166, 55.2), (166, 73), (170, 55.2), (175, 90), (180, 73)]
```

Получилась сортировка по возрастанию роста, при одинаковом росте сортируем по весу.

Теперь напишем функцию, которая будет смотреть только на вес и игнорировать рост.

```
def weight(t):
    h = t[0]
    w = t[1]
    return w  # или return t[1]

b = sorted(a, key=weight)
print(b)
# # [(166, 55.2), (157, 55.2), (170, 55.2), (166, 73), (180, 73), (175, 90)]
```

Заметим, что люди с одинаковым весом идут в том же порядке, что и в списке до сортировки: при весе 55.2 сначала 166, потом 157. При весе 73 сначала 166, потом 180.

Сортировка в питоне **стабильная**, то есть равные по одному признаку элементы будут идти в том же порядке, что и до сортировки.

Отсортируем по весу *по убыванию*. Не будем использовать `reverse = True`, сделаем это через функцию.

```
def weight_decr(t):
    h = t[0]
    w = t[1]
    return -w  # или return -t[1]

b = sorted(a, key=weight_decr)
print(b)
# [(175, 90), (166, 73), (180, 73), (166, 55.2), (157, 55.2), (170, 55.2)]
```

Отсортируем по весу (по возрастанию), а при равном весе - по росту (по возрастанию).

Для этого в функции, которая возвращает proxy values вернем (вес, рост).

```
def wh1(t):
    h = t[0]
    w = t[1]
    return (w, h)  # обязательно возвратить tuple

b = sorted(a, key=wh1)
print(b)
# [(157, 55.2), (166, 55.2), (170, 55.2), (166, 73), (180, 73), (175, 90)]
```

Как отсортировать по *возрастанию* веса и при равном весе по_убыванию роста?

```
def wh2(t):
    h = t[0]
    w = t[1]
    return (w, -h)  # обязательно возвратить tuple

b = sorted(a, key=wh2)
print(b)
# [(170, 55.2), (166, 55.2), (157, 55.2), (180, 73), (166, 73), (175, 90)]
```

То же самое через lambda-функцию:

```
b = sorted(a, key=lambda t: (t[1], -t[0]))
print(b)
# [(170, 55.2), (166, 55.2), (157, 55.2), (180, 73), (166, 73), (175, 90)]
```

Дополнительные материалы

Как обеспечить сравниение объектов классов

Об этом будет рассказано позже, когда будем говорить о классах и объектах.

Алгоритмы сортировки

Саммерфильд, стр 172.

В языке Python реализован адаптивный алгоритм устойчивой сортировки со слиянием, который отличается высокой скоростью и интеллектуальностью и особенно хорошо подходит для сортировки частично отсортированных списков, что встречается достаточно часто.

«адаптивный» - алгоритм сортировки адаптируется под определенные условия, например, учитывает наличие частичной сортировки данных.

«устойчивый» - одинаковые элементы не перемещаются относительно друг друга (в конце концов, в этом нет никакой необходимости)

«сортировка со слиянием» – это общее название используемых алгоритмов сортировки.

Алгоритм был создан Тимом Петерсом (Tim Peters). Интересное описание и обсуждение алгоритма можно найти в файле listsort.txt, который поставляется в составе исходных программных кодов Python.

Сравнение и сортировка в python 3.x

Лутц, стр 261

Сравнивание и сортировка в Python 3.0: В Python 2.6 и в более ранних версиях сравнивание выполняется по-разному для объектов разных типов (например, списков и строк) – язык задает способ упорядочения различных типов, который можно признать скорее детерминистским, чем эстетичным. Этот способ упорядочения основан на именах типов, вовлеченных в операцию сравнения, например любые целые числа всегда меньше любых строк, потому что строка "int" меньше, чем строка "str".

При выполнении операции сравнения никогда не выполняется преобразование типов объектов, за исключением сравнивания объектов числовых типов.

В Python 3.0 такой порядок был изменен: попытки сравнивания объектов различных типов возбуждают исключение – вместо сравнивания по названиям типов. Так как метод сортировки использует операцию сравнения, это означает, что инструкция [1, 2,

'spam'].sort() будет успешно выполнена в Python 2.x, но возбудит исключение в версии Python 3.0 и выше. Кроме того, в версии Python 3.0 больше не поддерживается возможность передачи методу sort произвольной функции сравнения, для реализации иного способа упорядочения. Чтобы обойти это ограничение, предлагается использовать именованный аргумент key=func, в котором предусматривать возможность трансформации значений в процессе сортировки, и применять именованный аргумент reverse=True для сортировки по убыванию. То есть фактически выполнять те же действия, которые раньше выполнялись внутри функции сравнения.

Задачи

Срезы списка

На одной строке через пробел даны целые числа. Напечатайте (1 ответ в 1 строку следующие мини-задачи).

Пример печати результата для введеного списка a = [10, 7, -6, 11, 13, 5, 1, 8, 13]

Задача	Результат
Напечатать список	[10, 7, -6, 11, 13, 5, 1, 8, 13]
первое число списка	10
последнее число списка	13
первые пять чисел списка	[10, 7, -6, 11, 13]
весь список, кроме последних двух чисел	[10, 7, -6, 11, 13, 5, 1]
все числа с четными номерами (считая, что индексация начинается с 0)	[10, -6, 13, 1, 13]
все числа с нечетными номерами	[7, 11, 5, 8]
все числа в обратном порядке	[13, 8, 1, 5, 13, 11, -6, 7, 10]
все числа строки через один в обратном порядке, начиная с последнего	[13, 1, 13, -6, 10]
длину списка	9
Заменить 2 первых числа на 1 22 333	[1, 22, 333, -6, 11, 13, 5, 1, 8, 13]

Расширение файла

Даны имена файлов (путь) по 1 файлу на строку. Напечатать все расширения файлов (после последней точки, без пробельных символов). Можно не вводить имена файлов, а создать список строк и отлаживать код на нем.

```
Input:
/cygdrive/c/Users/taty/GitBook/Library/tatyderb/python-express-course/chapter_seq/READ
ME.md
summary.html
../keys.png
C:\tmp\example.py
# Output:
md
html
png
py
```

Убрать повторы

Даны числа на одной строке через пробел. Напечатайте каждое число только 1 раз. Порядок печати - произвольный.

```
# Input:
5 3 4 -1 -2 5 7 3
# Output: (порядок печати может отличаться)
-1 -2 3 4 5 7
```

Только один раз

Даны числа на одной строке через пробел. Напечатайте каждое число только 1 раз. Порядок печати - в том, в котором числа первый раз встретились в последовательности.

```
# Input:
5 3 4 -1 -2 5 7 3
# Output: (строго такой порядок печати)
5 3 4 -1 -2 7
```

Уникальные числа

Даны числа на одной строке через пробел. Только те числа, которые были уникальными в исходной последовательности. Порядок печати любой.

```
# Input:
5 3 4 -1 -2 5 7 3
# Output:
4 -1 -2 7
```

Страны и города

Дан список городов по странам в формате страна и города через пробел.

Выведите список город страны где есть города с таким именем. Список отсортировать.

```
Input:
Russia Moscow Samara Peterburg Omsk
Ukraina Kiev Kharkov Nezhin
USA NewYork Peterburg Dallas Austin Houston
Output:
Austin USA
Dallas USA
Houston USA
Kiev Ukraina
Kharkov Ukraina
Moscow Russia
Nezhin Ukraina
NewYork USA
Omsk Russia
Peterburg Russia USA
Samara Russia
```

Даты

На каждой строке написана дата в формате dd/mm/yyyy. Выведите даты в порядке возрастания в формате yyyy/mm/dd.

```
Input:
01/12/1910
31/05/0861
22/06/2014

Output:
0861/05/31
1910/12/01
2014/06/22
```

Рост, вес

Даны рос и вес каждого человека. Отсортировать по уменьшению роста. При одинаковом росте сначала печатать больший вес.

```
Input:
156 66.2
178 66.8
178 56.4

Output:
178 66.8
178 56.4
156 66.2
```

Скобки-1

Дана скобочная последовательность из скобок (и) на одной строке. Напечатайте YES, если скобочная последовательность правильная. Иначе напечатайте NO.

Правильная скобочная последовательность:

- ()
- (()())
- (()) Неправильная скобочная последовательность:
- (()
- ())
-)(

Скобки-2

Дана скобочная последовательность из скобок разных типов (){}[]<> на одной строке. Напечатайте YES, если скобочная последовательность правильная. Иначе напечатайте NO.

Правильная скобочная последовательность:

- ()
- (()<>)
- ({})

Неправильная скобочная последовательность:

- (()
- ())
-)(
- (<)>

Источники

- python3 tutorial
- pep-328 Absolute and Relative Imports
-
-

Запуск

Для каждого файла определена переменная __name__, в которой определяется как этот файл выполняется - как подгружаемый модуль или как отдельная программа.

Пусть у нас есть файл fibo.py с функцией fib, который печатает свою переменную __name__

Импортируем его в интерактивном интерпретаторе:

```
>>> import fibo
>>> fibo.__name__
'fibo'
```

Запустим его как отдельную программу:

```
$python fibo.py
__main__
```

То есть если хочется выполнять код только в режиме программы, то пишем:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

Модули - введение

Модуль - это 1 файл на питоне или другом языке (расширение на C, C++, Java и тп).

Зачем?

- повторное использование программного кода;
- свое пространство имен;
- модульность программного кода (наружу API, внутренность можно рефакторить).

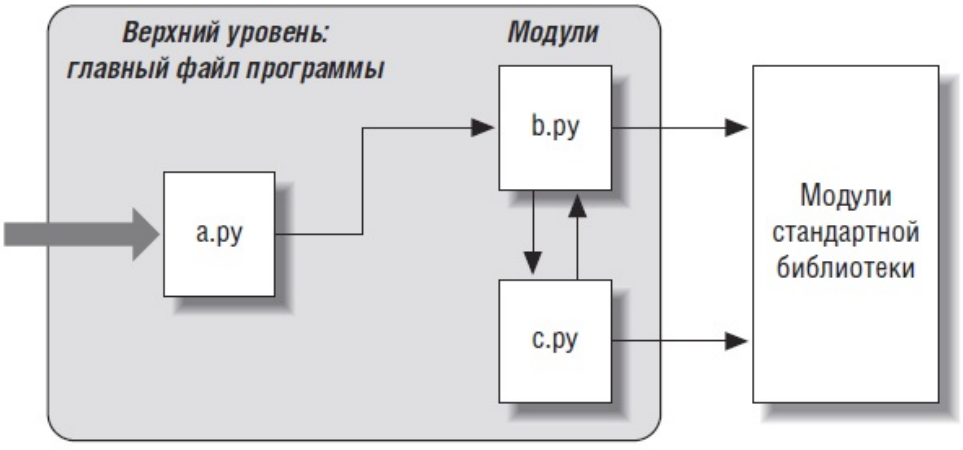
Как организована программа на языке Python

Предполагаем, что программа состоит из нескольких файлов и сторонних библиотек.

Архитектура программы = главный файл (сценарий) + дополнительные файлы (модули, которые подключаются к главному).

Обычно файлы модулей ничего не делают, если запустить их отдельно (или выполняют тесты данного модуля).

Пусть программа состоит из файлов a.py, b.py, c.py. Арихтектура программы:



`a.py` - текстовый файл с инструкциями, выполняются от начала до конца;

`b.py`, `c.py` - текстовые файлы с инструкциями, содержат переменные и функции (и быть может код), не выполняются самостоятельно.

b.py:

```
def spam(text):
    print(text, 'spam')
```

a.py:

```
import b
b.spam('qqq')
```

`import b` - (во время выполнения программы) загрузить файл b.py (если он еще не загружен) и дать доступ к его атрибутам через имя модуля b.

Атрибуты модуля - имена глобальных переменных этого модуля и его функции.

import - связь имен (модуля) и загруженных объектов (модулей). **Выполняется содержимое импортируемого файла**

`b.spam` - извлечь значение имени spam, расположенное в модуле b.

Любой файл может импортировать функциональные возможности из любого другого файла.

- a.py может импортировать b.py
- b.py может импортировать c.py
- c.py может импортировать b.py (который импортирует c.py, который импортирует b.py и так далее)
- и все могут импортировать любые модули стандартных и прочих библиотек.

Как работает import

- Ищется информация об этом модуле в словаре `sys.modules`.
  - Если модуль в словаре уже есть, берут расположенный в памяти объект модуля.
  - Если модуля в словаре нет (первый импорт):
    - Ищут файл модуля.
    - Компилируют его в байт-код (если нужно).
    - Выполняют его, чтобы создать его объекты.

Поиск файла модуля

О *пути поиска модулей* (где искать) - (см. далее).

НЕ пишем расширение файла .py. НЕ пишем путь к пакету.

Компиляция модуля

- Если .pyc файла нет, его создают;
- Интерпретатор проверяет время модификации файла .py и .pyc (кода и байт-кода). Если байт-код устарел (время модификации файла .py больше, чем время модификации файла .pyc), то перекомпилируют .py файл.
- далее используется .pyc файл.

Выполнение модуля

Все инструкции байт-кода выполняются по порядку, сверху вниз.

Все операции = создают новые атрибуты объекта модуля. Инструкции def определяют объекты для дальнейшего использования.

Заметьте, если в файле на верхнем уровне (вне всяких функций) был вызван print, то он выполнится в этот момент.

`imp.reload` - еще раз импортировать файл (см. далее).

Путь поиска модуля

Формируем `sys.path` из:

- домашний каталог программы;
- environment variable PYTHONPATH (если определена);
- каталоги стандартной библиотеки;
- содержимое файлов .pth (если они есть)

*Note On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words the directory containing the symlink is **not** added to the module search path.*

Иногда (в зависимости от ОС и версии питона) может быть добавлена директория, откуда запускали программму.

Домашний каталог программы - это либо где лежит запускаемый файл, либо в интерактивной оболочке - откуда была запущена эта оболочка.

Проверим, что "там, где лежит запускаемый файл". Файл `1_syspath.py` печатает содержимое переменной sys.path.

```
$ pwd
/ABC/chapter_mod
$ python3 examples/1_syspath.py
['/ABC/chapter_mod/examples', ...]
```

Если все файлы вашей программы в одном каталоге, то они всегда найдутся.

Осторожно! Не переопределите нужные модули, модулями в домашнем каталоге (если вы этого не хотите).

Файл .pth (от слова path) - список каталогов по 1 на строку. Альтернатива PYTHONPATH. См. описание модуля site

Куда положить такой файл?

- где установлен питон
  - Windows например C:\Python30
- подкаталог `site-packages` стандартной библиотеки:
  - `C:\Python30\Lib\site-packages` - Windows
  - `/usr/local/lib/python3.0/site-packages` - UNIX или
  - `/usr/local/lib/site-python`

Отфильтровываются несуществующие директории и повторяющиеся имена.

В `sys.path` удобно печатать директории (чтобы пороверить где ищем модули) и править ее "на лету", чтобы добавлять нужные директории во время выполнения программы.

Лутц, стр 618: Некоторым программам действительно требуется изменять sys.path. Сценарии, которые выполняются на веб-сервере, например, обычно выполняются с привилегиями пользователя «nobody» с целью ограничить доступ к системе. Поскольку такие сценарии обычно не должны зависеть от значения переменной окружения PYTHONPATH для пользователя «nobody», они часто изменяют список sys.path вручную, чтобы включить в него необходимые каталоги до того, как будет выполнена какая-либо инструкция import. Обычно для этого бывает достаточно вызова sys.path.append(dirname).

Почему не пишем расширение файла?

По инструкции `import b` может загрузить:

- файл с исходным кодом b.py
- байт-код этого файла b.pyc
- содержимое каталога b при импортировании пакета (см. далее "Пакеты модулей")

- Скомпилированный модуль расширения, написанный, как правило, на языке C или C++ и скомпонованный в виде динамической библиотеки (например, b.so в Linux и b.dll или b.pyd в Cygwin и в Windows).
- Скомпилированный встроенный модуль, написанный на языке C и статически скомпонованный с интерпретатором Python.
- Файл ZIP-архива с компонентом, который автоматически извлекается при импорте.
- Образ памяти для фиксированных двоичных исполняемых файлов.
- Класс Java в версии Jython.
- Компонент .NET в версии IronPython.

Если есть два файла b.py и b.so:

- в разных каталогах - возьмут из того каталога, который раньше в sys.path.
- в одном каталоге - не делайте так! (исключение .py и .pyc)

distutils - пакет для сборки и установки модулей (небольшой перевод статьи на русский)

Запуск модуля как модуля

Для запуска файла как модуля питона запустите интерпретатор с ключом **-m** и пишите файл без расширения. Т.е. файл `1_syspath.py` мы запишем как `1_syspath`

Обратите внимание, как изменилось содержимое sys.path

```
$ python3 -m 1_syspath
['', '/cygdrive/c/Users/taty/GitBook/Library/tatyderb/python-express-course/chapter_mo
d/examples/C', '/cygdrive/c/Users/taty/GitBook/Library/tatyderb/python-express-course/
chapter_mod/examples/\\Python27', '/usr/lib/python36.zip', '/usr/lib/python3.6', '/usr
/lib/python3.6/lib-dynload', '/usr/lib/python3.6/site-packages']
```

и

```
$ python3 1_syspath.py
['/cygdrive/c/Users/taty/GitBook/Library/tatyderb/python-express-course/chapter_mod/ex
amples', '/cygdrive/c/Users/taty/GitBook/Library/tatyderb/python-express-course/chapte
r_mod/examples/C', '/cygdrive/c/Users/taty/GitBook/Library/tatyderb/python-express-cou
rse/chapter_mod/examples/\\Python27', '/usr/lib/python36.zip', '/usr/lib/python3.6', '
/usr/lib/python3.6/lib-dynload', '/usr/lib/python3.6/site-packages']
```

Работа с модулями

Имена модулей - должны соотвествовать тем же правилам, что любые идентификаторы (алфавитные символы, цифры, подчеркивание; не ключевые слова).

То есть пробел в имени файла не допустим, даже если ОС это допускает.

Использование модулей

Пусть есть файл b.py:

```
def spam(text):
    print(text, 'spam')
```

Если мы хотим импортировать этот файл и вызвать функцию smap, то можем:

import	вызов	что значит
`import b`	b.spam('a')	b - имя переменной, которая ссылается на объект модуля после его загрузки
`from b import spam`	spam('a')	импорт модуля + копирование имени spam в пространство имен
`from b import *`	spam('a')	импорт модуля + копирование **всех** имен верхнего уровня (вне def или class)

`from .. *` в Python3 может использоваться только на верхнем уровне.

as - псевдонимы модулей

Иногда у модуля слишком длинное имя или мы указываем имя модуля с пакетами. Тогда вместе с полным именем можно указать псевдоним, заданный с помощью **as**

```
import numpy as np
import matplotlib.pyplot as plt
```

Инструкции import и from - это операции присваивания

import и from - **выполняемые инструкции**, а не "объявления времени компиляции".

- можно вложить в if (импортируем один модуль, иначе импортируем другой)
- можно вложить в def (внутри функции)
- работают только тогда, когда до них дойдет интерпретатор.

Т.е. имена и модули не доступны, пока не были выполнены нужные import.

Операция присваивания, те.

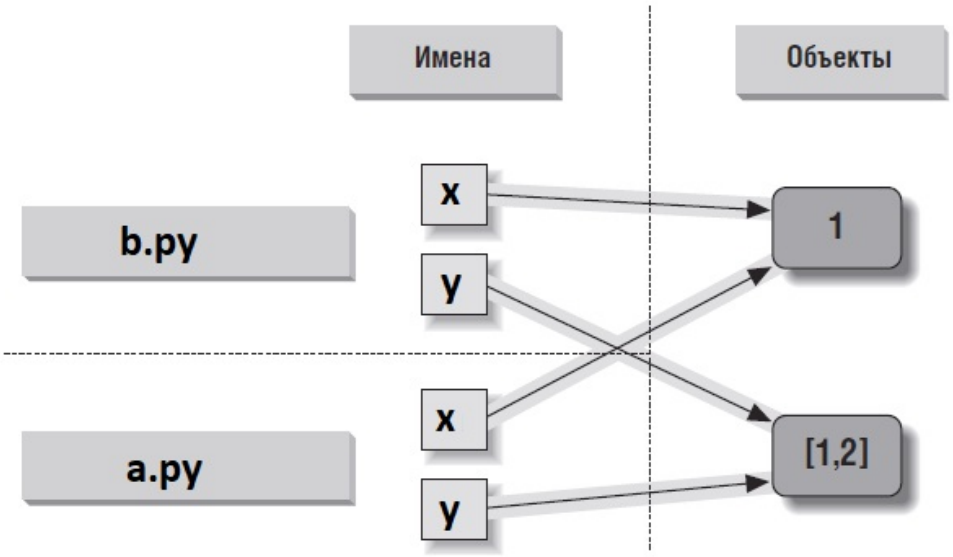
- import присваивает объект модуля единственному имени;
- from присваивает одно или более имен объектам с теми же именами в другом модуле. (т.е становятся ссылками на shared объекты)

b.py:

```
x = 1
y = [1, 2]
```

a.py:

```
from b import x, y  # скопировать два имени

x = 42              # изменяется только локальная переменная x
y[0] = 42           # изменяется объект по ссылке y

print(x, y)         # 42, [42, 2]
```



Хотим изменить x в модуле b.py.

- Это очень плохо, потому что потом не найдешь кто где менял эту x.
- Мы не контролируем изменения.
- Инкапсулируйте изменения через вызов функции-обертки в модуле b.py

Но очень хочется:

```
>>> from small import x, y  # Скопировать два имени
>>> x = 42                  # Изменить только локальное имя x
>>> import small            # Получить имя модуля
>>> small.x = 42            # Изменить x в другом модуле
```

Что лучше - import или from?

модуль.атрибут

- плюсы:
  - понятно из какого модуля используем этот атрибут (полезно при рефакторинге);
  - при одинаковых именах функциях в разных модулях мы явно прописываем из какого модуля используем функцию;
  - не может повредить существующее простанство имен (перезаписать существующую переменую x, которая теперь ссылается на атрибут модуля);
- минусы:
  - длинно, особенно если мы весь модуль, например, работаем с графическим модулем tkinter.
  - см. далее проблемы при reload

Рецепт (рекомендательный):

- import - предпочтительнее;
- from .. import - явно перечисляем имена;
- from .. import * - только при одном таком импорте.

Пространство имен модуля

Пространство имен - место, где создаются имена.

Атрибуты модуля - имена, которые находятся в модуле.

Каждое имя, которому присваивается некоторое значение на верхнем уровне файла модуля (то есть не вложенное в функции или в классы), превращается в атрибут этого модуля.

Если в файле b.py на верхнем уровне пишем `x = 1`, то имя x - это атрибут модуля b.

x - глобальная переменная для кода внутри b.py

К x извне можно доступиться по имени b.x

- Инструкции модуля выполняются при первом import.
  - создается объект модуля;
  - инструкции в модуле выполняются одна за другой.
- Присваивание на верхнем уровне создают атрибуты модуля (имена сохраняются в пространстве имен модуля):
  - **=**
  - def
  - class
- доступ к простанству имен модуля можно получить через b.**__dict__** или **dir(b)**:
  - __dict__ - все, что есть в пространстве - для работы
  - dir - вместе с унаследованными, не полное, отсортированное - для просмотра.
- модуль - единая область видимости (глобальная).

Файл module2.py:

```
Print('starting to load...')
import sys
name = 42
def func(): pass
class klass: pass
print('done loading.')
```

Импортируем его в интерпретаторе:

```
>>> import module2
starting to load...
done loading.
```

То есть код файла b.py действительно выполняется.

Область видимости модуля после импортирования превратилась в пространство имен атрибутов объекта модуля и они доступны по *модуль.атрибут*:

```
>>> module2.sys
<module 'sys' (built-in)>
>>> module2.name
42
>>> module2.func
<function func at 0x026D3BB8>>
>>> module2.klass
<class module2.klass>
```

Внутри интерпретатора пространство имен хранится в виде обычного словаря.
Разные пространства имен - разные словари.

```
>>> list(module2.__dict__.keys())
['name', '__builtins__', '__file__', '__package__', 'sys', 'klass', 'func',
'__name__', '__doc__']
```

- __file__ - имя файла, из которого был загружен модуль;
- __name__ - имя модуля (без диреторий и расширения файла).

Квалификация имен атрибутов

Это доступ по *объект.атрибут*

- x - **простая переменная** - ищется в текущих областях видимости по правилу LEGB.
- x.y - **квалифицированное имя** - ищем x в текущих областях видимости, далее ищем атрибут y в объекте x (а не областях видимости)
- x.y.z - **квалифицированные пути** - ищем сначала имя y в объекте x, потом имя z в объекте x.y

Квалификация имен применяется ко всем объектам, имеющим атрибуты (модули, классы, расширения на языке C и тд.)

Импортирование и область видимости

Никогда не можем получить автоматически доступ к переменным в другом файле. Чтобы доступаться к атрибутам, нужно всегда указывать у какого объекта этот атрибут.

Файл b.py:

```
x = 88          # x глобальна только для этого файла
def f():
    global x    # будем изменять x в этом файле
    x = 99      # имена в других модулях недоступны
```

Файл a.py:

```
x = 11          # x глобальна только для этого файла
import b        # получаем доступ к именам в модуле b
b.f()           # изменяет переменную b.x, но не x этого файла
print(x, b.x)   # 11 99
```

Запустим a.py, получим 11 99.

Когда вызвали b.f(), ее глобальной областью видимости будет тот файл, где она **написана**, а не там, откуда вызвана.

Вложенные пространства имен

Операция импорта не дает возможности из модуля b доступиться к переменным модуля a (к внешней области видимости).

Но можно доступиться к вложенным областям видимости.

```
# c.py:
x = 3
# b.py:
x = 2
import c
print(x, c.x)        # 2 3
# a.py:
x = 1
import b
print(x, b.x, b.c.x) # 1 2 3
```

Запустим файл a.py:

```
2 3
1 2 3
```

Повторная загрузка модулей

- модуль ищется, загружается и выполняется только при первом import;
- при следующих import будет использоваться объект уже загруженного модуля;
- функция **imp.reload** принудительно выполняет загрузку уже загруженного модуля и выполняет его код. Инструкции присваивания, которые выполняются при повтороном запуске, изменяют уже **существующий объект** модуля. Объект модуля **изменяется**, а не удаляется и создается повторно.

Используется, для ускорения разработки. Поменяли модуль - и можем не останавливая всю программу перегрузить его и выполнить заново какие-то функции.

Обновление сервисов, которые нельзя останавливать.

reload повторно загружает только модули на языке Python. На языке C, например, могут загружаться динамически, но не могут перегужаться через reload.

Отличия reload от import:

- Функция, а не инструкция.
- ей передается объект модуля, а не имя.
- не забудьте `import imp`

```
import module               # Первоначальное импортирование
что-то делаем, используя этот модуль
from imp import reload      # Импортировать функцию reload (в 3.0)
reload(module)              # Загрузить обновленный модуль
делаем что-то дальше, используя обновленные атрибуты модуля
```

- **Функция reload запускает новый программный код в файле модуля в текущем пространстве имен модуля**. При повторном выполнении программный код перезаписывает существующее пространство имен вместо того, чтобы удалять его и создавать вновь.
- **Инструкции присваивания на верхнем уровне файла замещают имена новыми значениями**. Например, повторный запуск инструкции def приводит к замещению предыдущей версии функции в пространстве имен модуля, выполняя повторную операцию присваивания имени функции.
- **Повторная загрузка оказывает воздействие на всех клиентов, использовавших инструкцию import для получения доступа к модулю.** Клиенты, использовавшие инструкцию import, получают доступ к атрибутам модуля, указывая полные их имена, поэтому после повторной загрузки они будут получать новые значения атрибутов.
- **Повторная загрузка будет воздействовать лишь на тех клиентов, которые еще только будут использовать инструкцию from в будущем.** Клиенты, которые использовали инструкцию from для получения доступа к атрибутам в прошлом, не заметят изменений, произошедших в результате повторной загрузки, – они по-прежнему будут ссылаться на старые объекты, получен- ные до выполнения перезагрузки.

Пакеты модулей

Модуль - 1 файл.

Пакет - 1 директория.

Можно импортировать пакеты.

Где указывали имя файла, можно указать список директорий через точку.

```
import dir1.dir2.mod
```

или

```
from dir1.dir2.mod import x
```

Т.е предполагается, что есть директория dir0 (находится в пути поиска). В нем есть директория dir1, в которой есть dir2, в которой лежит модуль mod (в котором есть атрибут x).

Пишем через точку, получаем платформо-независимый доступ к директориям пакета.

Делаем пакет. Файл __init__.py

Каждая директория, которая указана в пути импорта, должна содержать файл __init__.py

То есть в директориях dir1 и dir2 такие файлы должны быть. В директории dir0 - не обязательно (она не указана в инструкции import).

```
dir0\           # <- должен быть в sys.path
    dir1\
        __init__.py
        dir2\
            __init__.py
            mod.py
```

Что должно лежать в файле __init__.py? Можно оставить его пустым. Можно указать, что будет импортироваться при `from модуль import *`.

- **Инициализация пакета** - при первом импорте каталога, питон выполняет код в файле __init__.py этого каталога. Здесь можно открыть соединение с БД, создать файл с данными и тп.
- **Инициализация пространства имен модуля**. Получаем дерево вложенных объектов. Далее можно использовать dir1.dir2 - этот объект содержит все имена, которые определяет __init__.py из dir2.

В dir1 может не быть модулей (других файлов, кроме __init__.py).

__init__.py - создает пространство имен для объектов модулей (особенно, если модулей в смысле файлов, в директории нет).

- **from .. *** - импортируется или все имена из каталога, либо, если в __init__.py определен список **__all__**, то только имена из этого списка.

Пример импортирования пакета

Определим __init__.py и mod.py файлы описанного дерева:

```
# Файл: dir1\__init__.py
Print('dir1 init')
x = 1
# Файл: dir1\dir2\__init__.py
Print('dir2 init')
y = 2
# Файл: dir1\dir2\mod.py
Print('in mod.py')
z = 3
```

Запускаем:

```
% python
>>> import dir1.dir2.mod    # Сначала запускаются файлы инициализации
dir1 init
dir2 init
in mod.py
>>>
>>> import dir1.dir2.mod    # Повторное импортирование не выполняется
>>>
>>> from imp import reload  # Требуется в версии 3.0
>>> reload(dir1)
dir1 init
<module 'dir1' from 'dir1\__init__.pyc'>
>>>
>>> reload(dir1.dir2)
dir2 init
<module 'dir1.dir2' from 'dir1\dir2\__init__.pyc'>
```

Вложенные объекты:

```
>>> dir1
<module 'dir1' from 'dir1\__init__.pyc'>
>>> dir1.dir2
<module 'dir1.dir2' from 'dir1\dir2\__init__.pyc'>
>>> dir1.dir2.mod
<module 'dir1.dir2.mod' from 'dir1\dir2\mod.pyc'>
```

Можно доступиться к переменным, определенным в __init__.py и mod.py файлах:

```
>>> dir1.x
1
>>> dir1.dir2.y
2
>>> dir1.dir2.mod.z
3
```

Как написать короче

Приходится писать полные пути, короткие не работают:

```
>>> dir2.mod
NameError: name ‘dir2’ is not defined
>>> mod.z
NameError: name ‘mod’ is not defined
```

Лучше всего использовать **as**

```
% python
>>> from dir1.dir2 import mod   # Описание пути находится только в этом месте
dir1 init
dir2 init
in mod.py
>>> mod.z                       # Указывать полный путь не требуется
3
>>> from dir1.dir2.mod import z
>>> z
3
>>> import dir1.dir2.mod as mod # Использование короткого синонима
>>> mod.z                       # теперь работает
3
```

Запуск модуля

Все указанные директории обязаны иметь __ini__.py файлы.

Стоим в dir0:

```
python3 -m dir1.dir2.mod
```

Запуск пакета

Написанные на питоне сервисы обычно оформляют как пакеты. Для запуска используется корневой модуль пакета, т.е. запуск пакета как программы.

В корневой директории пакета должен быть файл __init__.py, в нем, с указанием `if __name__ == '__main__'` должен быть разбор аргументов командной строки при запуске и запуск сервиса.

Когда нужно импортировать пакеты

Разумно в путь поиска добавлять только корневую директорию проекта и дальше использовать импортирование пакетов (а не перечислять там все директории).

Разрешение неоднозначностей

Пусть есть проект1 в директории system1:

```
system1\
    utilities.py    # Общие вспомогательные функции, классы
    main.py         # Этот файл запускает программу
    other.py        # Импортирует и использует модуль utilities
```

И другой проект2 в директории system2:

```
system2\
    utilities.py    # Общие вспомогательные функции, классы
    main.py         # Этот файл запускает программу
    other.py        # Импортирует и использует модуль utilities
```

Эти две программы (каждая из них запускается из своего main.py) не мешают друг другу, так как при запуске main.py импорт ищется сначала в той директории, где лежал запускаемый файл.

Теперь я хочу в новом проекте использовать как функции из `system1\utilities.py`, так и из `system2\utilities.py`

Допустим, оба проекта у нас в пути поиска модулей. Тогда можно написать:

```
import utilities
utilities.func('spam')
```

Но из какого пакета нам достался модуль utilities?

Можно поставить очередность каталогов в sys.path. Какой из них сделать первым?

Разумно объединить проекты в одной директории dir0:

```
dir0\
    __init__.py      # работает и без этого, но хорошо бы добавить и корневой инит-файл

    system1\
        __init__.py     # добавили
        utilities.py
        main.py         # импорты здесь работают по-старому
        other.py
    system2\
        __init__.py     # добавили
        utilities.py
        main.py         # импорты здесь работают по-старому
        other.py
    system3\            # Здесь или в другом месте
        __init__.py     # располагается ваш новый программный код
        myfile.py
```

теперь можно импортировать в myfile с указанием каталогов:

```
import system1.utilities
import system2.utilities
system1.utilities.function('spam')
system2.utilities.function('eggs')
```

а еще лучше - написать разные as

```
import system1.utilities as util1   # придумайте себе более разумные сокращения, по ти
пу утилит
import system2.utilities as util2
util1.function('spam')
util2.function('eggs')
```

Не обязательно system3 класть в ту же dir0, но раз проекты берут что-то друг у друга, вдруг будет проект system4, который будет брать разные части из предыдущих трех проектов. Сразу задумаемся о будущем удобстве.

Изменения в Python 3.0

Лутц стр 650 Принцип действия операции импортирования внутри пакетов немного изме- нился в Python 3.0. Изменения коснулись лишь импортирования файлов паке- та из файлов, находящихся в каталогах этого же пакета, о котором мы говорим в этой главе, – операция импортирования других файлов действует, как и пре- жде. В Python 3.0 в операцию импортирования внутри пакетов было внесено два изменения:

- Изменилась семантика пути поиска модулей так, что теперь операция им- портирования модуля по умолчанию пропускает собственный каталог па- кета. Она проверяет только компоненты пути поиска. Эта операция называ- ется импортированием по «абсолютному» пути.
- Расширен синтаксис инструкции from так, что теперь имеется возможность явно указать, что поиск импортируемых модулей должен производиться только в каталоге пакета. Эта операция называется импортированием по «относительному» пути.

Суть этих изменений в версии 3.0 (и в 2.6, если они используются) состоит в том, что вы должны использовать специальный синтаксис инструкции from для импортирования модулей, находящихся в том же пакете, что и импорти- рующий модуль, если вы не указываете полный путь к модулю, начиная от корневого каталога пакета. Если не использовать этот синтаксис, интерпрета- тор не сможет отыскать требуемый модуль в пакете.

Относительные пути

Это пути с . или ..

- Если в пути указана . или .., то поиск модуля начинается с текущей директории (или директории выше)
- Если в пути не указаны точки в начале, то текущая директория не входит в путь поиска модулей.

Относительный путь можно указать только в операции from, в import - нельзя

Запускайте программу как модуль, иначе у нее только __main__ модуль и имеется: Note that relative imports are based on the name of the current module. Since the name of the main module is always "__main__", modules intended for use as the main module of a Python application must always use absolute imports.

```
import string   # Пропустит поиск модуля в пакете, взять build-in модуль string
from . import string    # Поиск выполняется только в пределах пакета
```

Пусть у нас есть пакет mypkg с модулем string:

```
from . import string # Импортирует mypkg.string (относительно пакета)
from .string import name1, name2 # Импортирует имена из mypkg.string
```

Пусть мы вызваем в модуле A.B.C импорты:

```
from . import D     # Импортирует A.B.D (. означает A.B)
from .. import E    # Импортирует A.E (.. означает A)
from .D import X    # Импортирует A.B.D.X (. означает A.B)
from ..E import X   # Импортирует A.E.X (.. означает A)
```

Импортирование по относительному пути происходит:

- только внутри пакета;
- только инструкция from;

Правила поиска модулей:

- Для простых имен пакетов (например, A) поиск выполняется во всех директориях sys.path, слева направо. Этот список делается из системных значений по умолчанию и из настроек пользователя.
- пакет - это директория с модулями (и файлом __ini__.py в каждой поддиректории). Чтобы доступиться к A.B.C в sys.path должна быть директория A.
- модули по абсолютным путям ищутся как раньше
- модули по относительным путям идет относительно текущего пакета (а обычный поиск в sys.path не выполняется) `from . import A` - ищет только в текущей директории пакета.

Примеры

Импорт за пределами пакета

Если в текущей директории нет файла string.py, то этот код загружает стандартный модуль string (т.е. импорт внешних модулей работает как обычно).

```
C:\test> c:\Python30\python
>>> import string
>>> string
<module 'string' from 'c:\Python30\lib\string.py'>
```

Добавим в текущий каталог string.py:

```
# test\string.py
print('I am local string module')
C:\test> c:\Python30\python
>>> import string
I am local string module
>>> string
<module 'string' from 'string.py'>
```

Вывод: обычный импорт сначала ищет в текущем каталоге.

Пакета на этом уровне нет (так как нет __ini__.py), поэтому поиск по . не проходит:

```
>>> from . import string
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: Attempted relative import in non-package
```

Импорт в других файлах тоже работает:

```
# test\main.py
import string
print(string)
C:\test> C:\python30\python main.py # Тот же результат получается в 2.6
I am local string module
<module 'string' from 'C:\test\string.py'>
```

Импортирование внутри пакета

Удалим локальный string.py и создадим директорию pkg с пустым __init__.py:

```
C:\test> del string*
C:\test> mkdir pkg
# test\pkg\spam.py
import eggs # <== Работает в 2.6, но не в 3.0!
print(eggs.X)
# test\pkg\eggs.py
X = 99999
import string
print(string)
```

Дополнительные возможности модулей

Сокрытие данных в модулях

Конструкция `from .. import *` импортирует иногда лишнее (что является внутренними переменными и методами модуля и не являются его API).

Как с этим бороться? (Никак, увы.)

- переменные (и функции) вида _x не доступны при таком импорте;
- можно в __init__.py файле определить список **__all__** и тогда будут импортироваться при * только указанные имена.

```
__all__ = ['Error', 'encode', 'decode'] # Экспортируются только эти имена
```

Ничто не мешает нужный метод или переменную импортировать, явно указав имя.

Изменение пути поиска модулей

Т.е. нужно изменить sys.path.

Например:

```
>>> import sys
>>> sys.path
['', 'C:\\users', 'C:\\Windows\\system32\\python30.zip', ...далее опущено...]
>>> sys.path.append('C:\\sourcedir')        # Дополнение пути поиска модулей
>>> import string                           # Новый каталог будет участвовать в поиске
```

Так можно полностью переписать пути поиска:

```
>>> sys.path = [r'd:\temp'] # Изменяет путь поиска модулей
>>> sys.path.append('c:\\lp4e\\examples') # Только для этой программы
>>> sys.path
['d:\\temp', 'c:\\lp4e\\examples']
>>> import string
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ImportError: No module named string
```

Добавить текущую директорию в sys.path (далее расскажут об используемых модулях и методах).

```
from os.path import dirname
sys.path.append(dirname(__file__))
```

Модули - это объекты

Пусть есть модуль M. В нем есть атрибут name.

Доступ к атрибуту:

```
M.name                  # Полное имя объекта
M.__dict__['name']      # Доступ с использованием словаря пространства имен
sys.modules['M'].name   # Доступ через таблицу загруженных модулей
getattr(M, 'name')      # Доступ с помощью встроенной функции
```

Документация модулей

В начале модуля можно задать строку документации в тройных кавычках. Файл mydir.py:

```
"""
mydir.py: описание для чего служит файл
"""

def listing(module, verbose=True):
    # далее код модуля
```

Написанную документацию для модуля mydir можно вывести как:

```
>>> import mydir
>>> help(mydir)
Help on module mydir:
NAME
    mydir - mydir.py: описание для чего служит файл
FILE
    c:\users\veramark\mark\mydir.py
FUNCTIONS
    listing(module, verbose=True)
```

Импортирование модуля в виде строки

Нельзя написать:

```
>>> import "string"
File "<stdin>", line 1
import "string"
^
SyntaxError: invalid syntax
```

или

```
x = "string"
import x        # тут пытаемся импортировать не модуль string, а ищем файл x.py
```

Выполним код с помощью **exec()**

```
>>> modname = 'string'
>>> exec('import ' + modname)   # Выполняется как строка программного кода
>>> string                      # Модуль был импортирован в пространство имен
<module 'string' from 'c:\Python30\lib\string.py'>
```

или вызовем функцию **__import__()**

```
>>> modname = 'string'
>>> string = __import__(modname)
>>> string
<module 'string' from 'c:\Python30\lib\string.py'>
```

Перегрузка модулей (еще раз)

При перегрузке модуля через reload иногда возникает проблема, что нужно перегрузить все модули, которые были в него импортированы.

Придется писать код, который "вручную" перегружает все необходимые модули. Можно для этого анализировать __dict__ Лутц, стр 679 - пример такого кода.

Контрольные вопросы и ответы по разделам

Лутц, стр 621

Контрольные вопросы

1. Каким образом файл с исходным программным кодом модуля превращается в объект модуля?
2. Зачем может потребоваться настраивать значение переменной окружения PYTHONPATH?
3. Назовите четыре основных компонента, составляющих путь поиска модулей.
4. Назовите четыре типа файлов, которые могут загружаться операцией импортирования.
5. Что такое пространство имен, и что содержит пространство имен модуля?

Ответы

1. Файл с исходными текстами модуля автоматически превращается в объект модуля в результате выполнения операции импортирования. С технической точки зрения исходный программный код модуля выполняется во время импортирования, инструкция за инструкцией, и все имена, которым по мере выполнения операций будут присвоены значения, превращаются в атрибуты объекта модуля.
2. Настройка переменной PYTHONPATH может потребоваться только в случае необходимости импортировать модули, размещенные в каталогах, отличных от того, в котором вы работаете (то есть отличных от текущего каталога при работе в интерактивной оболочке или от каталога, где находится главный файл программы).
3. Четырьмя основными компонентами, составляющими путь поиска модулей, являются: домашний каталог главного файла программы (каталог, в котором он находится), все каталоги, перечисленные в переменной окружения PYTHONPATH, каталоги стандартной библиотеки и все каталоги в файлах с расширением .pth, размещенных в стандартных местах. Из них доступны для настройки переменная окружения PYTHONPATH и файлы с расширением .pth.
4. Интерпретатор Python может загружать файлы с исходными текстами (.py), файлы с байт-кодом (.pyc), файлы расширений, написанных на языке C (например,

файлы с расширением .so в Linux или с расширением .dll в Windows), или каталог с указанным именем, в случае импортирования пакета. Операция импортирования может также загружать менее обычные файлы, такие как компоненты из архивов в формате ZIP, классы Java в Jython – версии Python, компоненты .NET в IronPython и статически скомпонованные расширения, написанные на языке C, которые вообще не представлены в виде файлов. С помощью программных ловушек, которые имеет реализация операции импорта, можно загрузить все, что угодно.

5. Пространство имен – это независимый пакет переменных, известных как атрибуты пространства имен объекта. Пространство имен модуля содер- жит все имена, присваивание значений которым производится программ- ным кодом на верхнем уровне модуля (то есть не вложенным в инструкции def или class). С технической точки зрения глобальная область видимости трансформируется в пространство имен атрибутов объекта модуля. Про- странство имен модуля может изменяться с помощью операций присваива- ния из других файлов, которые импортируют данный модуль, хотя это и не приветствуется (подробнее об этом рассказывается в главе 17).

Контрольные вопросы

1. Как создать модуль?
2. Как взаимосвязаны инструкции from и import?
3. Какое отношение к операции импортирования имеет функция reload?
4. Когда вместо инструкции from следует использовать инструкцию import?
5. Назовите три потенциальных ловушки инструкции from.

Ответы

1. Чтобы создать модуль, достаточно просто создать текстовый файл с ин- струкциями на языке Python; любой файл с исходным программным кодом автоматически становится модулем – нет никаких синтаксических кон- струкций для его объявления. Можно также создать модуль, написав про- граммный код на другом языке программирования, таком как C или Java, но такие модули находятся вне рассмотрения этой книги.
2. Инструкция from импортирует модуль целиком, как и инструкция import, но кроме этого она еще копирует одно или более имен из импортируемого модуля в ту область видимости, где находится инструкция from. Это позво- ляет использовать импортированные имена напрямую (name), без дополне- ния их именем модуля (module.name).
3. По умолчанию модуль импортируется один раз за все время выполнения

программы. Функция reload принудительно выполняет повторное импор- тирование. Она часто используется, чтобы загрузить новую версию исходного программного кода модуля в процессе разработки и в случаях динами- ческой настройки.

4. Инструкция import обязательно должна использоваться вместо инструкции from, только когда необходимо обеспечить доступ к одному и тому же имени в двух разных модулях, – поскольку вы будете вынуждены указывать име- на вмещающих модулей, эти два имени будут уникальны.
5. Инструкция from может делать непонятным смысл переменной (в каком модуле она определена), вызывать проблемы при использовании функ- ции reload (имена могут ссылаться на прежние версии объектов) и может повреждать пространства имен (может приводить к перезаписи значений имен, используемых в вашей области видимости). Самой худшей, во многих отношениях, является форма from * – она может приводить к серьезным по- вреждениям пространств имен и скрывать смысл переменных – эту форму инструкции следует использовать с большой осторожностью.

Контрольные вопросы

1. Для чего служит файл __init__.py в каталогах пакетов модулей?
2. Как избежать необходимости снова и снова вводить полное имя пакета при каждом обращении к содержимому пакетов?
3. В каких каталогах необходимо создавать файл __init__.py?
4. В каких случаях вместо инструкции from приходится использовать ин- струкцию import?
5. Чем отличаются инструкции from mypkg import spam и from . import spam?

Ответы

1. Файлы __init__.py служат для объявления и инициализации пакета, – ин- терпретатор автоматически запускает программный код в этих файлах, когда каталог импортируется программой впервые. Переменные, которым выполняется присваивание в этих файлах, становятся атрибутами объекта модуля для соответствующего каталога. Присутствие этих файлов в ката- логах пакетов обязательно – вы не сможете импортировать пакеты при от- сутствии этих файлов в каталогах.
2. Используйте инструкцию from, чтобы скопировать имена из пакета или воспользуйтесь расширением as инструкции import, чтобы переименовать полный путь в короткий синоним. В обоих случаях полный путь будет при- сутствовать

только в одном месте – в инструкции from или import.

3. Каждый каталог, перечисленный в инструкции import или from, должен со- держать файл __init__.py. Другие каталоги, включая каталог, содержащий самый первый компонент пути к пакету, не требуют наличия этого файла.
4. Инструкция import должна использоваться вместо инструкции from, только если вам необходимо обеспечить доступ к одному и тому же имени более чем в одном каталоге. Благодаря инструкции import употребление полного пути обеспечивает уникальность ссылок, тогда как инструкция from допускает наличие только одной версии любого имени.
5. Инструкция from mypkg import spam выполняет импорт по абсолютному пути – при поиске mypkg она пропускает каталог пакета и пользуется спи- ском каталогов в sys.path. Инструкция from . import spam, напротив, выпол- няет импорт относительно текущего пакета – поиск модуля spam выпол- няется относительно пакета, внутри которого находится эта инструкция.

Задачи на исключения

Реализовать сложение и скалярное умножение векторов, чтобы возбуждалось исключение, если длина векторов не совпадает.

Исследовать работу кода, если используется функция zip, map, reduce и тп.

Задачи

```
from os.path import dirname
sys.path.append(dirname(__file__))
```

импорт модуля

В одном файле определите функцию, которая считает nod для двух целых чисел по алгоритму Евклида.

В другом файле этой же директории используйте функцию nod (например, дана дробь в виде числителя и знаменателя через пробел, сократите дробь и напечатайте ее в виде "числитель / знаменатель".

импорт пакета

относительный импорт

Импортируйте модуль как .b, как ..b и как ..subdir2.b

тесты в отдельной директории

Возьмите задачу и напишите тесты по этой задаче в отдельной директории tests. Т.е. для файла a.py разместите тесты в tests/a.py

Задачи из Лутца

Упражнения к пятой части Решения приводятся в разделе «Часть V, Модули» приложения A.

1. Основы импортирования. Напишите программу, которая подсчитывает количество строк и символов в файле (в духе утилиты wc в операционной систе- мы UNIX). В своем текстовом редакторе создайте модуль с именем mymod. py, который экспортирует три имени: •• Функцию countLines(name), которая читает входной файл и подсчитывает число строк в нем (подсказка: большую часть работы можно выполнить с помощью метода file.readlines, а оставшуюся часть – с помощью функ- ции len). •• Функцию countChars(name), которая читает входной файл и подсчитыва- ет число символов в нем (подсказка: метод file.read возвращает единую строку). •• Функцию test(name), которая вызывает две предыдущие функции с за- данным именем файла. Вообще говоря, имя файла можно жестко определить в программном коде, принимать ввод от пользователя или принимать имя как параметр командной строки через список sys.argv – но пока исходите из предположения, что оно передается как аргумент функции. Все три функции в модуле mymod должны принимать имя файла в виде стро- ки. Если размер любой из функций превысит две-три строки, это значит, что вы делаете лишнюю работу, – используйте подсказки, которые я вам дал!

Затем проверьте свой модуль в интерактивной оболочке, используя ин- струкцию import и полные имена экспортируемых функций. Следует ли до- бавить в переменную PYTHONPATH каталог, где находится ваш файл mymod. py? Попробуйте проверить модуль на самом себе: например, test(“mymod.py”). Обратите внимание, что функция test открывает файл дважды, – если вы достаточно честолюбивы, попробуйте оптимизировать программный код, передавая двум функциям счета объект открытого файла (подсказка: метод file.seek(0) выполняет переустановку указателя в начало файла).

1. from/from . *Проверьте модуль mymod из упражнения 1 в интерактивной оболочке, используя для загрузки экспортируемых имен инструкцию from – сначала по имени, а потом с помощью формы from* .
2. **main**. Добавьте в модуль mymod строку, в которой автоматически про- изводился бы вызов функции test, только когда модуль выполняется как самостоятельный сценарий, а не во время импортирования. Добавляемая вами строка, вероятно, должна содержать проверку значения атрибута **name** на равенство строке “**main**”, как было показано в этой главе. Попробуйте запустить модуль из системной командной строки, затем им- портируйте модуль и проверьте работу функций в интерактивном режиме. Будут ли работать функции в обоих режимах?
3. Вложенное импортирование. Напишите второй модуль myclient.py, кото- рый импортирует модуль mymod и проверяет работу его функций, затем за- пустите myclient из системной командной строки. Будут ли доступны функ- ции из mymod

на верхнем уровне myclient, если импортировать их с помощью инструкции from? А если они будут импортированы с помощью инструкции import? Попробуйте реализовать оба варианта в myclient и проверить в инте- рактивном режиме, импортируя модуль myclient и проверяя содержимое его атрибута **dict**.

4. Импорт пакетов. Импортируйте ваш файл из пакета. Создайте каталог с именем mypkg, вложенный в каталог, находящийся в пути поиска моду- лей. Переместите в него файл mymod.py, созданный в упражнении 1 или 3, и попробуйте импортировать его как пакет, инструкцией import mypkg.mymod. Вам потребуется добавить файл **init**.py в каталог, куда был помещен ваш модуль. Это упражнение должно работать на всех основных платфор- мах Python (это одна из причин, почему в языке Python в качестве разде- лителя компонентов пути используется символ «.»). Каталог пакета может быть простым подкаталогом в вашем рабочем каталоге – в этом случае он будет обнаружен интерпретатором при поиске в домашнем каталоге и вам не потребуется настраивать путь поиска. Добавьте какой-нибудь программ- ный код в **init**.py и посмотрите, будет ли он выполняться при каждой операции импортирования.
5. Повторная загрузка. Поэкспериментируйте с возможностью повторной за- грузки модуля: выполните тесты, которые приводятся в примере changer.py в главе 22, многократно изменяя текст сообщения и/или поведение модуля, без остановки интерактивного сеанса работы с интерпретатором Python. В зависимости от операционной системы файл модуля можно было бы ре- дактировать в другом окне или, приостановив интерпретатор, редактиро- вать модуль в том же окне (в UNIX комбинация клавиш Ctrl-Z обычно при- водит к приостановке текущего процесса, а команда fg – возобновляет его работу).
6. Попробуйте импортировать модули рекурсивно. Что произойдет?

запуск файла

запуск модуля

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.

Термины на английском языке

Понадобятся при поиске в интернете.

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

Class - A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Class variable - A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.

Data member - A class variable or instance variable that holds data associated with a class and its objects.

Function overloading - The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.

Instance variable - A variable that is defined inside a method and belongs only to the current instance of a class.

Inheritance - The transfer of the characteristics of a class to other classes that are derived from it.

Instance - An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.

Instantiation - The creation of an instance of a class.

Method - A special kind of function that is defined in a class definition.

Object - A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Operator overloading - The assignment of more than one function to a particular operator.

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(
```

При чтении переменных идет поиск этой переменной в пространстве имен.

При = изменяется сам объект. (Быть может создается атрибут этого объекта).

```
class A(object):
    shared_data = 42

x = A()
y = A()
print(x.shared_data, y.shared_data, A.shared_data) # 42, 42, 42

A.shared_data = 99      # A - объект - класс
print(x.shared_data, y.shared_data, A.shared_data) # 99, 99, 99
                        # x - объект - экземпляр класса
x.shared_data = 100     # создали новый атрибут ЭКЗЕМПЛЯРА класса
print(x.shared_data, y.shared_data, A.shared_data) # 100, 99, 99
```

В классе и экземпляре класса может быть поле с одинаковым именем (не пишите так!):

```
class B(object):
    data = 'shared'                 # присваивание атрибуту класса
    def __init__(self, data):
        self.data = data            # присваивание атрибуту экземпляра
    def prn(self):
        print(self.data, B.data)    # атрибут экземпляра, атрибут класса

x = B(1)
y = B(2)
x.prn()     # 1 shared
y.prn()     # 2 shared
```

Вызов методов

Для любого объекта класса `класс` допустимы варианты вызова метода экземпляра класса:

```
экземпляр.метод(аргументы...)
класс.метод(экземпляр, аргументы...)
```

Например:

```
class A(object):
    def func(self, value):
        print(value)

x = A()
x.func('первый вызов')      # первый вызов
A.func(x, 'второй вызов')   # второй вызов
```

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

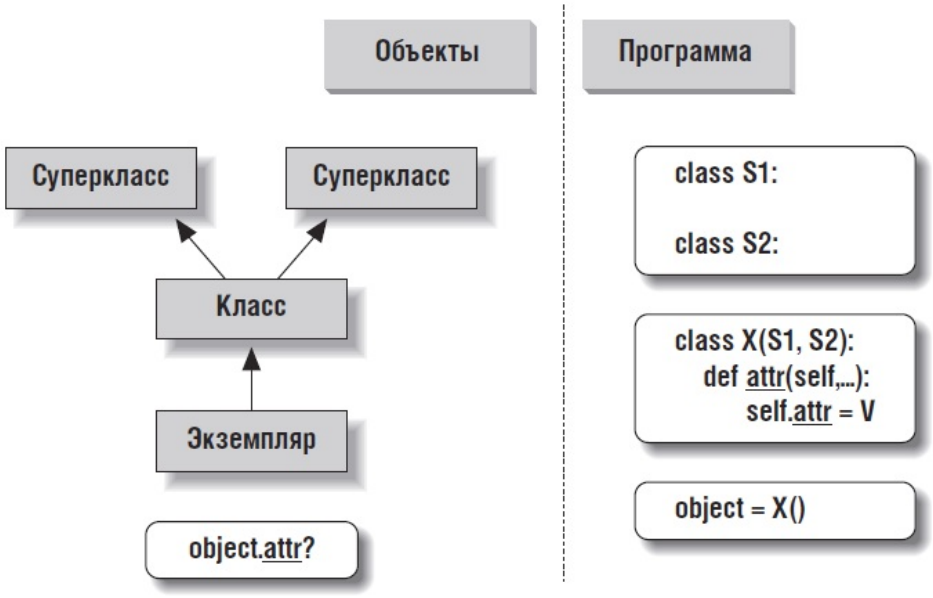
Лутц, стр 773

Имена и области видимости

Дерево атрибутов

- атрибуты экземпляра - создаются `self.атрибут =` в методах;
- атрибуты класса - создаются `атрибут =` внутри инструкции class;
- ссылки на суперклассы - перечисление при наследовании.

Поиск по дереву атрибутов - при любом обращении к атрибуту (даже при `self.атрибут`)



Функции при наследовании

Так как поиск атрибутов идет снизу вверх, то в подклассе можно:

- *заместить* атрибут суперкласса;
- *предоставить* атрибут суперкласса;
- *расширить* методы суперкласса за счет их вызова из метода подкласса.

Расширение:

```
>>> class Super:
...     def method(self):
...         print('in Super.method')
...
>>> class Sub(Super):
...     def method(self):                   # Переопределить метод
...         print('starting Sub.method')    # Дополнительное действие
...         Super.method(self)              # Выполнить действие по умолчанию
...         print(ending Sub.method')
```

Взаимодействие классов при наследовании

Файл примера specialize.py

- *Super* - имеет методы *method* и *delegate* (он хочет метод *action* в наследнике).
- *Inheritor* - ничего своего, все только наследует от *Super*.
- *Replacer* - переопределяет *Super.method* своей собственной версией.
- *Extender* - переопределяет *Super.method* так, что вызывает и его, для выполнения действий по умолчанию.
- *Provider* - реализует метод *action*, который ожидается метдом *Super.delegate*

```
class Super:
    def method(self):
        print('in Super.method')    # Поведение по умолчанию
    def delegate(self):
        self.action()               # Ожидаемый метод

class Inheritor(Super):             # Наследует методы, как они есть
    pass

class Replacer(Super):              # Полностью замещает method
    def method(self):
        print('in Replacer.method')

class Extender(Super):              # Расширяет поведение метода method
    def method(self):
        print('starting Extender.method')
        Super.method(self)
        print('ending Extender.method')

class Provider(Super):              # Определяет необходимый метод
    def action(self):
        print('in Provider.action')

if __name__ == '__main__':
    for myclass in (Inheritor, Replacer, Extender):
        print('\n' + myclass.__name__ + '...')
        myclass().method()
    print('\nProvider...')
    x = Provider()
    x.delegate()
```

Напечатает:

```
Inheritor...
in Super.method

Replacer...
in Replacer.method

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider...
in Provider.action
```

Абстрактные суперклассы

Когда класс *Provider* вызывается метод `delegate`, начинаются **две** независимых процедуры поиска.

- первый вариант:
  - при вызове `x.delegate()` интерпретатор ищет метод, начиная от класса *Provider* вверх по дереву наследования.
  - Экземпляр `x` передается в виде аргумента self.
- внутри метода `Super.delegate` выражение `self.action` запускает новый независимый поиск в дереве наследования, начиная от экземпляра self и далее вверх по дереву. Но *self* ссылается на экземпляр класса *Provider*, метод `action` будет найден в подклассе *Provider*.

Абстрактный суперкласс - класс, который ожидает, что часть его функционала будет реализована его детьми.

Хороший тон: сделать "абстрактность" неработоспособной, возбуждая `NotImplementedError` или написав `assert` (проверяет логическое выражение, если истина, идет дальше, если ложь, то останавливает выполнение с сообщением об ошибке):

```
class Super:
    def delegate(self):
        self.action()
    def action(self):
        assert False, 'action must be defined!' # При вызове этой версии

>>> X = Super()
>>> X.delegate()
AssertionError: action must be defined!
```

или через возбуждение исключения

```
class Super:
    def delegate(self):
        self.action()
    def action(self):
        raise NotImplementedError('action must be defined!')

>>> X = Super()
>>> X.delegate()
NotImplementedError: action must be defined!
```

Если наследник не реализует метод, то мы получим исключение и у него:

```
>>> class Sub(Super): pass
...
>>> X = Sub()
>>> X.delegate()
NotImplementedError: action must be defined!
```

до тех пор, пока не реализуем:

```
>>> class Sub(Super):
...     def action(self): print('Реализация тут')
...
>>> X = Sub()
>>> X.delegate()
Реализация тут
```

Абстрактные суперклассы в питоне 2.6 и 3.0

Абстрактный суперкласс можно определить с помощью специальных синтаксических конструкций. Они разные в питоне 2.6 и 3.0

Эти конструкции **запретят создавать экземпляры, если методы не будут определены в дереве ниже**.

Python 3.0 (подробнее разберем позже):

```
from abc import ABCMeta, abstractmethod
class Super(metaclass=ABCMeta):
    @abstractmethod
    def method(self, ...):
        pass
```

Python 2.6:

```
class Super:
    __metaclass__ = ABCMeta
    @abstractmethod
    def method(self, ...):
        pass
```

Чтобы запретить это для методов *delegate* и *action*, напишем эту конструкцию для питона 3.0:

```
>>> from abc import ABCMeta, abstractmethod
>>>
>>> class Super(metaclass=ABCMeta):
...     def delegate(self):
...         self.action()
...     @abstractmethod
...     def action(self):
...         pass

>>> X = Super()
TypeError: Can’t instantiate abstract class Super with abstract methods action
>>> class Sub(Super): pass
...
>>> X = Sub()
TypeError: Can’t instantiate abstract class Sub with abstract methods action
>>> class Sub(Super):
... def action(self): print('Реализация тут')
...
>>> X = Sub()
>>> X.delegate()
Реализация тут
```

Плюс: получаем ошибку раньше - при **создании экземпляра класса**, а не при попытке вызвать у экземпляра метод, которого нет.

Минус: увеличение кода.

О декораторах функций (@abstractmethod) и объявлении метаклассов расскажем потом.

Пространство имен

- **Неквалифицированные имена** (например, X) располагаются в областях видимости (namespace).
- **Квалифицированные имена атрибутов** (например, object.X) принадлежат простанствам имен объектов.
- Некоторые области видимости инициализируют пространства имен объектов (в модулях и классах)

Простые имена - глобальные, пока нет =

Правило нахождения имени LEGB.

- **Присваивание** (X = value) - делает имена локальными, создает или изменяет имя X в текущей локальной области видимости, если имя не объявлено глобальным
- **Ссылка** (X) - пытается отыскать имя X по правилу LEGB.

Имена атрибутов: пространства имен объектов

Квалифицированные имена атрибутов ссылаются на атрибуты конкретных объектов и к ним применяются правила, предназначенные для модулей и классов. Для объектов классов и экземпляров эти правила дополняются включением процедуры поиска в дереве наследования:

- **Присваивание** (object.X = value) Создает или изменяет атрибут с именем X в пространстве имен объекта object, и ничего больше.

Восхождение по дереву наследования происходит только при попытке получить ссылку на атрибут, но не при выполнении операции присваивания.

- **Ссылка** (object.X) Для объектов, созданных на основе классов, поиск атрибута X производится сначала в объекте object, затем во всех классах, расположенных выше в дереве наследования. В случае объектов, которые создаются не из классов, таких как модули, атрибут X извлекается непосредственно из объекта object.

Классификация имен происходит при =.

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)

if __name__ == '__main__':
    print(X)        # 11: модуль (за пределами файла manynames.X)
    f()             # 11: глобальная
    g()             # 22: локальная
    print(X)        # 11: переменная модуля не изменилась
    obj = C()       # Создать экземпляр
    print(obj.X)    # 33: переменная класса, унаследованная экземпляром
    obj.m()         # Присоединить атрибут X к экземпляру
    print(obj.X)    # 55: экземпляр
    print(C.X)      # 33: класс (она же obj.X, если в экземпляре нет X)
    #print(C.m.X)   # ОШИБКА: видима только в методе
    #print(g.X)     # ОШИБКА: видима только в функции
```

Теперь в другом файле:

```
# otherfile.py
import manynames
X = 66
print(X)            # 66: здешняя глобальная переменная
print(manynames.X)  # 11: глобальная, ставшая атрибутом в результате импорта
manynames.f()       # 11: X в manynames, не здешняя глобальная!
manynames.g()       # 22: локальная в функции, в другом файле
print(manynames.C.X) # 33: атрибут класса в другом модуле
I = manynames.C()
print(I.X)          # 33: все еще атрибут класса
I.m()
print(I.X)          # 55: а теперь атрибут экземпляра!
```

Добавим global и nonlocal:

```
X = 11              # Глобальная в модуле
def g1():
    print(X)        # Ссылка на глобальную переменную в модуле
def g2():
    global X
    X = 22          # Изменит глобальную переменную в модуле
def h1():
    X = 33          # Локальная в функции
    def nested():
        print(X)    # Ссылка на локальную переменную в объемлющей функции
def h2():
    X = 33          # Локальная в функции
    def nested():
        nonlocal X  # Инструкция из Python 3.0
        X = 44      # Изменит локальную переменную в объемлющей функции
```

Старайтесь не писать одинаковые имена в разных областях (контекстах)

Словари пространств имен (исследуем какие есть имена)

Пространства имен реализованы как словари и доступны по встроенному атрибуту **__dict__**

Аналогично объекты классов и экземпляров: обращение к квалифицированному имени - это операция доступа к элементу словаря.

```
>>> class super:
...     def hello(self):
...         self.data1 = 'spam'
...
>>> class sub(super):
...     def hola(self):
...         self.data2 = 'eggs'
...
```

- экземпляры по атрибуту **__class__** получают ссылку на класс;
- класс по атрибуту **__bases__** получает кортеж со ссылками на суперклассы.

```
>>> X = sub()
>>> X.__dict__              # Словарь пространства имен экземпляра
{}
>>> X.__class__             # Класс экземпляра
<class '__main__.sub'>
>>> sub.__bases__           # Суперклассы данного класса
(<class '__main__.super'>,)
>>> super.__bases__         # В Python 2.6 возвращает пустой кортеж ()
(<class 'object'>,)
```

Исследуем экземпляр подкласса:

```
>>> Y = sub()
>>> X.hello()
>>> X.__dict__
{'data1': 'spam'}
>>> X.hola()
>>> X.__dict__
{'data1': 'spam', 'data2': 'eggs'}
>>> sub.__dict__.keys()
['__module__', '__doc__', 'hola']
>>> super.__dict__.keys()
['__dict__', '__module__', '__weakref__', 'hello', '__doc__']
>>> Y.__dict__              # до сих пор пустой!!!
{}
```

Имена с __ определяются автоматически.

Можно доступиться к атрибуту по квалифицированному имени или индексироваться по ключу:

```
>>> X.data1, X.__dict__['data1']
('spam', 'spam')
>>> X.data3 = 'toast'
>>> X.__dict__
{'data1': 'spam', 'data3': 'toast', 'data2': 'eggs'}
>>> X.__dict__['data3'] = 'ham'
>>> X.data3
'ham'
```

Заметим, что унаследованный атрибут X.hello недоступен через `X.__dict__['hello']`

Функция **dir(object)** напоминает вызов **object.dict.keys()**, но:

- *dir* - сортирует свой список и включает в него системные атрибуты.
- собирает унаследованные атрибуты (с версии 2.2)
- вместе с атрибутами класса *object*

```
>>> X.__dict__, Y.__dict__
{ ({'data1': 'spam', 'data3': 'ham', 'data2': 'eggs'}, {})
>>> list(X.__dict__.keys())         # Необходимо в Python 3.0
['data1', 'data3', 'data2']
                                    # В Python 2.6
>>> dir(X)
['__doc__', '__module__', 'data1', 'data2', 'data3', 'hello', 'hola']
>>> dir(sub)
['__doc__', '__module__', 'hello', 'hola']
>>> dir(super)
['__doc__', '__module__', 'hello']
                                    # В Python 3.0:
>>> dir(X)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
...часть строк опущена...
'data1', 'data2', 'data3', 'hello', 'hola']
>>> dir(sub)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
...часть строк опущена...
'hello', 'hola']
>>> dir(super)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
...часть строк опущена...
'hello'
]
```

Строки документации (docstring)

Написаны в тройных кавычках.

Автоматически сохраняются интерпретатором в их объектах.

Сделаем блоки документации:

```
"""I am: docstr.__doc__"""
def func(args):
    """I am: docstr.func.__doc__"""
    pass
class spam:
    """I am: spam.__doc__ or docstr.spam.__doc__"""
    def method(self, arg):
        """I am: spam.method.__doc__ or self.method.__doc__"""
        pass
```

Доступ к объектам docstr:

```
>>> import docstr
>>> docstr.__doc__
'I am: docstr.__doc__'
>>> docstr.func.__doc__
'I am: docstr.func.__doc__'
>>> docstr.spam.__doc__
'I am: spam.__doc__ or docstr.spam.__doc__'
>>> docstr.spam.method.__doc__
'I am: spam.method.__doc__ or self.method.__doc__'
```

Сделаем из строк документации help:

```
>>> help(docstr)
Help on module docstr:
NAME
    docstr - I am: docstr.__doc__
FILE
    c:\misc\docstr.py
CLASSES
    spam
    class spam
    | I am: spam.__doc__ or docstr.spam.__doc__
    |
    | Methods defined here:
    |
    | method(self, arg)
    |   I am: spam.method.__doc__ or self.method.__doc__
FUNCTIONS
    func(args)
        I am: docstr.func.__doc__
```

Классы и модули

- Модули
  - Это пакеты данных и исполняемого кода.
  - Создаются как файлы с программным кодом на языке Python или как расширения на языке C.
  - Задействуются операцией import.
- Классы
  - Реализуют новые объекты.
  - Создаются с помощью инструкции class.
  - Задействуются операцией вызова.
  - Всегда располагаются внутри модуля. Кроме того, классы поддерживают дополнительные возможности, недоступные в модулях, такие как перегрузка

операторов, создание множества экземпляров и наследование. Несмотря на то что и классы, и модули являются пространствами имен, между ними есть различия.

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>}
```

Перегрузка операторов

Мы можем легко найти минимальное число, сумму чисел или отсортировать последовательность:

```
>>> a = [3, 7, -1, 10]
>>> min(a)
-1
>>> sorted(a)
[-1, 3, 7, 10]
>>> sum(a)
19
```

Хочется, чтобы эти функции работали и для классов. Для этого нужно, чтобы классы умели сравнивать < (для min и sorted), и складывать через + (для sum).

Мы научились, чтобы `print` могла печатать экземпляры класса. Для этого мы в классе пишем функцию **__str__(self)**. Если есть класс A, и `x = A()` - экземпляр класса A, то при `print(x)` вызывается `str(x)`, которая вызывает `x.__str__()` `print([x])` вызовет функцию `repr(x)`, которая вызовет `x.__repr__()`

То есть, если мы определим в классе функцию со специальным именем, он будет вызываться при выполнении операций над экземплярами класса.

Можно переопределить в классе функции, чтобы работали операторы:

Специальная функция	Оператор	Значение
__lt__(self, other)	x < y	Возвращает True, если x меньше, чем y
__le__(self, other)	x <= y	Возвращает True, если x меньше или равно y
__eq__(self, other)	x == y	Возвращает True, если x равно y
__ne__(self, other)	x != y	Возвращает True, если x НЕ равно y
__gt__(self, other)	x > y	Возвращает True, если x больше, чем y
__ge__(self, other)	x >= y	Возвращает True, если x больше или равно y

Специальная функция	Оператор	Коментарий
__add__(self, other)	x + y	
__sub__(self, other)	x - y	
__mul__(self, other)	x * y	
__truediv__(self, other)	x / y	
__floordiv__(self, other)	x // y	
__mod__(self, other)	x % y	
__pow__(self, other)	x ** y	

Вспомним умножение строки на число 'hi'*3. Можно написать 3*'hi', получим такой же результат.

Для того, чтобы написать функцию число * строку, нужно переопределить для строки метод __rmul__ .

```
some_object + other
```

Вызывает __add__()

```
other + some_object
```

Вызывает __radd__(). У нее первый операнд other, а второй self.

Ее можно реализовать как:

```
def __radd__(self, other):
    return __add__(other, self)
```

Пример с точкой на плоскости XY

В классе Point (точка на плоскости XY) переопределим функции для == и для <

```
class Point(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return '({} {})'.format(self.x, self.y)
```

```
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __lt__(self, other):
        """ Меньше та точка, у которой меньше x. При одинаковых x, та, у которой меньш
e y."""
        if self.x == other.x:
            return self.y < other.y
        return self.x < other.x

    # другие функции класса: move, dir, dist...

# Тестируем функции класса:
def test():
    p0 = Point(3, 5)
    p1 = Point(3, 5)
    p2 = Point(-1, 7)
    p3 = Point(3, 1.17)

    print('p0=', p0)       # 3 5
    print('p1=', p1)       # 3 5
    print('p2=', p2)       # -1 7
    print('p3=', p3)       # 3 1.17

    print('p0 == p1', p0 == p1) # True
    assert(p0 == p1)
    print('p1 == p2', p1 == p2) # False
    assert(not p1 == p2)

    print('p0 != p1', p0 != p1) # False
    assert(not(p0 != p1))
    print('p1 != p2', p1 != p2) # True
    assert(p1 != p2)

    print('p2 < p1', p2 < p1)   # True
    assert(p2 < p1)
    print('p1 < p2', p1 < p2)   # False
    assert(not(p1 < p2))

    print('p3 < p1', p3 < p1)   # True
    assert(p3 < p1)
    print('p1 < p3', p1 < p3)   # False
    assert(not (p1 < p3))

    a = [p0, p1, p2, p3]
    pmin = min(a)
    print('pmin =', pmin)
    assert(p2 == pmin)

    b = sorted(a)
    print(b)
    assert(b == [p2, p3, p0, p1])
```

```
test()
```

Прочие методы (таблица)

Метод	Перегружает	Вызывается
__init__	Конструктор	При создании объекта: `X = Class(args)`
__del__	Деструктор	При уничтожении объекта
__add__	Оператор +	X + Y, X += Y, если отсутствует метод __iadd__
__or__	Оператор (побитовое ИЛИ)	`X \	Y , X \	= Y`, если отсутствует метод __ior__
__repr__ __str__	Вывод, преобразование	print(X), repr(X), str(X)
__call__	Вызовы функции	`X(*args, **kargs)`
__getattr__	Обращение к атрибуту	X.undefined
__setattr__	Присваивание атрибуту	X.any = value
__delattr__	Удаление атрибута	del X.any
__getattribute__	Обращение к атрибуту	X.any
__getitem__	Доступ к элементу по индексу, извлечение среза, итерации	X[key], X[i:j], циклы for и другие конструкции итерации, при отсутствии метода __iter__
__setitem__	Присваивание элементу по индексу или срезу	X[key] = value, X[i:j] = sequence
__delitem__	Удаление элемента по индексу или среза	del X[key], del X[i:j]

__len__	Длина	len(X), проверка истинности, если отсутствует метод __bool__
__bool__	Проверка логического значения	bool(X) , проверка истинности (в версии 2.6 называется __nonzero__)
__radd__	Правосторонний оператор +	Не_экземпляр + X
__iadd__	Добавление (увеличение)	X += Y (в ином случае __add__)
__iter__, __next__	Итерационный контекст	I=iter(X), next(I); циклы for, оператор in (если не определен метод __contains__), все типы генераторов, map(F, X) и другие (в версии 2.6 метод __next__ называется next))
__contains__	Проверка на вхождение	item in X (где X – любой итерируемый объект)
__index__	Целое число	hex(X), bin(X) , oct(X) , O[X] , O[X:] (замещает методы __oct__, __hex__ в Python 2)
__enter__, __exit__	Менеджеры контекстов TODO: ссылку на раздел	`with obj as var:`
__get__, __set__, __delete__	Дескрипторы атрибутов	X.attr, X.attr = value, del X.attr
__new__	Создание	Вызывается при создании объектов, перед вызовом метода __init__

Обращение к элементу по индексу и итерации

Доступ к элементам по индексу и извлечение срезов: __getitem__ и __setitem__

Класс возвращает квадрат значения индекса:

```
>>> class Indexer:
...     def __getitem__(self, index):
...         return index ** 2
...
>>> X = Indexer()
>>> X[2]                    # Выражение X[i] вызывает X.__getitem__(i)
4
>>> for i in range(5):
...     print(X[i], end=' ') # Вызывает __getitem__(X, i) в каждой итерации
...
0 1 4 9 16
```

Объект slice

При указании среза передается объект slice.

`L = [5, 6, 7, 8, 9]`

Срез	Объект slice	Результат
`L[2:4]`	`L[slice(2, 4)]`	`[7, 8]`
`L[1:]`	`L[slice(1, None)]`	`[6, 7, 8, 9]`
`L[:-1]`	`L[slice(None, -1)]`	`[5, 6, 7, 8]`
`L[::2]`	`L[slice(None, None, 2)]`	`[5, 7, 9]`

```
>>> class Indexer(object):
        data = [5, 6, 7, 8, 9]
        def __setitem__(self, index, value):    # Реализует присваивание
...                                             # по индексу или по срезу
            self.data[index] = value            # Присваивание по индексу или по срезу
...     def __getitem__(self, index):           # Вызывается при индексировании или
...         print('getitem:', index)            # извлечении среза
...         return self.data[index]             # Выполняет индексирование
...                                             # или извлекает срез
>>> X = Indexer()
>>> X[0]                # При индексировании __getitem__ получает целое число
getitem: 0
5
>>> X[1]
getitem: 1
6
>>> X[-1]
getitem: -1
9
>>> X[2:4]              # При извлечении среза __getitem__ получает объект среза
getitem: slice(2, 4, None)
[7, 8]
>>> X[1:]
getitem: slice(1, None, None)
[6, 7, 8, 9]
>>> X[:-1]
getitem: slice(None, -1, None)
[5, 6, 7, 8]
>>> X[::2]
getitem: slice(None, None, 2)
[5, 7, 9]
```

До Python 3.0 были отдельные методы _getslice__, __setslice__. Сейчас вместо них используют __getitem__, __setitem__ (в них добавлена обработка объектов-срезов)_

Преобразование в целое число __index__

Иногда объект могут использовать как индекс. Для работы такого синтаксиса переопределите __index__

```
>>> class C:
...     def __index__(self):
...         return 255
...
>>> X = C()
>>> hex(X)          # Целочисленное значение
'0xff'
>>> bin(X)
'0b11111111'
>>> oct(X)
'0o377'
>>> ('C' * 256)[255]
'C'
>>> ('C' * 256)[X]  # X используется как индекс (не X[i])
'C'
>>> ('C' * 256)[X:] # X используется как индекс (не X[i:])
'C'
```

В Python 2.6 были отдельные методы __hex__ и __oct__.

Итерация по индексам __getitem__

Операция **for** использует операцию индексирования к последовательности, где индексы от 0 и выше, пока не выйдет за границу последовательности (исключение, for его сам обработает). То есть __getitem__ - один из способов реализовать перебор в for.

Что реализовано с использованием __getitem__:

- **for**
- **in**
- **map** (функция)
- генераторы списков (кортежей)
- присваивание списов (кортежей)

```
>>> class stepper:
...     def __getitem__(self, i):
...         return self.data[i]
...
>>> X = stepper()       # X - это экземпляр класса stepper
>>> X.data = 'Spam'
>>>
>>> X[1]                # Индексирование, вызывается __getitem__
'p'
>>> for item in X:      # Циклы for вызывают __getitem__
...     print(item, end=' ')    # Инструкция for индексирует элементы 0..N
...
S p a m
>>> 'p' in X            # Во всех этих случаях вызывается __getitem__
True
>>> [c for c in X]      # Генератор списков
['S', 'p', 'a', 'm']
>>> list(map(str.upper, X)) # Функция map (в версии 3.0
['S', 'P', 'A', 'M']    # требуется использовать функцию list)
>>> (a, b, c, d) = X    # Присваивание последовательностей
>>> a, c, d
('S', 'a', 'm')
>>> list(X), tuple(X), ''.join(X)
(['S', 'p', 'a', 'm'], ('S', 'p', 'a', 'm'), 'Spam')
>>> X
<__main__.stepper instance at 0x00A8D5D0>
```

Итераторы __iter__ и __next__

На самом деле, сначала пытается вызваться __iter__, а если его нет, то метод __getitem__

Как перебирается последовательность:

- вызывается iter(), которая вызывает __iter__() - возвращает объект итератора.
  - вызывается метод next(it) (который вызывает it.__next__()), пока не получим исключение StopIteration.
- иначе вызывается getitem(), пока не получим исключение IndexError.

Можно определить свои итераторы

Можно определить итерацию по циклу (уже есть в `itertools.cycle`

```
from itertools import cycle

li = [0, 1, 2, 3]

running = True
licycle = cycle(li)
while running:
    elem = next(licycle)
```

Можно возвращать квадраты индексов:

```
class Squares:
    def __init__(self, start, stop): # Сохранить состояние при создании
        self.value = start - 1
        self.stop = stop
    def __iter__(self):              # Возвращает итератор в iter()
        return self
    def __next__(self):              # Возвращает квадрат в каждой итерации
        if self.value == self.stop: # Также вызывается функцией next
            raise StopIteration
        self.value += 1
        return self.value ** 2
```

использование:

```
>>> from iters import Squares
>>> for i in Squares(1, 5):         # for вызывает iter(), который вызывает __iter__()
...     print(i, end=' ')           # на каждой итерации вызывается __next__()
...
1 4 9 16 25
>>> X = Squares(1, 5)               # Выполнение итераций вручную: эти действия выполн
яет
                                    # инструкция цикла
>>> I = iter(X)                     # iter вызовет __iter__
>>> next(I)                         # next вызовет __next__
1
>>> next(I)
4
...часть строк опущена...
>>> next(I)
25
>>> next(I)                         # Исключение можно перехватить с помощью инструкци
и try
StopIteration
```

Заметим, что реализация такого поведения через __getitem__ будет сложнее (индексы start..stop придется отображать на 0..stop-strart)

__iter__ предназначена для обхода **один раз**. __getitem__ - для множественного обращения к элементу.

Квадраты проще реализовать через написание генератора, а не через определение нового класса с итератором.

```
>>> def gsquares(start, stop):
... for i in range(start, stop+1):
... yield i ** 2
...
>>> for i in gsquares(1, 5): # или: (x ** 2 for x in range(1, 5))
... print(i, end=' ')
...
1 4 9 16 25
>>> [x ** 2 for x in range(1, 6)]
[1, 4, 9, 16, 25]
```

Несколько итераторов в одном объекте

В строках можно сделать несколько итераторов по одной и той же строке. Заметим, что они работают независимо:

```
>>> S = ‘ace’
>>> for x in S:
... for y in S:
... print(x + y, end=’ ‘)
...
aa ac ae ca cc ce ea ec ee
```

- как внешний, так и внутренний цикл получает *свой* итератор, вызывая `iter()`
- каждый итератор хранит свою информацию о положении в строке (не зависит от других циклов).
- Однократные проходы:
  - функции-генераторы и выражения-генераторы;
  - map, zip
- Многократные (независимые) проходы:
  - range
  - list, tuple и тп

Для независимых итераторов __iter__() должен не просто возвращать self, а создавать новый объект со своей информацией о состоянии.

```
class SkipIterator:
    def __init__(self, wrapped):
        self.wrapped = wrapped              # Информация о состоянии
        self.offset = 0
    def next(self):
        if self.offset >= len(self.wrapped): # Завершить итерации
            raise StopIteration
        else:
            item = self.wrapped[self.offset] # Иначе перешагнуть и вернуть
            self.offset += 2
            return item

class SkipObject:
    def __init__(self, wrapped):            # Сохранить используемый элемент
        self.wrapped = wrapped
    def __iter__(self):
        return SkipIterator(self.wrapped)   # Каждый раз новый итератор

if __name__ == '__main__':
    alpha = 'abcdef'
    skipper = SkipObject(alpha)             # Создать объект-контейнер
    I = iter(skipper)                       # Создать итератор для него
    print(next(I), next(I), next(I))        # Обойти элементы 0, 2, 4
    for x in skipper:                       # for вызывает __iter__ автоматически
        for y in skipper:                   # Вложенные циклы for также вызывают __ite
r__
            print(x + y, end=' ')           # Каждый итератор помнит свое состояние, с
мещение
```

Напечатает:

```
a c e
aa ac ae ca cc ce ea ec ee
```

Сравним с кодом, который использует уже существующие инструменты:

```
>>> S = 'abcdef'
>>> S = S[::2]                              # Новый объект
>>> for x in S:
...     for y in S:
...         print(x + y, end=' ')
...
aa ac ae ca cc ce ea ec ee
```

- Код уже написан и отлажен (плюс).
- Создаются новые объекты (срезы), а не честная итерация в том же объекте (минус).

Где нужно писать такие множественные итераторы? Итерации по выборке из БД.

Проверка на вхождение: __contains__, __iter__, __getitem__

Проверку на вхождение можно организовать через __iter__ или __getitem__, но лучше реализовать специальный метод __contains__.

Посмотрим, что когда вызывается (для этого сделаем класс на с хорошо итерируемыми данными, например, list):

```
class Iters:
    def __init__(self, value):
        self.data = value
    def __getitem__(self, i):           # Крайний случай для итераций
        print('get[%s]:' % i, end='')   # А также для индексирования и срезов
        return self.data[i]
    def __iter__(self):                 # Предпочтительный для итераций
        print('iter=> ', end='')        # Возможен только 1 активный итератор
        self.ix = 0
        return self
    def __next__(self):
        print('next:', end='')
        if self.ix == len(self.data):
            raise StopIteration
        item = self.data[self.ix]
        self.ix += 1
        return item
    def __contains__(self, x):          # Предпочтительный для оператора 'in'
        print('contains: ', end='')
        return x in self.data

X = Iters([1, 2, 3, 4, 5])      # Создать экземпляр
print(3 in X)                   # Проверка на вхождение
for i in X:                     # Циклы
    print(i, end=' | ')
print()
print([i ** 2 for i in X])      # Другие итерационные контексты
print( list(map(bin, X)) )
I = iter(X)                     # Обход вручную (именно так действуют
while True:                     # другие итерационные контексты)
    try:
        print(next(I), end=' @ ')
    except StopIteration:
        break
```

Напечатает:

```
contains: True
iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]
iter=> next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100', '0b101']
iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:
```

Теперь закомментируем метод __contains__ и запустим код еще раз:

```
iter=> next:next:next:True
iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]
iter=> next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100', '0b101']
iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:
```

Видим `iter=> next:next:next:True`, что проверка делается через __iter__.

Если закомментируем и __contains__, и __iter__, то будет вызываться __getitem__:

```
get[0]:get[1]:get[2]:True
get[0]:1 | get[1]:2 | get[2]:3 | get[3]:4 | get[4]:5 | get[5]:
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:[1, 4, 9, 16, 25]
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:['0b1', '0b10', '0b11', '0b100','0b101']
get[0]:1 @ get[1]:2 @ get[2]:3 @ get[3]:4 @ get[4]:5 @ get[5]:
```

Атрибуты и ограничение доступа через переопределение операторов

Обращение к атрибутам __getattr__ и __setattr__

__getattr__ - получить ссылку на атрибут.

НЕ вызывается, если интерпретатор может найти атрибут в дереве наследования.

Вызывается, если пытается получить ссылку на *неопределенный* (несуществующий) атрибут. Атрибут вычисляется **динамически**.

Сделаем класс, который ведет себя так, как у его экземпляров есть атрибут age и нет других атрибутов.

```
>>> class empty:
... def __getattr__(self, attrname):
...     if attrname == 'age':
...         return 40
...     else:
...         raise AttributeError, attrname
...
>>> x = empty()
>>> x.age
40
>>> x.name
...текст сообщения об ошибке опущен...
AttributeError: name
```

__setattr__ - вызывается **всегда**, когда пишем `self.atr = value` .

Вызывается `self.__setattr__('atr', value)`

Если внутри этого метода нужно сделать `self.name = x` , то не пишите так (получите бесконечный цикл рекурсивных вызовов), а пишите `self.__dict__['name'] = x`

```
>>> class accesscontrol:
... def __setattr__(self, attr, value):
...     if attr == 'age':
...         self.__dict__[attr] = value
...     else:
...         raise AttributeError, attr + ' not allowed'
...
>>> x = accesscontrol()
>>> x.age = 40              # Вызовет метод __setattr__
>>> x.age
40
>>> x.name = 'mel'
...текст сообщения об ошибке опущен...
AttributeError: name not allowed
```

Как еще управлять атрибутами (см. позже):

- __getattribute__ - обращение к любым атрибутам (даже уже существующим). Пишем обращение к атрибуту так же, как в __setattr__, чтобы избежать рекурсии: `self.__dict__['name'] = x`
- функция `property` ()
- *дескрипторы* связывают методы __get__ и __set__ с доступом к нужным атрибутам класса.

Пример: ограничиваем права доступа через __setattr__

```
class PrivateExc(Exception): pass               # Создали пользовательское исключение
class Privacy:
    def __setattr__(self, attrname, value):     # Вызывается self.attrname = value
        if attrname in self.privates:
            raise PrivateExc(attrname, self)
        else:
            self.__dict__[attrname] = value     # ибо self.attrname = value будет реку
рсивно вызывать __setattr__

class Test1(Privacy):
    privates = ['age']

class Test2(Privacy):
    privates = ['name', 'pay']
    def __init__(self):
        self.__dict__['name'] = 'Tom'

x = Test1()
y = Test2()
x.name = 'Bob'
y.name = 'Sue'          # <== ошибка
y.age = 30
x.age = 40              # <== ошибка
```

Полная реализация будет позже.

__repr__ и __str__ - представим объект в виде строки

Эти методы должны вернуть строку

Эти методы вызываются, когда объект пытаются преобразовать в строку:

```
>>> class A:
...     def __str__(self):  return 'call str'
...     def __repr__(self): return 'call repr'
...
>>> x = A()
>>> x
call repr
>>> b = [x]
>>> b
[call repr]
>>> print(x)
call str
>>> str(x)
'call str'
>>> repr(x)
'call repr'
```

- **__str__** - пытается вызваться при print() и str(); показывает объект для пользователя.
- **__repr__** - в остальных случаях; содержит полезный для разработчика код (для отладки) или строку, которую можно выполнить и воссоздать этот объект.

Если метода __str__ нет, то будет пытаться вызваться метод __repr__. Но не наоборот, если есть только __repr__, то не будут пытаться вместо нее вызвать __str__ (Вдруг нужно полученную строку потом будет выполнять, а результат str выполнять не предполагается).

Арифметические операции

Рассмотрим арифметические операции на примере сложения. Возможные варианты сложения:

```
x + y       # x и y - объекты одного класса
x + 12      # складываем объект с другим типом
12 + x      # складываем другой тип с объектом
x += y      # сложение с присваиванием
```

__add__, __radd__, __iadd__

Чем отличаются эти методы и какие операции реализуют?

Пусть в классе Drob реализован метод __add__, который складывает две дроби.

Что делать, если хотим сложить дробь с числом опишем потом.

Что делать, если хотим сложить число с дробью: `3 + Drob(1, 2)`. Метод __add__ класса Drob будет вызван, если левый операнд - дробь. Если дробь прибавляем к int, то должен вызываться метод __add__ класса int (который ничего не знает о классе Drob).

Метод **__radd__** класса вызывается, когда экземпляр класса находится справа от +, а слева от + не является экземпляром класса.

```
>>> class Commuter:
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):
...         print('add', self.val, other)
...         return self.val + other # если other экземпляр этого же класса, то получим
 вызов __radd__
...     def __radd__(self, other):
...         print('radd', self.val, other)
...         return other + self.val
...
>>> x = Commuter(88)
>>> y = Commuter(99)
>>> x + 1 # __add__: экземпляр + не_экземпляр
add 88 1
89
>>> 1 + y # __radd__: не_экземпляр + экземпляр
radd 99 1
100
>>> x + y # __add__: экземпляр + экземпляр
add 88 <__main__.Commuter instance at 0x02630910>
radd 99 88
187
```

Разберем порядок вызовов при x+y. Так как x слева и у него есть метод __add__, вызывается этот метод.

При вызове метода __add__ доходим до вычисления `self.val + other` и self.val тут число, а other - объект класса, значит вызывается `other.__radd(self.val)` .

Если результатом операции должен быть экземпляр класса, то его нужно вернуть. Если мы пишем неизменяемый тип, то создать новый экземпляр.

Заметьте, если не проверить в __add__, что other тоже экземпляр класса, то получим бесконечную рекурсию:

```
>>> class Commuter: # Тип класса распространяется на результат
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):
...         if isinstance(other, Commuter): other = other.val
...             return Commuter(self.val + other)
...     def __radd__(self, other):
...         return Commuter(other + self.val)
...     def __str__(self):
...         return '<Commuter: %s>' % self.val
...
>>> x = Commuter(88)
>>> y = Commuter(99)
>>> print(x + 10)       # Результат – другой экземпляр класса Commuter
<Commuter: 98>
>>> print(10 + y)
<Commuter: 109>
>>> z = x + y           # Нет вложения: не происходит рекурсивный вызов __radd__
>>> print(z)
```

Какие методы будут вызваны, если в классе Drob1 реализован метод __add__, в классе Drob2 реализован метод __radd__, и вычисляется выражение Drob1(1, 2) + Drob2(1, 2) (если классы не связаны отношением наследования)? Вызывается метод __add__ для **левого** операнда.

Правосторонние методы реализуют, если в класс должен поддерживать перестановку операндов.

Классу Drob такой метод нужен. Классу Student - скорее всего нет.

+= - изменение объекта

Оператор += реализован через метод __iadd__. Если его нет, вызывается __add__.

В __iadd__ можно изменять сам объект, не создавая новые объекты (как list.extend).

```
>>> class Number1:
...     def __init__(self, val):
...         self.val = val
...     def __iadd__(self, other):  # __iadd__ явно реализует операцию x += y
...         self.val += other       # Обычно возвращает self
...         return self
...
>>> x = Number1(5)
>>> x += 1
>>> x += 1
>>> x.val
7
>>> class Number2:
... def __init__(self, val):
...     self.val = val
... def __add__(self, other):       # __add__ - как крайнее средство: x=(x + y)
...     return Number2(self.val + other) # Распространяет тип класса
...
>>> x = Number2(5)
>>> x += 1
>>> x += 1
>>> x.val
7
```

Другие арифметические операции имеют аналогичный набор методов.

__call__

Вызывается, когда мы обращаемся к **экземпляру класса** как к **функции**.

Передаются любые позиционные и именованные аргументы.

```
>>> class Callee:
...     def __call__(self, *pargs, **kargs):    # Реализует вызов экземпляра
...         print('Called:', pargs, kargs)      # Принимает любые аргументы
...
>>> C = Callee()
>>> C(1, 2, 3)                                  # C – вызываемый объект
Called: (1, 2, 3) {}
>>> C(1, 2, 3, x=4, y=5)
Called: (1, 2, 3) {'y': 5, 'x': 4}
```

Можно реализовать метод __call__ одним из способов:

```
class C:
    def __call__(self, a, b, c=5, d=6): ...     # Обычные и со значениями по умолчанию
class C:
    def __call__(self, *pargs, **kargs): ...    # Произвольные аргументы
class C:
    def __call__(self, *pargs, d=6, **kargs): ... # Аргументы, которые могут передават
ься
                                                 # только по имени в версии 3.0
```

и при этом любая вышеописанная реализация __call__ подойдет для следующих вызовов:

```
X = C()
X(1, 2)                         # Аргументы со значениями по умолчанию опущены
X(1, 2, 3, 4)                   # Позиционные
X(a=1, b=2, d=4)                # Именованные
X(*[1, 2], **dict(c=3, d=4))    # Распаковывание произвольных аргументов
X(1, *(2,), c=3, **dict(d=4))   # Смешанные режимы
```

Очень распространненый метод. Экземпляры класса имитируют поведение функции, а так же могут сохранять свое состояние между вызовами.

```
>>> class Prod:
...     def __init__(self, value): # Принимает единственный аргумент
...         self.value = value
...     def __call__(self, other):
...         return self.value * other
...
>>> x = Prod(2)                    # "Запоминает" 2 в своей области видимости
>>> x(3)                           # 3 (передано) * 2 (сохраненное значение)
6
>>> x(4)
8
```

Callback

Библиотека для GUI tkinter позволяет регистрировать функции как event handler (обработчики событий), они же callback: возникло событие - tkinter вызывает зарегистрированные объекты.

Если нужно, чтобы обработчик событий запоминал свое состояние между вызовами (в терминах C - имел статическую переменную), то регистрируем либо связанный метод класса (следующая глава), либо экземпляр класса, который можно вызвать как функцию с нужными параметрами благодаря методу __call__.

В примерах `x.comp` (первый пример) и `x` (второй пример) могут передаваться как объекты функций.

Кнопка при нажатии должна окрашиваться в цвет. Этот цвет хранится в объекте класса. Этот объект может вызываться как функция.

```
class Callback:
    def __init__(self, color):      # Функция + информация о состоянии
        self.color = color
    def __call__(self):             # Поддерживает вызовы без аргументов
        print('turn', self.color)
```

GUI сделан так, что обработчики событий вызываются как функции без аргументов. Зарегистрируем объекты класса Callback, как обработчики событий:

```
cb1 = Callback('blue')      # 'Запомнить' голубой цвет
cb2 = Callback('green')
b1 = Button(command=cb1)    # Зарегистрировать обработчик
b2 = Button(command=cb2)    # Зарегистрировать обработчик
```

Когда кнопка будет нажата, cb1 и cb2 будут вызваны как функции без аргументов. Так как эти объекты сохранили цвет в атрибутах экземпляра класса, то будет использован нужный цвет:

```
cb1() # По событию: выведет 'turn blue'
cb2() # Выведет 'turn green'
```

Лучше сохранять информацию так, чем использовать глобальные переменные, ссылки в область видимости объемлющей функции и изменяемые аргументы со значениями по умолчанию.

Аргументы со значением по умолчанию (пример):

```
cb3 = (lambda color='red': 'turn ' + color) # Или: по умолчанию
print(cb3())
```

Связанные методы класса. Запоминается экземпляр self + ссылка на функцию. Потом вызваем функцию без использования экземпляра. (пример):

```
class Callback:
    def __init__(self, color):  # Класс с информацией о состоянии
        self.color = color
    def changeColor(self):      # Обычный метод экземпляра класса
        print('turn', self.color)
cb1 = Callback('blue')
cb2 = Callback('yellow')
# ссылка, а не вызов функции; запоминаются функция + self:
b1 = Button(command=cb1.changeColor)
b2 = Button(command=cb2.changeColor)
```

При нажатии кнопки имитируется поведение GUI и вызывается метод changeColor. Он обрабатывает информацию о состоянии объекта:

```
object = Callback('blue')
cb = object.changeColor         # Регистрация обработчика событий
cb()                            # По событию выведет 'blue'
```

В следующей главе о связанных методах подробнее. Еще позже будет рассказано об использовании __call__ для реализации декоратора функции.

Сравнение

- В отличие от __add__ и __radd__ методы сравнения не имеют правосторонних версий. Если операцию поддерживает только один операнд, используется зеркальный метод сравнения (методы __lt__ и __gt__ зеркальны друг другу)
- Не факт, что истинность операции == означает ложность !=. В этом случае реализуйте оба метода: __eq__ и __ne__.
- Метод __cmp__ и встроенный метод cmp() устарели. Не используйте их.

__bool__ и __len__

- Сначала интерпретатор пытаеся вызвать __bool__. Если не находит, пытается вызвать __len__ и проверить результат на 0 (это будет False).
- Если не реализован ни один метод, объект считается **True**

__del__ - уничтожение объектов

Вызывается при вызове сборщика мусора для объекта. (Неизвестно когда будет вызван).

То есть может быть ситуация, что за все время работы программы __del__ не вызвали.

Поэтому если нужно, чтобы гарантированно очистка произошла, пишите try/finally блок.

- Что будет, если внутри __del__ возникнет исключение? Плохо будет, потому что сборщик мусора уже начал свою работу.
- циклические ссылки собираются автоматически, если в них **нет** метода __del__

Подробнее далее (TODO)

- ссылка на пример по __enter__ и __exit__ - менеджеры контеста с with;
- ссылка на пример по __get__ и __set__ - дескрипторы для методов чтения и записи значений атрибутов класса;
- ссылка на пример с исползованием __new__ в контексте метакласса.

Лутц, стр 826.

Контрольные вопросы

1. Какие два метода перегрузки операторов можно использовать для поддержки итераций в классах?
2. Какие два метода перегрузки операторов можно использовать для вывода и в каких случаях?
3. Как реализовать в классе операции над срезами?
4. Как реализовать в классе операцию приращения значения самого объекта?
5. Когда следует использовать методы перегрузки операторов?

Ответы

1. Классы могут обеспечить поддержку итераций, определив (или унаследовав) метод __getitem__ или __iter__. Во всех итерационных контекстах интерпретатор Python сначала пытается использовать метод __iter__ (который возвращает объект, поддерживающий итерационный протокол в виде метода __next__): если метод __iter__ не будет найден в результате поиска по дереву наследования, интерпретатор возвращается к использованию метода извлечения элемента по его индексу __getitem__ (который вызывается многократно и при каждом вызове получает постоянно увеличивающиеся значения индексов).

2. Вывод объекта реализуют методы __str__ и __repr__. Первый из них вызывается встроенными функциями print и str; последний также вызывается функциями print и str, если в классе отсутствует метод __str__, и всегда - встроенной функцией repr, функцией автоматического вывода интерактив- ной оболочки и при выводе вложенных экземпляров. То есть метод __repr__ используется везде, за исключением функций print и str, если определен метод __str__. Метод __str__ обычно используется для вывода объектов в удобочитаемом представлении; метод __repr__ выводит дополнительные подробности об объекте или представляет объект в виде программного кода.

3. Операцию извлечения среза можно перехватить с помощью метода __getitem__: в этом случае ему передается не простой числовой индекс, а объект среза. В Python 2.6 точно так же можно использовать метод __getslice__ (в версии 3.0 он уже не используется).

4. Операция приращения пытается сначала вызвать метод __iadd__, а затем метод __add__ с последующим присваиванием. Тот же самый прием может применяться для всех двухместных операторов. Кроме того, правостороннее сложение можно реализовать с помощью метода __radd__.

5. Когда класс естественным образом соответствует встроенным типам или должен подражать их поведению. Например, классы коллекций могут имитировать поведение последовательностей или отображений. Как правило, не следует реализовать методы перегрузки операторов, если они не являются естественными для ваших объектов, – лучше использовать методы с обычными именами.

Задачи

Напишите класс Drob, который представляет дроби в виде *целых* числителя и знаменателя.

```
class Drob(object):
    """ Дробь вида a/b"""
    def __init__(self, a=0, b=1):
        self.a = a
        self.b = b
        self.normalize()

    def normalize(self):
        """ Приводит дробь вида 4/6 к 2/3"""

    def __str__(self):
        return '{}/{}'.format(self.a, self.b)

    # реализуйте функции
    # __eq__
    # __lt__
    # __add__
    # __sub__
    # __mul__
    # __truediv__
    # __floordiv__
    # __mod__
    # и ПРОВЕРЬТЕ каждую функцию
```

Этот класс нам понадобится, когда будем изучать тестирование. Сохраните его, пожалуйста.

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

Шаблоны проектирования

Как использовать классы для решения задач (архитектура)?

- наследование
- композиция
- делегирование
- фабрики

Три кита ООП:

- наследование Основано на механизме поиска атрибутов объекта в выражении `x.atr`
- полиморфизм Значение выражения `x.method` зависи от того, какого типа объект x.
- инкапсуляция по умолчанию данные спрятаны, поведение объекта реализуем через методы.

Перегрузка (overload) функций

Так как в питоне нет объявления типов (в агрументах функций объявлений точно нет), то нет и обычной для других языков (C++, Java) перегрузки, когда реализуем методы с одинаковым именем, но разным набором аргументов (различается количество и/или тип аргументов).

```
class C:
    def meth(self, x):
        ...
    def meth(self, x, y, z):
        ...
```

Это работоспобоный код. У нас выполняется 2 присваивания. Один за другим. Все равно, что сначала написать x = 1, а потом x = 2. (Определение функции - это тоже операция присвоения).

isinstance

Можно реализовать через *args и проверку **isinstance**

```
>>> x = 3
>>> isinstance(x, int)
True
>>> isinstance(x, str)
False
```

isinstance определяет и базовые классы в наследованииё

```
>>> class A(object):
      pass
...
>>> class B(A):
       pass
...
>>> x = B()
>>> x
<__main__.B object at 0xffd77f70>
>>> isinstance(x, A)
True
>>> isinstance(x, B)
True
>>> isinstance(x, int)
False
```

Лучше не писать проверки аргументов функции на тип, а использовать полиморфизм:

```
class C:
    def meth(self, x):
        x.operation()   # Предполагается, что x работает правильно
```

Вызов методов базового класса

Надо вызвать метод базового класса из метода, который переопределен в производном классе.

Из конструктора дочернего класса нужно *явно* вызывать конструктор родительского класса.

Обращение к базовому классу происходит с помощью **super()**

Нужно явно вызывать конструктор базового класса

```
class A(object):
    def __init__(self, x=5):
        print('A.__init__')
        self.x = x

class B(A):
    def __init__(self, y=2):
        print('B.__init__')
        self.y = y

k = B(7)                    # B.__init__

print('k.y =', k.y)         # k.y = 7
#print('k.x =', k.x)        # AttributeError: 'B' object has no attribute 'x'
```

Видно, что без явного вызова конструктора класса A не вызывается `A.__init__` и не создается поле x класса A.

Вызовем конструктор явно.

Конструктор базового класса стоит вызывать раньше, чем инцилизировать поля класса-наследника, потому что поля наследника могут зависеть (быть сделаны из) полей экземпляра базового класса.

```
class A(object):
    def __init__(self, x=5):
        print('A.__init__')
        self.x = x

class B(A):
    def __init__(self, y=2):
        print('B.__init__')
        super().__init__(y/2)
        self.y = y

k = B(7)                    # B.__init__
                            # A.__init__
print('k.y =', k.y)         # k.y = 7
print('k.x =', k.x)         # k.x = 3.5
```

super() или прямое обращение к классу?

Метод класса можно вызвать, используя синтаксис вызова через имя класса:

```
class Base(object):
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')

k = A()                     # Base.__init__
                            # A.__init__
```

Все работает. Но при дальнейшем развитии классов могут начаться проблемы:

```
class Base(object):
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')

class B(Base):
    def __init__(self):
        Base.__init__(self)
        print('B.__init__')

class C(A, B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        print('C.__init__')

x = C()                     # Base.__init__
                            # A.__init__
                            # Base.__init__ - второй вызов
                            # B.__init__
                            # C.__init__
```

Видно, что конструктор `Base.__init__` вызывается дважды. Иногда это недопустимо (считаем количество созданных экземпляров класса, увеличивая в конструкторе счетчик на 1; выдаем очередное auto id какому-то нашему объекту, например, номер пропуска или паспорта или номер заказа).

То же самое через super():

- вызов конструктора Base.__init__ происходит только 1 раз.
- вызваны конструкторы всех базовых классов.
- порядок вызова конструкторов для классов A и B не определен.

```
class Base(object):
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        super().__init__()
        print('A.__init__')

class B(Base):
    def __init__(self):
        super().__init__()
        print('B.__init__')

class C(A, B):
    def __init__(self):
        super().__init__()
        print('C.__init__')

x = C()                    # Base.__init__
                           # B.__init__ - вызваны конструкторы обоих базовых классов
                           # A.__init__ - порядок вызова
                           # C.__init__
```

Как это работает?

Рассмотрим method resolution order, который определен для каждого класса: `print(C.__mro__)` Получим `(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class '__main__.Base'>, <class 'object'>)`

Для реализации наследования питон ищет вызванный атрибут начиная с первого класса до последнего. Этот список создается слиянием (merge sort) списков базовых классов:

- дети проверяются раньше родителей.
- если родителей несколько, то проверяем в том порядке, в котором они перечислены.
- если подходят несколько классов, то выбираем первого родителя.

При вызове super() продолжается поиск, начиная со следующего имени в MRO. Пока каждый переопределенный метод вызывает super() и вызывает его только один раз, будет перебран весь список MRO и каждый метод будет вызван только один раз.

Не забываем вызывать метод суперкласса

А если где-то не вызван метод суперкласса?

```
class Base(object):
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        #super().__init__()     - НЕ вызываем super()
        print('A.__init__')

class B(Base):
    def __init__(self):
        super().__init__()
        print('B.__init__')

class C(A, B):
    def __init__(self):
        super().__init__()
        print('C.__init__')

x = C()                     # A.__init__
                            # C.__init__

print(C.__mro__)
# (<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class '__main__.
Base'>, <class 'object'>)
```

Заметим, что хотя в `B.__init__` есть вызов super(), то до вызова `B.__init__` не доходит.

- Вызываем у объекта класса C метод __init__.
- Ищем его в mro и находим C.__init__. Выполняем его.
- В этом методе вызов super() - ищем метод __init__ далее по списку от найденного.
- Находим A.__init__. Выполняем его. В нем нет никаких super() - дальнейший поиск по mro прекращается.

Нет метода в своем базовом классе, есть у родителя моего сиблинга

Определим класс, который пытается вызвать метод, которого нет в базовом классе:

```
class A(object):
    def spam(self):
        print('A.spam')
        super().spam()

x = A()
x.spam()
```

получим, как и ожидалось:

```
A.spam
Traceback (most recent call last):
  File "examples/oop_super_3.py", line 7, in <module>
    x.spam()
  File "examples/oop_super_3.py", line 4, in spam
    super().spam()
AttributeError: 'super' object has no attribute 'spam'
```

Определим метод spam в классе B. Класс C, наследник A и B, вызывает метод A.spam(), который вызывает B.spam - класс B не связан с классом A.

```
class A(object):
    def spam(self):
        print('A.spam')
        super().spam()

class B(object):
    def spam(self):
        print('B.spam')

class C(A, B):
    pass

y = C()
y.spam()
print(C.__mro__)
```

получим:

```
A.spam
B.spam
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
```

Для объекта класса C вызвали метод spam(). Ищем его в MRO. Находим A.spam() и вызываем. Далее для super() из A.spam() идем дальше от найденного по списку mro и находим B.spam().

Отметим, что при другом порядке описания родителей `class C(B, A)`, вызывается метод B.spam() у которого нет super():

```
B.spam
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

Вызываем метод spam для объекта класса C. В C его нет, ищем дальше в B. Находим. Вызваем. Далее super() нет и дальнейший поиск не производится.

Чтобы не было таких сюрпризов при переопределении методов придерживайтесь правил:

- все методы в иерархии с одинаковым именем имеют одинаковую сигнатуру вызова (количество аргументов и их имена для именованных аргументов).
- реализуйте метод в самом базовом классе, чтобы цепочка вызовов закончилась хоть каким

Обращение к дедушке

Игнорируем родителя

Если у нас есть 3 одинаковых метода foo(self) в наследуемых классах A, B(A), C(B), и нужно из C.foo() вызвать сразу A.foo() минуя B.foo(), то наши классы неправильно сконструированы (почему нужно игнорировать B? может, нужно было наследовать C от A, а не от B?). Нужен рефакторинг.

Но можно всегда вызвать метод по имени класса:

```
class A(object):
    def spam(self):
        print('A.spam')

class B(A):
    def spam(self):
        print('B.spam')

class C(B):
    def spam(self):
        A.spam(self)
        print('C.spam')

y = C()
y.spam()
print(C.__mro__)
```

получим

```
A.spam
C.spam
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

Метод определен только у дедушки

Если в B такого метода нет, и из C.foo() нужно вызвать A.foo() (или в базовом классе выше по иерархии), вызываем super().foo() и больше не думаем, у какого пра-пра-пра-дедушки реализован этот метод.

Просто воспользуйтесь super() для поиска по mro.

```
class A(object):
    def spam(self):
        print('A.spam')

class B(A):
    pass

class C(B):
    def spam(self):
        super().spam()
        print('C.spam')

y = C()
y.spam()
print(C.__mro__)
```

получим:

```
A.spam
C.spam
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

super().super() не работает

Или мы ищем какого-то родителя в mro, или точно указываем из какого класса нужно вызвать метод.

Литература

- Документация по python
- Python's super() considered super!
- Python Cookbook, chapter 8.7 Calling a Method on a Parent Class

Множественное наследование

При создании класса можно указать более одного базового класса.

При поиске атрибутов интерпретатор обходит указанные в заголовке классы слева направо, пока не найдет совпадение.

Так как сами базовые классы могут быть наследниками других классов, получаем дерево (не в терминах графов) наследования.

Разница в поиске атрибутов:

- старые классы - поиск атрибутов сначала продолжается по направлению снизу вверх всеми возможными путями, вплоть до вершины дерева наследования, а затем слева направо.
- новые классы (в Python 3 все классы новые) - поиск в ширину (см. раздел про super() и mro()).

Пример множественного наследования для реализации __repr__

Идея: написать 1 класс, который будет реализовывать удобный __repr__ для произвольного класса и использовать его.

Класс ListInstance в файле lister.py: получим список атрибутов экземпляра.

```
class ListInstance(object):
    def __str__(self):
        return '<Instance of {}, id = {}>:\n{}'.format(
                    self.__class__.__name__,    # имя класса реального объекта
                    id(self),
                    self.__attrnames()          # список пар атрибут - значение
                )

    def __attrnames(self):
        """Список пар атрибут - значение объекта в виде строки с отступом"""
        return '\n'.join(('\tname {} = {}'.format(attr, self.__dict__[attr]) for
                          attr in sorted(self.__dict__)))
```

Использование:

```
class A(object):
    def __init__(self, x):
        self.x = x
    def foo(self):
        pass
class B(A, ListInstance):
    def __init__(self, x, y):
        super().__init__(x)
        self.y = y
    def bzz(self):
        pass
x = ListInstance()
print(x)

b = B(1, 2)
print(b)
```

напечатает:

```
<Instance of ListInstance, id = 4292312080>:

<Instance of B, id = 4292312240>:
        name bzz = <bound method B.bzz of <__main__.B object at 0xffd77e10>>
        name foo = <bound method A.foo of <__main__.B object at 0xffd77e10>>
        name x = 1
        name y = 2
```

Метод __attrnames() сделан с __, чтобы ни в каком классе не был определен метод с таким же методом (ибо он разворачивается в _имякласса__имяметода как и прочие атрибуты, начинающиеся с __ (но не оканчивающиеся на __).

Дополонительные атрибуты тоже будут напечатаны:

```
>>> import lister
>>> class C(lister.ListInstance): pass
...
>>> x = C()
>>> x.a = 1; x.b = 2; x.c = 3
>>> print(x)
<Instance of C, address 40961776>:
    name a=1
    name b=2
    name c=3
```

Преимущества такой реализации __str__:

- реализация вывода сделана в одном месте, ее можно централизованно поменять;

- можно использовать в различных классах.

Изменим класс ListInstance (все атрибуты, вместе с наследуемыми)

Хотим, чтобы print(x) печатало все наследуемые атрибуты (кроме специальных, начинающихся и оканчивающихся __).

Для этого заменим вызов __dir__ на dir() - получим атрибуты всех классов и чтобы находили значение этого атрибута, заменим self.__dir__[attr] на getattr(self, attr), которая будет искать в дереве наследования:

```
class ListInstance(object):
    def __str__(self):
        return '<Instance of {}, id = {}>:\n{}'.format(
                    self.__class__.__name__,    # имя класса реального объекта
                    id(self),
                    self.__attrnames()          # список пар атрибут - значение
                )

    def __attrnames(self):
        """Список пар атрибут - значение объекта в виде строки с отступом"""
        return '\n'.join(('\tname {} = {}'.format(
                            attr,
                            '<>' if attr.startswith('__') and attr.endswith('__') else
 getattr(self, attr)
                            )
                            for attr in sorted(dir(self))
                        ))
```

получим:

```
<Instance of B, id = 4292312336>:
        name _ListInstance__attrnames = <bound method ListInstance.__attrnamesf <__mai
n__.B object at 0xffd77d10>>
        name __class__ = <>
        .... еще много-много методов вида __метод__ .....
        name __str__ = <>
        name bzz = <bound method B.bzz of <__main__.B object at 0xffd77e10>>
        name foo = <bound method A.foo of <__main__.B object at 0xffd77e10>>
        name x = 1
        name y = 2
```

Задача

- Выводить атрибуты, разбив их по классам;

Универсальные фабрики объектов

Иногда до этапа выполнения неизвестно, какие объекты нужно создавать.

Для создания объектов "по требованию", используют шаблон проектирования фабрика.

Сделаем функцию factory, которая создает объект нужного класса с указанными параметрами:

```
def factory(aClass, *args): # Кортеж с переменным числом аргументов
    return aClass(*args)    # Вызов aClass

class Spam:
    def doit(self, message):
        print(message)
class Person:
    def __init__(self, name, job):
        self.name = name
        self.job = job

object1 = factory(Spam)                     # Создать объект Spam
object2 = factory(Person, 'Guido', 'guru')  # Создать объект Person
```

Можно сделать factory гибче:

```
def factory(aClass, *args, **kwargs):       # +kwargs
    return aClass(*args, **kwargs)          # Вызвать aClass
```

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.

Еще раз об __name

mangling - изменение имен, чтобы имена имели принадлежность к *классу*. Это делается не столько для ограничения доступа к именам, а чтобы не было конфликта имен переменных в разных классах.

Имена внутри инструкции class, которые начинаются с __, но не заканчиваются на __ автоматически расширяются именем класса.

Зачем нужно? Делаем класс C3, который наследует классы C1 и C2. В обоих классах есть атрибут x:

```
class C1:
    def meth1(self): self.X = 88    # Предполагается, что X - это мой атрибут
    def meth2(self): print(self.X)
class C2:
    def meth1(self): self.X = 99    # и мой тоже
    def meth2(self): print(self.X)
class C3(C1, C2):
    pass

I = C3() # У меня только один атрибут X!
```

Значение self.X зависит от того, кто последний присвоил значение в этот атрибут, ибо *он один*.

Используем псевдочастотные имена для предотвращения конфликтов:

```
class C1:
    def meth1(self): self.__X = 88      # Теперь X - мой атрибут
    def meth2(self): print(self.__X)    # Превратится в _C1__X
class C2:
    def metha(self): self.__X = 99      # И мой тоже
    def methb(self): print(self.__X)    # Превратится в _C2__X
class C3(C1, C2): pass

I = C3()                                # В I два имени __X (с именами классов)
I.meth1(); I.metha()
print(I.__dict__)
I.meth2(); I.methb()
```

Внимание: псевдочастотные имена мы создаем еще **при разработке классов C1 и C2**. При создании класса C3 мы пользуемся результатом правильного создания. Позволяет избежать *случайных* конфликтов.

@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.

Связанные и несвязанные методы

Несвязанные методы *класса* - это методы без аргумента *self*.

При обращении к функциональному атрибуту класса через имя класса получаем объект несвязанного метода. Для вызова метода нужно передать экземпляр класса в первый аргумент (*self*).

Связанные методы *экземпляра* - пара *self* + функция.

Попытка обращения к функциональному атрибуту класса через имя экземпляра возвращает объект связанного метода. Интерпретатор автоматически упаковывает экземпляр с функцией в объект связанного метода, поэтому вам не требуется передавать экземпляр в вызов такого метода.

И несвязанные методы класса, и связанные методы экземпляра - объекты (так же, как числа и строки) и могут передаваться в виде аргументов (так же как числа или строки). При запуске оба требуют экземпляр в первом аргументе (self).

При вызове связанного метода интерпретатор автоматически подставляет первым аргументом экземпляр, который использовался для создания объекта связанного метода.

```
class Spam:
    def doit(self, message):
        print(message)

object1 = Spam()
object1.doit('hello world')
```

На самом деле создается **объект связанного метода** `object1.doit`, в который упакованы вместе экземпляр `object1` и метод `Spam.doit`.

Можно присвоить этот объект переменной и использовать переменную для вызова, как простую функцию:

```
x = object1.doit    # Объект связанного метода: экземпляр+функция
x('hello world')    # То же, что и object1.doit('...')
```

Вызовем метод через имя класса:

```
object1 = Spam()
t = Spam.doit       # Объект несвязанного метода
t(object1, 'howdy') # Передать экземпляр в первый аргумент
```

self.method - это объект связанного метода экземпляра, так как *self* - объект экземпляра.

```
class Eggs:
    def m1(self, n):
        print(n)
    def m2(self):
        x = self.m1     # Еще один объект связанного метода
        x(42)           # Выглядит как обычная функция

Eggs().m2()             # Выведет 42
```

В Python 3 несвязанные методы - это функции

В Python 3 можно в классе создавать методы без аргумента self и не писать декоратор @staticmethod.

```
class A(object):
    def __init__(self, x):
        self.x = x

    def __str__(self):
        return str(self.x)

    @staticmethod
    def new_A(s):
        t = A(int(s))
        return t

    @staticmethod
    def common_foo(x, k):
        return x * k

    def a_foo(self, k):
        self.x = __class__.common_foo(self.x, k)

    def func_foo(x, k):
        return x * k

    def a_func_foo(self, k):
        self.x = __class__.func_foo(self.x, k)

a1 = A(1)
print('a1 =', a1)

a2 = A.new_A("2")
print('a2 =', a2)

z = A.common_foo(3, 4)
print('z =', z)

a1.a_foo(5)
print('a1 =', a1)

z = A.func_foo(3, 4)
print('z =', z)

a1.a_func_foo(5)
print('a1 =', a1)
```

Использование декоратора @staticmethod повышает читаемость кода.

Связанные методы и другие вызываемые объекты

Связанные методы экземпляра класса - это объекты, которые хранят и экземпляр, и метод. Их можно использовать как обычные функции:

```
>>> class Number:
... def __init__(self, base):
...     self.base = base
... def double(self):
...     return self.base * 2
... def triple(self):
...     return self.base * 3
...
>>> x = Number(2)   # Объекты экземпляров класса
>>> y = Number(3)   # Атрибуты + методы
>>> z = Number(4)
>>> x.double()      # Обычный непосредственный вызов
4
>>> acts = [x.double, y.double, y.triple, z.double] # Список связанных методов
>>> for act in acts:                                # Вызовы откладываются
...     print(act())                                # Вызов как функции
...
4
6
9
8
```

Можно посмотреть на атрибуты, которые дают доступ к объекту экземпляра и к методу:

```
>>> bound = x.double
>>> bound.__self__, bound.__func__
(<__main__.Number object at 0x0278F610>, <function double at 0x027A4ED0>)
>>> bound.__self__.base
2
>>> bound() # Вызовет bound.__func__(bound.__self__, ...)
4
```

Можно обрабатывать одинаково:

- функции, определенные через def или lambda;
- экземпляры, наследующие метод __call__;
- связанные методы экземпляров.

```
>>> def square(arg):
...     return arg ** 2 # Простые функции (def или lambda)
...
>>> class Sum:
...     def __init__(self, val): # Вызываемые экземпляры
...         self.val = val
...     def __call__(self, arg):
...         return self.val + arg
...
>>> class Product:
...     def __init__(self, val): # Связанные методы
...         self.val = val
...     def method(self, arg):
...         return self.val * arg
...
>>> sobject = Sum(2)
>>> pobject = Product(3)
>>> actions = [square, sobject, pobject.method] # Функция, экземпляр, метод
>>> for act in actions:                         # Все 3 вызываются одинаково
...     print(act(5))                           # Вызов любого вызываемого
...                                             # объекта с 1 аргументом
25
7
15
>>> actions[-1](5)                              # Индексы, генераторы, отображения
15
>>> [act(5) for act in actions]
[25, 7, 15]
>>> list(map(lambda act: act(5), actions))
[25, 7, 15]
```

Класс - тоже вызываемый объект. Но он вызывается для создания экземпляра:

```
>>> class Negate:
...     def __init__(self, val):    # Классы - тоже вызываемые объекты
...         self.val = -val         # Но вызываются для создания объектов
...     def __repr__(self):         # Реализует вывод экземпляра
...         return str(self.val)
...
>>> actions = [square, sobject, pobject.method, Negate] # Вызвать класс тоже можно
>>> for act in actions:
...     print(act(5))
...
25
7
15
-5
>>> [act(5) for act in actions]     # Вызовет __repr__, а не __str__!
[25, 7, 15, -5]
```

Посмотрим, какие это объекты:

```
>>> table = {act(5): act for act in actions}    # генератор словарей
>>> for (key, value) in table.items():
...     print('{0:2} => {1}'.format(key, value))
...
-5 => <class '__main__.Negate'>
25 => <function square at 0x025D4978>
15 => <bound method Product.method of <__main__.Product object at 0x025D0F90>>
7 => <__main__.Sum object at 0x025D0F70>
```

Связанные методы и callback

Пример использования связанных методов - GUI на tkinter.

Везде, где можно использовать функцию, можно использовать связанный метод.

Можно написать через функцию (или лямбда-выражение):

```
def handler():
...     сохраняет информацию о состоянии в глобальных переменных...
...
widget = Button(text='spam', command=handler)
```

Можно использовать связанный метод:

```
class MyWidget:
    def handler(self):
...     сохраняет информацию о состоянии в self.attr...
    def makewidgets(self):
        b = Button(text='spam', command=self.handler)
```

self.handler - объект связанного метода. В нем хранятся self и MyWidget.handler. Так как self ссылается на оригинальный экземпляр, то потом, когда метод handler будет вызван для обработки событий, у него будет доступ к экземпляру и его атрибутам (где можно хранить информацию о состоянии объекта между событиями).

Еще один вариант хранения информации между событиями - переопределение метода __call__ (см. перегрузку операторов).

Декораторы

Декораторы - это функции, которые изменяют работу других функций. Это делает код короче.

Рассмотрим где их можно использовать и как написать свой декоратор.

Источники

- Intermediate python
- Python Decorator Library
- Лутц, Изучаем Python. глава 38. Декораторы
- Python3 Patterns, Recipes and Idioms

Пакеты с декорататорами, используемые в заданиях:

- attrs - attrs is the Python package that will bring back the joy of writing classes by relieving you from the drudgery of implementing object protocols
- Contracts - проверка аргументов функций и возвращаемых значений; документация
- numba и User manual - Numba is a compiler for Python array and numerical functions that gives you the power to speed up your applications with high performance functions written directly in Python.

Для чего нужны декораторы

- декораторы функций - управляют не только вызовами функций, но и объектами функций (могут, например, зарегистрировать функции в каком-то прикладном интерфейсе).
- декораторы классов - не только управляет вызовами классов для создания экземпляров класса, но и может изменять объект класса (например, добавить методы), в этом декораторы аналогичны метаклассам.

Плюсы декораторов:

- очевидный синтаксис; они более заметны, чем вызов вспомогательных функций, которые могут быть далеко по тексту кода от функций и классов, к которым они применяются.

- применяются к функции или классу только один раз, когда они определяются, не надо писать дополнительный программный код при каждом их вызове.

Недостатки (любой обертывающей логики):

- изменяют типы декорируемых объектов;
- порождают дополнительные вызовы функций;

Вспоминаем работу с функциями

Чтобы понять как работают декораторы, напишем свой декоратор.

Для этого вспомним некоторые идеи питона.

Функции являются объектами

Функция - это тоже объект. Ее можно присваивать переменной. Вызывать функции можно по имени этой переменной.

```
def hi(name="yasoob"):
    return "Привет " + name

print(hi())
# Вывод: 'Привет yasoob'

# Мы можем присвоить функцию переменной:
greet = hi
# Мы не используем здесь скобки, поскольку наша задача не вызвать функцию,
# а передать её объект переменной. Теперь попробуем запустить

print(greet())
# Вывод: 'Привет yasoob'

# Посмотрим что произойдет, если мы удалим ссылку на оригинальную функцию
del hi
print(hi())
# Вывод: NameError

print(greet())
# Вывод: 'Привет yasoob'
```

Функции внутри функции

В питоне можно определить функцию внутри функции. Ее можно будет вызвать внутри объемлющей функции. Но нельзя напрямую вызвать вне объемлющей функции.

```
def hi(name="yasoob"):
    print("Вы внутри функции hi()")

    def greet():
        return "Вы внутри функции greet()"

    def welcome():
        return "Вы внутри функции welcome()"

    print(greet())
    print(welcome())
    print("Вы внутри функции hi()")

hi()
# Вывод:
# Вы внутри функции hi()
# Вы внутри функции greet()
# Вы внутри функции welcome()
# Вы внутри функции hi()

# Пример демонстрирует, что при вызове hi() вызываются также функции
# greet() и welcome(). Кроме того, две последние функции недоступны
# извне hi():

greet()
# Вывод: NameError: name 'greet' is not defined
```

Видим, что нельзя снаружи объемлющей функции напрямую вызывать вложенную функцию. Но вложенную функцию можно вернуть и потом вызывать из любого места:

Возвращаем функцию из функции

Заметьте, внутри функции hi вложенные функции greet и welcome НЕ вызываются. Они только возвращаются. Вызываются они в других местах (куда функции вернули).

```
def hi(name="yasoob"):
    def greet():
        return "Вы внутри функции greet()"

    def welcome():
        return "Вы внутри функции welcome()"

    if name == "yasoob":
        return greet
    else:
        return welcome

a = hi()
print(a)
# Вывод: <function greet at 0x7f2143c01500>

# Это наглядно демонстрирует, что переменная 'a' теперь указывает на
# функцию greet() в функции hi(). Теперь попробуйте вот это

print(a())
# Вывод: Вы внутри функции greet()
```

Обратите внимание: greet - ссылка на фукнцию, greet() - вызов функции. Скобки имеют значение.

Из функции возвращаются объекты функций greet и welcome.

`hi()` - это вызов функции. Она исполняется и возвращается ссылка на greet.

При вызове `hi(name=ali)` вызовется функции hi и возвратит ссылку на функцию welcome.

Чтобы вызвать ту функцию, которую вернули, можно вызвать `hi()()` - напечатает "Мы внутри greet".

Функцию можно передать аргументом другой функции:

```
def hi():
    return "Привет yasoob!"

def doSomethingBeforeHi(func):
    print("BEFORE hi()")
    print(func())

doSomethingBeforeHi(hi)

# Вывод:
# BEFORE hi()
# Привет yasoob!
```

Декораторы позволяют исполнять код до и после вызова функции

Пишем декоратор

Мы уже по сути написали декоратор. Добавим в него еще действия после вызова функции.

```
def a_new_decorator(a_func):

    def wrapTheFunction():
        print("BEFORE a_func()")

        a_func()

        print("AFTER a_func()")

    return wrapTheFunction

def a_function_requiring_decoration():
    print("Я функция, которая требует декорации")

a_function_requiring_decoration()
# Вывод: "Я функция, которая требует декорации"

a_function_requiring_decoration = a_new_decorator(a_function_requiring_decoration)
# Теперь функция a_function_requiring_decoration обернута в wrapTheFunction()

a_function_requiring_decoration()

# Вывод:
# BEFORE a_func()
# Я функция, которая требует декорации
# AFTER a_func()
```

Мы обернули вызов нашей функции.

По сути, есть только переменные (глобальные и разных сортов локальные) и вызов функций (в т.ч. compile). Есть замыкания (если функция объявляется внутри другой, то из внутренней функции доступны не только свои локальные переменные, но и локальные переменные внешней)

wrapTheFunction - не функция, а локальная переменная для a_new_decorator При каждом вызове a_new_decorator внутри первой строкой вызывается compile, результат которого записывается в локальную переменную wrapTheFunction. Этот compile, среди прочего, запоминает значение a_func. Вот этот compile и возвращается.

Декоратор можно написать через синтаксис с @.

Выражение @a_new_decorator это сокращенная версия следующего кода:

```
a_function_requiring_decoration = a_new_decorator(a_function_requiring_decoration)
```

Пример через @.

```
@a_new_decorator
def a_function_requiring_decoration():
    """Эй ты! Задекорируй меня полностью!"""
    print("Я функция, которая требует декорации")

a_function_requiring_decoration()
# Вывод:
# BEFORE a_func()
# Я функция, которая требует декорации
# AFTER a_func()

# Выражение @a_new_decorator это сокращенная версия следующего кода:
# a_function_requiring_decoration = a_new_decorator(a_function_requiring_decoration)
```

Но у нас перестало работать __name__ так, как мы хотим (печатать имя вот этой самой функции):

```
print(a_function_requiring_decoration.__name__)     # wrapTheFunction, хотим a_functio
n_requiring_decoration
```

Наша функция a_function_requiring_decoration была заменена на wrapTheFunction. Она перезаписала имя и строку документации оригинальной функции. Это можно исправить, используя **functools.wrap**:

```
from functools import wraps

def a_new_decorator(a_func):
    @wraps(a_func)
    def wrapTheFunction():
        print("BEFORE a_func()")
        a_func()
        print("AFTER a_func()")
    return wrapTheFunction

@a_new_decorator
def a_function_requiring_decoration():
    """Эй ты! Задекорируй меня полностью!"""
    print("Я функция, которая требует декорации")

print(a_function_requiring_decoration.__name__)
# Вывод: a_function_requiring_decoration
```

Как сделать декоратор

Примечание: @wraps принимает на вход функцию для декорирования и добавляет функциональность копирования имени, строки документации, списка аргументов и т.д. Это открывает доступ к свойствам декорируемой функции из декоратора.

```
from functools import wraps
def decorator_name(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        if not can_run:
            return "Функция не будет исполнена"
        return f(*args, **kwargs)
    return decorated

@decorator_name
def func():
    return("Функция исполняется")

can_run = True
print(func())
# Вывод: Функция исполняется

can_run = False
print(func())
# Вывод: Функция не будет исполнена
```

Использования декораторов

Авторизация

Декораторы могут использоваться в веб-приложениях для проверки авторизации пользователя, перед тем как открывать ему доступ к функционалу. Они активно используются в веб-фреймворках Flask и Django. Вот пример проверки авторизации на декораторах:

```
from functools import wraps

def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if not auth or not check_auth(auth.username, auth.password):
            authenticate()
        return f(*args, **kwargs)
    return decorated
```

Логирование

```
from functools import wraps

def logit(func):
    @wraps(func)
    def with_logging(*args, **kwargs):
        print(func.__name__ + " была исполнена")
        return func(*args, **kwargs)
    return with_logging

@logit
def addition_func(x):
    """Считаем что-нибудь"""
    return x + x


result = addition_func(4)
# Вывод: addition_func была исполнена
```

Декораторы с аргументами

Является ли @wraps декоратором или она обычная функция, которая может принимать аргументы?

Когда используется синтаксис @my_decorator, мы применяем декорирующую функцию с аргументом в виде декорируемой функции. В Python все является объектом, в том числе и функци. Можно написать функции, возвращающие декорирующие функции.

Вложенные декораторы внутри функций

Добавим к примеру с логером аргумент в виде файла, куда мы будем логировать:

```
from functools import wraps

def logit(logfile='out.log'):
    def logging_decorator(func):
        @wraps(func)
        def wrapped_function(*args, **kwargs):
            log_string = func.__name__ + " была исполнена"
            print(log_string)
            # Открываем логфайл и записваем данные
            with open(logfile, 'a') as opened_file:
                # Мы записываем логи в конкретный файл
                opened_file.write(log_string + '\n')
        return wrapped_function
    return logging_decorator

@logit()
def myfunc1():
    pass

myfunc1()
# Вывод: myfunc1 была исполнена
# Файл out.log создан и содержит строку выше

@logit(logfile='func2.log')
def myfunc2():
    pass

myfunc2()
# Вывод: myfunc2 была исполнена
# Файл func2.log создан и содержит строку выше
```

Декораторы классов

Пусть наш лог-декоратор находится на продакшене и теперь мы хотим, кроме регулярной записи в лог-файл, иметь возможность экстренного уведомления по емейл в случае ошибок.

Звучит как повод написать класс-наследник, но мы до этого работали с декораторами функций. Они связывали имена функций с другим вызываемым объектом на этапе определения фукнции.

Посмортрим на декораторы классов, которые будут связывать имя класса с другим вызываемым объектом на этапе его пределения.

```
class logit(object):
    def __init__(self, logfile='out.log'):
        self.logfile = logfile

    def __call__(self, func):
        log_string = func.__name__ + " была исполнена"
        print(log_string)
        # Открываем логфайл и записваем данные
        with open(self.logfile, 'a') as opened_file:
            # Мы записываем логи в конкретный файл
            opened_file.write(log_string + '\n')
        # Отправляем сообщение
        self.notify()

    def notify(self):
        # Только записываем логи
        pass
```

Такое решение имеет дополнительно преимущество в краткости, в сравнении с вложенными функциями, при этом синтаксис декорирования функции остается прежним:

```
@logit()
def myfunc1():
    pass
```

Расширим класс logit, чтобы notify мог посылать емейл:

```
class email_logit(logit):
    """
    Реализация logit для отправки писем администраторам при вызове
    функции
    """
    def __init__(self, email='admin@myproject.com', *args, **kwargs):
        self.email = email
        super(email_logit, self).__init__(*args, **kwargs)

    def notify(self):
        # Отправляем письмо в self.email
        # Реализация не будет здесь приведена
        pass
```

@email_logit будет работать также как и @logit, при этом отправляя сообщения на почту администратору помимо журналирования.

Еще о декораторах

Управление функцией сразу после ее создания:

```
def decorator(F):
    # Обработка функции F
    return F

@decorator
def func(): ...         # func = decorator(func)
```

Так как функции func присваивается та же оригинальная функция func, то такой декоратор просто что-то делает после определения функций.

Можно использовать для регистрации функции в прикладном интерфейсе, присоединения атрибутов функции и тп.

Мы изучали больше перехват **вызова** функции:

```
def decorator(F):
    # Сохраняет или использует функцию F
    # Возвращает другой вызываемый объект:
    # вложенная инструкция def, class с методом __call__ и так далее.
@decorator
def func(): ...         # func = decorator(func)
```

Декоратор вызывается на этапе декорирования. Возвращаемый объект будет вызываться при вызове функции. Возвращаемый объект может принимать любые аргументы, которые передаются в декорируемую функцию.

Аналогично, при декорировании класса объект экземпляра является всего лишь первым аргументом возвращаемого вызываемого объекта.

Т.е. при декорировании фукнции:

```
def decorator(F):       # На этапе декорирования @
    def wrapper(*args): # Обертывающая функция
        # Использование F и аргументов
        # F(*args) – вызов оригинальной функции
    return wrapper

@decorator              # func = decorator(func)
def func(x, y):         # func передается декоратору в аргументе F
    ...
func(6, 7)              # 6, 7 передаются функции wrapper в виде *args
```

- когда в программе будет вызвана функция func, в действительности будет вызвана функция wrapper, возвращаемая декоратором;
- функция wrapper может вызвать оригинальную функцию func, которая остается доступной ей в области видимости объемлющей функции.

Для каждой декорированной функции создается новая область видимости, в которой сохраняется информация о состоянии.

При декорировании с помощью класса:

```
class decorator:
    def __init__(self, func):   # На этапе декорирования @
        self.func = func
    def __call__(self, *args):  # Обертка вызова функции
        # Использование self.func и аргументов
        # self.func(*args) – вызов оригинальной функции

@decorator
def func(x, y):                 # func = decorator(func)
    ...                         # func будет передана методу __init__

func(6, 7)                      # 6, 7 передаются методу __call__ в виде *args
```

- Когда в программе будет вызывана функция func, в действительности будет вызван метод `__call__` перегрузки операторов экземпляра, созданного декоратором;
- метод `__call__` вызовет оригинальную функцию func, которая доступна ему в

виде атрибута экземпляра.

Для каждой декорированной функции создается новый экземпляр, хранящий информацию о состоянии в своих атрибутах.

В методах класса первым аргуметом идет self, поэтому такой класс-декоратор **не работает для декорирования методов**.

Декорирование методов

```
class decorator:
    def __init__(self, func):   # func – это метод, не связанный
        self.func = func        # с экземпляром класса decorator

    def __call__(self, *args):  # self – это экземпляр декоратора
        # вызов self.func(*args) потерпит неудачу!
        # Экземпляр C отсутствует в args!

class C:
    @decorator
    def method(self, x, y):     # method = decorator(method)
        ...                     # то есть имени method присваивается экземпляр
                                # класса decorator
```

Для **одновременной поддержки** возможности декорирования функций и методов лучше всего применять вложенные функции:

```
def decorator(F):               # F – функция или метод, не связанный с экземпляром
    def wrapper(*args):         # для методов - экземпляр класса в args[0]
        # F(*args) – вызов функции или метода
    return wrapper

@decorator
def func(x, y):                 # func = decorator(func)
    ...
func(6, 7) # В действительности вызовет wrapper(6, 7)

class C:
    @decorator
    def method(self, x, y):     # method = decorator(method)
        ...                     # Присвоит простую функцию

X = C()
X.method(6, 7)                  # В действительности вызовет wrapper(X, 6, 7)
```

Вложенные функции - самый простой способ создания декораторов.

Декораторы классов

Можно декорировать класс. Конструкция

```
@decorator          # Декорирование класса
    class C:
        ...

x = C(99)           # Создает экземпляр
```

эквивалентна конструкции

```
class C:
    ...
C = decorator(C)    # Присваивает имени класса результат,
                    # возвращаемый декоратором
x = C(99)           # Фактически вызовет decorator(C)(99)
```

Результат работы декоратора вызывается, когда позднее в программе нужно создать экземпляр класса. Например, чтобы выполнить некоторые операции сразу после создания класса, нужно вернуть сам оригинальный класс:

```
def decorator(C):
    # Обработать класс C
    return C

@decorator
class C: ...        # C = decorator(C)
```

Если нужно добавить обертывающую логику, которая будет перехватывать создание экземпляра класса C, то нужно возвращать вызываемый объект:

```
def decorator(C):
    # Сохранить или использовать класс C
    # Возвращает другой вызываемый объект:
    # вложенная инструкция def, class с методом __call__ и так далее.

@decorator
class C: ...        # C = decorator(C)
```

Пример такого обертывания класса, когда к классу добавляется интерфейс (тут - обработка обращений к неопределенным атрибутам):

```
def decorator(cls):                  # На этапе декорирования @
    class Wrapper:
        def __init__(self, *args):  # На этапе создании экземпляра
            self.wrapped = cls(*args)
        def __getattr__(self, name): # Вызывается при обращении к атрибуту
            return getattr(self.wrapped, name)
    return Wrapper

@decorator
class C:                             # C = decorator(C)
    def __init__(self, x, y):        # Вызывается методом Wrapper.__init__
        self.attr = 'spam'

x = C(6, 7)                          # В действительности вызовет Wrapper(6, 7)
print(x.attr)                        # Вызовет Wrapper.__getattr__, выведет 'spam'
```

Цитата (Лутц, с 1096) В этом примере декоратор присвоит оригинальному имени класса другой класс, который сохраняет оригинальный класс в области видимости объемлющей функции, создает и встраивает экземпляр оригинального класса при вызове. Когда позднее будет выполнена попытка прочитать значение атрибута экземпляра, она будет перехвачена методом __getattr__ обертки и делегирована встроенному экземпляру оригинального класса. Кроме того, для каждого декорированного класса создается новая область видимости объемлющей функции, в которой сохраняется оригинальный класс.

Поддержка множества экземпляров

Тут будет ОДИН экземпляр класса C

```
class Decorator:
    def __init__(self, C):      # На этапе декорирования @
        self.C = C
    def __call__(self, *args):  # На этапе создания экземпляра
        self.wrapped = self.C(*args)
        return self
    def __getattr__(self, attrname): # Вызывается при обращении к атрибуту
        return getattr(self.wrapped, attrname)

@Decorator
class C: ...                     # C = Decorator(C)

x = C()
y = C()                          # Затрет x!
```

пример подробнее с печатью:

```
class Decorator:
    def __init__(self, C):      # На этапе декорирования @
        print('Decorator.__init__, C=', C)
        self.C = C
    def __call__(self, *args):  # На этапе создания экземпляра
        print('Decorator.__call__, *args', *args, 'self=', id(self))
        self.wrapped = self.C(*args)
        return self
    def __getattr__(self, attrname): # Вызывается при обращении к атрибуту
        return getattr(self.wrapped, attrname)

@Decorator
class C:                        # C = Decorator(C)
    def __init__(self, n):
        print("create C, id=", id(self))
        self.n = n

x = C(1)
y = C(2)                         # Затрет x!

print(id(x))
print(id(y))

print(x.n)
print(y.n)
```

выведет:

```
Decorator.__init__, C= <class '__main__.C'>
Decorator.__call__, *args 1 self= 4293020976
create C, id= 4293020944
Decorator.__call__, *args 2 self= 4293020976
create C, id= 4292307344
4293020976
4293020976
2
2
```

Можно исправить, вернув из `Decorator.__call__` не self, а self.wrapped.

Альтернативный подход будет описан ниже.

Вложение декораторов

Конструкция

```
@A
@B
@C
def f(...):
    ...
```

равноценна следующей:

```
def f(...):
    ...
f = A(B(C(f)))
```

пример:

```
def d1(F): return lambda: 'X' + F()
def d2(F): return lambda: 'Y' + F()
def d3(F): return lambda: 'Z' + F()

@d1
@d2
@d3
def func():         # func = d1(d2(d3(func)))
    return 'spam'

print(func())       # XYZsmap
```

Пример декоратора - измерение производительности

Напишем декоратор, который измеряет сколько времени занимает вызов функции и будет суммировать это время для разных вызовов.

До Python 3.3 для этого можно использовать функцию time.clock(). Начиная с Python 3.3 лучше использовать time.perf_counter() или time.process_time().

Измерение времени генерации списков против времени создания с использованием функции map имеет смысл до Python 2.6, потому что далее map возвращает генератор (т.е. отрабатывает мгновенно) и нужно сравнивать уже итерацию по генератору и по списку.

```
import time

class timer:
    def __init__(self, func):
        self.func = func
        self.alltime = 0

    def __call__(self, *args, **kargs):
        start = time.clock()
        result = self.func(*args, **kargs)
        elapsed = time.clock() - start
        self.alltime += elapsed
        print('%s: %.5f, %.5f' % (self.func.__name__, elapsed, self.alltime))
        return result

@timer
def listcomp(N):
    return [x * 2 for x in range(N)]

@timer
def mapcall(N):
    return map((lambda x: x * 2), range(N))

result = listcomp(5)    # Хронометраж данного вызова, всех вызовов,
listcomp(50000)         # возвращаемое значение
listcomp(500000)
listcomp(1000000)
print(result)
print('allTime = %s' % listcomp.alltime) # Общее время всех вызовов listcomp
print('')

result = mapcall(5)
mapcall(50000)
mapcall(500000)
mapcall(1000000)
print(result)
print('allTime = %s' % mapcall.alltime)         # Общее время всех вызовов mapcall

print('map/comp = %s' % round(mapcall.alltime / listcomp.alltime, 3))
```

получим (Python 2.6):

```
listcomp: 0.00002, 0.00002
listcomp: 0.00910, 0.00912
listcomp: 0.09105, 0.10017
listcomp: 0.17605, 0.27622
[0, 2, 4, 6, 8]
allTime = 0.276223304917

mapcall: 0.00003, 0.00003
mapcall: 0.01363, 0.01366
mapcall: 0.13579, 0.14945
mapcall: 0.27648, 0.42593
[0, 2, 4, 6, 8]
allTime = 0.425933533452
map/comp = 1.542
```

Добавим аргументы в декоратор timer

Хочется, чтобы каждый таймер имел метку для вывода (например, печатать `==>`), а так же отключать и включать сообщения.

Для передачи аргумента в декоратор можно написать функцию-декоратор:

```
def timer(label=''):
    def decorator(func):
        def onCall(*args):      # Аргументы args передаются функции
            ...                 # func сохраняется в объемлющей области
            print(label, ...    # label сохраняется в объемлющей области
        return onCall
    return decorator            # Возвращает фактический декоратор

@timer('==>')                   # То же, что и listcomp = timer('==>')(listcomp)
def listcomp(N): ...            # Имени listcomp присваивается декоратор

listcomp(...)                   # В действительности вызывается функция decorator
```

Задачи

Задача 0. @logit

Дописать в декоратор logit логирование аргументов, с которым была вызвана функция

Задача 1. @timer в виде вложенных функций

Реализовать декоратор @timer через вложенные функции. Должно работать:

1. При каждом вызове декорируемой функции печатать имя функции и сколько она выполнялась.
2. Убедиться, что если функция что-то возвращала, то и после декорирования она возвращает то же значение.
3. Добавить в печать суммарное время всех вызовов этой функции. Убедиться, что для другой функции суммарный счетчик времени - другой (а не один общий на все функции).
4. После декорирования функция должна получить атрибут alltime, который можно вызвать после первого же вызова декорируемой функции. (Например, `print(listcomp.alltime)`
5. Проверить, что декоратор корректно работает для методов классов.

Задача 2. @logit + @timer

Проверить, что оба декоратора могут быть одновременно применены к одной и той же функции.

Задача 3. класс-декоратор

Написать декоратор @timer с использованием класса так, чтобы им можно было декорировать методы класса.

Задача 4. Два класс-декоратора

Реализовать два декоратора через классы (@timer, @logit).

Можно ли ими декорировать одновременно?

Задача 5. Декоратор класса

Написать декоратор класса, который ... TODO.

Использовать декоратор

Напишите пример использования декоратора Easy Dump of Function Arguments

Посмотрите, как он работает для переданных в виде аргументов списков и словарей.

Работа со временем и датами

Для работы со временем и датами нужно импортировать пакет datetime.

- **time** - функции, соответствующие функциям языка C. clock time и the processor run time, перевод времени в строку по формату и разбор строки по формату.
- **datetime** - более высокоуровневый интерфейс работы с датами и временем и их комбинированными значениями (дата + время). Поддерживаются арифметические операции, сравнение и конфигурация таймзоны.
- **calendar** - формативанное представление недель, месяцев и лет. Тут вычисляем день недели по дате и тп.

time

Таблица преобразования типов:

From \ To	timestamp	time tuple	string
timestamp	-	gmtime (UTC) localtime (local time)	-
time tuple	calendar.timegm (UTC) mktime (local time)	-	strftime
string	-	strptime	-

Таблица форматов

Directive	Meaning
%a	Locale’s abbreviated weekday name.
%A	Locale’s full weekday name.
%b	Locale’s abbreviated month name.
%B	Locale’s full month name.
%c	Locale’s appropriate date and time representation.
%d	Day of the month as a decimal number [01,31].
%H	Hour (24-hour clock) as a decimal number [00,23].
%I	Hour (12-hour clock) as a decimal number [01,12].
%j	Day of the year as a decimal number [001,366].
%m	Month as a decimal number [01,12].
%M	Minute as a decimal number [00,59].
%p	Locale’s equivalent of either AM or PM.
%S	Second as a decimal number [00,61].
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
%w	Weekday as a decimal number [0(Sunday),6].
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.
%x	Locale’s appropriate date representation.
%X	Locale’s appropriate time representation.
%y	Year without century as a decimal number [00,99].
%Y	Year with century as a decimal number.
%z	Time zone offset indicating a positive or negative time difference from UTC/GMT of the form +HHMM or -HHMM, where H represents decimal hour digits and M represents decimal minute digits [-23:59, +23:59].
%Z	Time zone name (no characters if no time zone exists).
%%	A literal '%' character.

Литература

- документация по питону

- TutorialsPoint: Python - Date & Time
- Guru99
- Python Cookbook, chapter chapters 3.12-3.16
- PyMOTW

time - clock time

timestamp - количество секунд с 1 января 1970 года (Epoch).

Так как функции этого модуля обращаются к аналогичным функциям языка C, то они платформено-зависимые.

- **time()** - на основе системного вызова time() - "wall clock"
- **monotonic()** - used to measure elapsed time in a long-running process because it is guaranteed never to move backwards, even if the system time is changed.
- **perf_counter()** - самое высокое разрешение для коротких измерений времени (обычно измерение производительности).
- **clock()** - cpu time.
- **process_time()** - returns the combined processor time and system time

Сравним часы

TODO: сделать сравнение в виде таблицы: |process_time | adjustable | implementation | monotonic | resolution | current | |--|--|--|--|--|--| |process_time | adjustable | implementation | monotonic | resolution | current |

```
    clock:
        adjustable    : False
        implementation: clock()
        monotonic     : True
        resolution    : 0.001
        current       : 0.25

    monotonic:
        adjustable    : False
        implementation: clock_gettime(CLOCK_MONOTONIC)
        monotonic     : True
        resolution    : 3.8e-07
        current       : 519313.118129497

    perf_counter:
        adjustable    : False
        implementation: clock_gettime(CLOCK_MONOTONIC)
        monotonic     : True
        resolution    : 3.8e-07
        current       : 519313.118386327

    process_time:
        adjustable    : False
        implementation: clock_gettime(CLOCK_PROCESS_CPUTIME_ID)
        monotonic     : True
        resolution    : 0.015625
        current       : 0.25

    time:
        adjustable    : True
        implementation: clock_gettime(CLOCK_REALTIME)
        monotonic     : False
        resolution    : 0.015625
        current       : 1524093750.628306
```

time.time() - wall clock time

```
import time

print('time()               =', time.time())        # 1524116343.1965854
print('ctime()              =', time.ctime())       # Thu Apr 19 08:39:03 2018
later = time.time() + 15                            # 15 sec later
print('ctime(time()+15 sec) =', time.ctime(later))  # Thu Apr 19 08:39:18 2018
```

Так как *time()* обращается к system clock, и они могут изменяться, когда пользовательские или системные сервисы пытаются синхронизировать время у многих компьютеров, то вызывая последовательно *time()* можно получить времена как по возрастанию, так и по убыванию.

time.monotonic() - время строго по возрастанию

Чтобы гарантированно получить время по возрастанию, используйте **monotonic()**.

Это не время с Эпохи. Это время, чтобы можно было сравнивать.

```
import time

start = time.monotonic()
time.sleep(0.1)
end = time.monotonic()
print('start : {:>9.2f}'.format(start))
print('end   : {:>9.2f}'.format(end))
print('span  : {:>9.2f}'.format(end - start))
```

Получим:

```
start : 543779.74
end   : 543779.84
span  :      0.10
```

Processor Clock Time

time() - возвращает wall clock time, **clock()** - processor clock time - реальное время, которое потратила программа на выполнение.

В примере считается md5 checksum.

```
import hashlib
import time

# Data to use to calculate md5 checksums
data = open(__file__, 'rb').read()

for i in range(5):
    h = hashlib.sha1()
    print(time.ctime(), ': {:0.3f} {:0.3f}'.format(
        time.time(), time.clock()))
    for i in range(300000):
        h.update(data)
    cksum = h.digest()
```

Получим:

```
Thu Apr 19 09:19:08 2018 : 1524118748.980 0.234
Thu Apr 19 09:19:10 2018 : 1524118750.501 1.687
Thu Apr 19 09:19:12 2018 : 1524118752.074 3.155
Thu Apr 19 09:19:13 2018 : 1524118753.588 4.609
Thu Apr 19 09:19:15 2018 : 1524118755.113 6.077
```

time() изменился на 6.132999897, clock() - на 5.843 секунд.

Обычно, если процессор ничего не делает, processor time почти не изменяется. В примере процесс спит. Время wall clock идет, а processor time небольшой.

```
import time

template = '{} - {:0.2f} - {:0.2f}'

print(template.format(
    time.ctime(), time.time(), time.clock())
)

for i in range(3, 0, -1):
    print('Sleeping', i)
    time.sleep(i)
    print(template.format(
        time.ctime(), time.time(), time.clock())
    )
```

получим

```
Thu Apr 19 09:29:30 2018 - 1524119370.68 - 0.19
Sleeping 3
Thu Apr 19 09:29:33 2018 - 1524119373.68 - 0.19
Sleeping 2
Thu Apr 19 09:29:35 2018 - 1524119375.68 - 0.19
Sleeping 1
Thu Apr 19 09:29:36 2018 - 1524119376.69 - 0.19
```

Вызов *sleep()* приостанавливает выполнение текущего thread до тех пор, пока система его не разбудет. Если в программе 1 thread, то приложение ничего не делает пока спит.

Performance Counter

Мерить производительность лучше с помощью **perf_counter()**

Как и в *monotonic()*, это не время с Эпохи, это время чтобы сравнивать.

```
import hashlib
import time

# Data to use to calculate md5 checksums
data = open(__file__, 'rb').read()

loop_start = time.perf_counter()

for i in range(5):
    iter_start = time.perf_counter()
    h = hashlib.sha1()
    for i in range(300000):
        h.update(data)
    cksum = h.digest()
    now = time.perf_counter()
    loop_elapsed = now - loop_start
    iter_elapsed = now - iter_start
    print(time.ctime(), ': {:0.3f} {:0.3f}'.format(
        iter_elapsed, loop_elapsed))
```

получим:

```
Thu Apr 19 09:57:08 2018 : 2.216 2.216
Thu Apr 19 09:57:10 2018 : 2.212 4.428
Thu Apr 19 09:57:13 2018 : 2.192 6.620
Thu Apr 19 09:57:15 2018 : 1.856 8.477
Thu Apr 19 09:57:16 2018 : 1.915 10.392
```

Время как структура struct_time

Когда нужно время, разбитое на составляющие части (год, месяц, день, час, день недели и тп), используйте struct_time

- **gmtime()** - время в UTC
- **localtime()** - время в локальной таймзоне компьютера
- **mktime()** - из времени (структуры) получить timestamp

```
import time

def show_struct(s):
    print('  tm_year :', s.tm_year)
    print('  tm_mon  :', s.tm_mon)
    print('  tm_mday :', s.tm_mday)
    print('  tm_hour :', s.tm_hour)
    print('  tm_min  :', s.tm_min)
    print('  tm_sec  :', s.tm_sec)
    print('  tm_wday :', s.tm_wday)
    print('  tm_yday :', s.tm_yday)
    print('  tm_isdst:', s.tm_isdst)


print('gmtime:')
show_struct(time.gmtime())
print('\nlocaltime:')
show_struct(time.localtime())
print('\nmktime:', time.mktime(time.localtime()))
```

получим:

```
gmtime:
  tm_year : 2018
  tm_mon  : 4
  tm_mday : 19
  tm_hour : 7
  tm_min  : 4
  tm_sec  : 25
  tm_wday : 3
  tm_yday : 109
  tm_isdst: 0

localtime:
  tm_year : 2018
  tm_mon  : 4
  tm_mday : 19
  tm_hour : 10
  tm_min  : 4
  tm_sec  : 25
  tm_wday : 3
  tm_yday : 109
  tm_isdst: 0

mktime: 1524121465.0
```

Time zone

Функции определения времени используют таймзоны заданные в программе или в таймзоне системы по умолчанию. Изменение таймзоны не изменяет реальное время, только его представление.

Для изменения таймзоны установите с помощью **tzset()** переменную окружения TZ. У таймзоны много специфической информации. Обычно используют имя таймзоны, чтобы библиотеки более низкого уровня получили эту информацию.

Изменим несколько раз таймзону и посмотрим на результат работы функций:

```
import time
import os


def show_zone_info():
    print('  TZ    :', os.environ.get('TZ', '(not set)'))
    print('  tzname:', time.tzname)
    print('  Zone  : {} ({})'.format(
        time.timezone, (time.timezone / 3600)))
    print('  DST   :', time.daylight)
    print('  Time  :', time.ctime())
    print()


print('Default :')
show_zone_info()

ZONES = [
    'GMT',
    'Europe/Amsterdam',
]

for zone in ZONES:
    os.environ['TZ'] = zone
    time.tzset()
    print(zone, ':')
    show_zone_info()
```

получим:

```
Default :
  TZ    : Europe/Moscow
  tzname: ('MSK', 'MSD')
  Zone  : -10800 (-3.0)
  DST   : 1
  Time  : Thu Apr 19 10:15:33 2018

GMT :
  TZ    : GMT
  tzname: ('GMT', '   ')
  Zone  : 0 (0.0)
  DST   : 0
  Time  : Thu Apr 19 07:15:33 2018

Europe/Amsterdam :
  TZ    : Europe/Amsterdam
  tzname: ('CET', 'CEST')
  Zone  : -3600 (-1.0)
  DST   : 1
  Time  : Thu Apr 19 09:15:33 2018
```

Преобразование времени в строку и разбор строки

- **strftime(format, *tupletime*)** - из структуры в строку по формату, если *tupletime* не задан, то берется текущее время.
- **strptime(string, *format*)** - из строки в структуру (как в gmtime или localtime)

Таблица преобразования типов: |From \ To | timestamp | time tuple | string | |-|-|-|-| | timestamp |-| gmtime (UTC)
localtime (local time)|-| | time tuple | calendar.timegm (UTC)
mktime (local time)|-| strftime | | string |-| strptime |-|

```
import time


def show_struct(s):
    print('  tm_year :', s.tm_year)
    print('  tm_mon  :', s.tm_mon)
    print('  tm_mday :', s.tm_mday)
    print('  tm_hour :', s.tm_hour)
    print('  tm_min  :', s.tm_min)
    print('  tm_sec  :', s.tm_sec)
    print('  tm_wday :', s.tm_wday)
    print('  tm_yday :', s.tm_yday)
    print('  tm_isdst:', s.tm_isdst)


now = time.ctime(1483391847.433716)
print('Now:', now)

parsed = time.strptime(now)
print('\nParsed:')
show_struct(parsed)

print('\nFormatted:',
      time.strftime("%a %b %d %H:%M:%S %Y", parsed))
```

получим

```
Now: Tue Jan  3 00:17:27 2017

Parsed:
  tm_year : 2017
  tm_mon  : 1
  tm_mday : 3
  tm_hour : 0
  tm_min  : 17
  tm_sec  : 27
  tm_wday : 1
  tm_yday : 3
  tm_isdst: -1

Formatted: Tue Jan 03 00:17:27 2017
```

Таблица форматов

Directive	Meaning
%a	Locale’s abbreviated weekday name.
%A	Locale’s full weekday name.
%b	Locale’s abbreviated month name.
%B	Locale’s full month name.
%c	Locale’s appropriate date and time representation.
%d	Day of the month as a decimal number [01,31].
%H	Hour (24-hour clock) as a decimal number [00,23].
%I	Hour (12-hour clock) as a decimal number [01,12].
%j	Day of the year as a decimal number [001,366].
%m	Month as a decimal number [01,12].
%M	Minute as a decimal number [00,59].
%p	Locale’s equivalent of either AM or PM.
%S	Second as a decimal number [00,61].
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
%w	Weekday as a decimal number [0(Sunday),6].
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.
%x	Locale’s appropriate date representation.
%X	Locale’s appropriate time representation.
%y	Year without century as a decimal number [00,99].
%Y	Year with century as a decimal number.
%z	Time zone offset indicating a positive or negative time difference from UTC/GMT of the form +HHMM or -HHMM, where H represents decimal hour digits and M represents decimal minute digits [-23:59, +23:59].
%Z	Time zone name (no characters if no time zone exists).
%%	A literal '%' character.

datetime - time, date, date+time

Разбор и представление даты и времени, арифметические операции над ними, сравнение.

datetime.time - чисто время (часы, минуты, секунды, милисекунды, таймзона)

Только время без даты.

```
import datetime

t = datetime.time(1, 2, 3)
print(t)
print('hour       :', t.hour)
print('minute     :', t.minute)
print('second     :', t.second)
print('microsecond:', t.microsecond)
print('tzinfo     :', t.tzinfo)
```

получим:

```
01:02:03
hour       : 1
minute     : 2
second     : 3
microsecond: 0
tzinfo     : None
```

Диапазон данных:

```
import datetime

print('Earliest  :', datetime.time.min)
print('Latest    :', datetime.time.max)
print('Resolution:', datetime.time.resolution)
```

получим:

```
Earliest  : 00:00:00
Latest    : 23:59:59.999999
Resolution: 0:00:00.000001
```

Время не может устанавливаться точнее, чем в милисекундах.

```
import datetime

for m in [1, 0, 0.1, 0.6]:
    try:
        print('{:02.1f} :'.format(m),
              datetime.time(0, 0, 0, microsecond=m))
    except TypeError as err:
        print('ERROR:', err)
```

получим:

```
1.0 : 00:00:00.000001
0.0 : 00:00:00
ERROR: integer argument expected, got float
ERROR: integer argument expected, got float
```

datetime.date - дни, месяцы, года

```
import datetime

today = datetime.date.today()
print(today)
print('ctime  :', today.ctime())
tt = today.timetuple()
print('tuple  : tm_year  =', tt.tm_year)
print('         tm_mon   =', tt.tm_mon)
print('         tm_mday  =', tt.tm_mday)
print('         tm_hour  =', tt.tm_hour)
print('         tm_min   =', tt.tm_min)
print('         tm_sec   =', tt.tm_sec)
print('         tm_wday  =', tt.tm_wday)
print('         tm_yday  =', tt.tm_yday)
print('         tm_isdst =', tt.tm_isdst)
print('ordinal:', today.toordinal())
print('Year   :', today.year)
print('Mon    :', today.month)
print('Day    :', today.day)
```

получаем:

```
2018-04-19
ctime  : Thu Apr 19 00:00:00 2018
tuple  : tm_year  = 2018
         tm_mon   = 4
         tm_mday  = 19
         tm_hour  = 0
         tm_min   = 0
         tm_sec   = 0
         tm_wday  = 3
         tm_yday  = 109
         tm_isdst = -1
ordinal: 736803
Year   : 2018
Mon    : 4
Day    : 19
```

ordinal - количество *дней* с 1 января 1 года.

```
import datetime
import time

o = 736803
print('o               :', o)
print('fromordinal(o)  :', datetime.date.fromordinal(o))

t = time.time()
print('t               :', t)
print('fromtimestamp(t):', datetime.date.fromtimestamp(t))
```

получаем:

```
o               : 736803
fromordinal(o)  : 2018-04-19
t               : 1524125286.857128
fromtimestamp(t): 2018-04-19
```

Диапазон date

```
import datetime

print('Earliest  :', datetime.date.min)
print('Latest    :', datetime.date.max)
print('Resolution:', datetime.date.resolution)
```

получаем:

```
Earliest  : 0001-01-01
Latest    : 9999-12-31
Resolution: 1 day, 0:00:00
```

Создадим дату в виде строки и заменим в строке год

```
import datetime

d1 = datetime.date(2008, 3, 29)
print('d1:', d1.ctime())

d2 = d1.replace(year=2009)
print('d2:', d2.ctime())
```

получим

```
d1: Sat Mar 29 00:00:00 2008
d2: Sun Mar 29 00:00:00 2009
```

Острожнее с таким заменами. Вы можете в дате 29 февраля високосного года заменить год на невисокосный. Что делать? Изменять год через timedelta.

datetime.timedelta - разница времен

Если объект datetime - время отправления и прибытия поезда (с датой!), то timedelta - время в пути.

```
import datetime

print('microseconds:', datetime.timedelta(microseconds=1))
print('milliseconds:', datetime.timedelta(milliseconds=1))
print('seconds     :', datetime.timedelta(seconds=1))
print('minutes     :', datetime.timedelta(minutes=1))
print('hours       :', datetime.timedelta(hours=1))
print('days        :', datetime.timedelta(days=1))
print('weeks       :', datetime.timedelta(weeks=1))
```

получим:

```
microseconds: 0:00:00.000001
milliseconds: 0:00:00.001000
seconds     : 0:00:01
minutes     : 0:01:00
hours       : 1:00:00
days        : 1 day, 0:00:00
weeks       : 7 days, 0:00:00
```

total_seconds() - длительность в секундах (большое число)

```
import datetime

for delta in [datetime.timedelta(microseconds=1),
              datetime.timedelta(milliseconds=1),
              datetime.timedelta(seconds=1),
              datetime.timedelta(minutes=1),
              datetime.timedelta(hours=1),
              datetime.timedelta(days=1),
              datetime.timedelta(weeks=1),
              ]:
    print('{:15} = {:8} seconds'.format(
        str(delta), delta.total_seconds())
    )
```

получим

```
0:00:00.000001  =    1e-06 seconds
0:00:00.001000  =    0.001 seconds
0:00:01         =      1.0 seconds
0:01:00         =     60.0 seconds
1:00:00         =   3600.0 seconds
1 day, 0:00:00  =  86400.0 seconds
7 days, 0:00:00 = 604800.0 seconds
```

Арифметические операции над временем

```
import datetime

today = datetime.date.today()
print('Today    :', today)

one_day = datetime.timedelta(days=1)
print('One day  :', one_day)

yesterday = today - one_day
print('Yesterday:', yesterday)

tomorrow = today + one_day
print('Tomorrow :', tomorrow)

print()
print('tomorrow - yesterday:', tomorrow - yesterday)
print('yesterday - tomorrow:', yesterday - tomorrow)
```

получим:

```
Today    : 2018-04-19
One day  : 1 day, 0:00:00
Yesterday: 2018-04-18
Tomorrow : 2018-04-20

tomorrow - yesterday: 2 days, 0:00:00
yesterday - tomorrow: -2 days, 0:00:00
```

Можно увеличить timedelta с помощью арифметических операций:

```
import datetime

one_day = datetime.timedelta(days=1)
print('1 day    :', one_day)
print('5 days   :', one_day * 5)
print('1.5 days :', one_day * 1.5)
print('1/4 day  :', one_day / 4)

# assume an hour for lunch
work_day = datetime.timedelta(hours=7)
meeting_length = datetime.timedelta(hours=1)
print('meetings per day :', work_day / meeting_length)
```

получим

```
1 day    : 1 day, 0:00:00
5 days   : 5 days, 0:00:00
1.5 days : 1 day, 12:00:00
1/4 day  : 6:00:00
meetings per day : 7.0
```

Сравнение времен

```
import datetime
import time

print('Times:')
t1 = datetime.time(12, 55, 0)
print('  t1:', t1)
t2 = datetime.time(13, 5, 0)
print('  t2:', t2)
print('  t1 < t2:', t1 < t2)

print()
print('Dates:')
d1 = datetime.date.today()
print('  d1:', d1)
d2 = datetime.date.today() + datetime.timedelta(days=1)
print('  d2:', d2)
print('  d1 > d2:', d1 > d2)
```

получаем

```
Times:
  t1: 12:55:00
  t2: 13:05:00
  t1 < t2: True

Dates:
  d1: 2018-04-19
  d2: 2018-04-20
  d1 > d2: False
```

Комбинируем дату и время

Класс **datetime** - и дата, и время вместе.

```
import datetime

print('Now    :', datetime.datetime.now())
print('Today  :', datetime.datetime.today())
print('UTC Now:', datetime.datetime.utcnow())
print()

FIELDS = [
    'year', 'month', 'day',
    'hour', 'minute', 'second',
    'microsecond',
]

d = datetime.datetime.now()
for attr in FIELDS:
    print('{:15}: {}'.format(attr, getattr(d, attr)))
```

получим

```
Now    : 2018-04-19 11:26:41.266448
Today  : 2018-04-19 11:26:41.266670
UTC Now: 2018-04-19 08:26:41.266753

year           : 2018
month          : 4
day            : 19
hour           : 11
minute         : 26
second         : 41
microsecond    : 267463
```

Есть так же **fromordinal()** и **fromtimestamp()**

```
import datetime

t = datetime.time(1, 2, 3)
print('t :', t)

d = datetime.date.today()
print('d :', d)

dt = datetime.datetime.combine(d, t)
print('dt:', dt)
```

получим:

```
t : 01:02:03
d : 2018-04-19
dt: 2018-04-19 01:02:03
```

Разбор строки и преобразование в строку

По умолчанию объект datetime представлен в ISO-8601 формате (YYYY-MM-DDTHH:MM:SS.mmmmmm).

strftime() - представим в виде строки, используя другие форматы. **strptime()** - из строки (parse) в объект datetime

```
import datetime

format = "%a %b %d %H:%M:%S %Y"

today = datetime.datetime.today()
print('ISO     :', today)

s = today.strftime(format)
print('strftime:', s)

d = datetime.datetime.strptime(s, format)
print('strptime:', d.strftime(format))
```

получим

```
ISO     : 2018-04-19 11:31:58.397024
strftime: Thu Apr 19 11:31:58 2018
strptime: Thu Apr 19 11:31:58 2018
```

format поддерживает форматирование datetime

```
import datetime

today = datetime.datetime.today()
print('ISO     :', today)
print('format(): {:%a %b %d %H:%M:%S %Y}'.format(today))
```

получим

```
ISO     : 2018-03-18 16:20:35.006116
format(): Sun Mar 18 16:20:35 2018
```

Что получим по каким форматам

если возьмем 5:00 PM January 13, 2016 in the US/Eastern time zone:

Symbol	Meaning	Example
%a	Abbreviated weekday name	'Wed'
%A	Full weekday name	'Wednesday'
%w	Weekday number – 0 (Sunday) through 6 (Saturday)	'3'
%d	Day of the month (zero padded)	'13'
%b	Abbreviated month name	'Jan'
%B	Full month name	'January'
%m	Month of the year	'01'
%y	Year without century	'16'
%Y	Year with century	'2016'
%H	Hour from 24-hour clock	'17'
%I	Hour from 12-hour clock	'05'
%p	AM/PM	'PM'
%M	Minutes	'00'
%S	Seconds	'00'
%f	Microseconds	'000000'
%z	UTC offset for time zone-aware objects	'-0500'
%Z	Time Zone name	'EST'
%j	Day of the year	'013'
%W	Week of the year	'02'
%c	Date and time representation for the current locale	'Wed Jan 13 17:00:00 2016'
%x	Date representation for the current locale	'01/13/16'
%X	Time representation for the current locale	'17:00:00'
%%	A literal % character	'%'

Таймзона

Within datetime, time zones are represented by subclasses of tzinfo. Since tzinfo is an abstract base class, applications need to define a subclass and provide appropriate implementations for a few methods to make it useful.

datetime does include a somewhat naive implementation in the class timezone that uses a fixed offset from UTC, and does not support different offset values on different days of the year, such as where daylight savings time applies, or where the offset from UTC has changed over time.

```
import datetime

min6 = datetime.timezone(datetime.timedelta(hours=-6))
plus6 = datetime.timezone(datetime.timedelta(hours=6))
d = datetime.datetime.now(min6)

print(min6, ':', d)
print(datetime.timezone.utc, ':',
      d.astimezone(datetime.timezone.utc))
print(plus6, ':', d.astimezone(plus6))

# convert to the current system timezone
d_system = d.astimezone()
print(d_system.tzinfo, '      :', d_system)
```

получим

```
UTC-06:00 : 2018-04-19 02:40:28.133666-06:00
UTC : 2018-04-19 08:40:28.133666+00:00
UTC+06:00 : 2018-04-19 14:40:28.133666+06:00
MSK       : 2018-04-19 11:40:28.133666+03:00
```

The third party module pytz is a better implementation for time zones. It supports named time zones, and the offset database is kept up to date as changes are made by political bodies around the world.

Задачи

0. Создать файл (директорию) со временем запуска теста

Запускается программа. Для хранения данных конкретного запуска создайте файл с датой и временем этого запуска в формате yyyymmdd_hhmm.txt

1. Сколько времени длился тест

Дан файл логов теста (bet.log). Первая колонка - timestamp. Напечатать сколько времени длился тест в днях (если длился более суток), часах, минутах и секундах.

1.1 Длительность времени теста записать в часах в дробном формате

Например, если тест длился 1 день 3 часа 15 минут, вывести `27.25 h`

Прочие задачи - разобрать и записать формально.

1. Написать функции преобразования из строки (заданного формата + информация о том что строка в UTC, в локальном или в заданном поясе, это могут быть три отдельные задачи) в unix timestamp.
2. Потом из UNIX timestamp в строку заданного формата (и в заданном часовом поясе).
3. Округлить до ближайшей 10-минутки (15-минутки, часа или ещё чего), обычно делаю через преобразование к timestamp, округление и обратное преобразование, может придумаешь что-то лучше.
4. Вывод дат когда будут следующие семинары (взять текущую дату и в цикле прибавлять дельту в 7 дней, опционально пропускать праздники)
5. Что-то про вывод дней недели.
6. Вывод дат для годичного отчёта с детализацией по месяцам: взять текущую дату (она может передаваться как параметр), truncate до начала года (т.е. до первого января текущего года, через replace или ещё как), потом в цикле выводить записи

вида "первый день - последний день" до текущего дня (не из параметра, а именно текущего, т.е. идём до конца года, не залезая в будущее) или до 31го декабря (что раньше), примеры: а) 2018-01-01 - 2018-01-31 2018-02-01 - 2018-02-28 2018-03-01 - 2018-03-23 б) 2016-01-01 - 2016-01-31 2016-02-01 - 2016-02-29 ... 2016-12-01 - 2016-12-31 Подсказка: в цикле прибавлять дельту в одни сутки, переходить на новую строку когда месяц текущих просматриваемых суток не равен месяцу предыдущих. Есть другой вариант: идти по первым числам месяцев и вычитать один день.

7. Вывод дат с детализацией по дням, аналогично предыдущей задаче (многоточия потому что мне влом копировать, в выводе их не должно быть): 2018-01: 1 2 3 ... 31 2018-02: 1 2 ... 28 2018-03: 1 2 ... 23

Разбор аргументов командной строки

- getopt - как getopt в C.
- optparse - deprecated
- argparse - будем изучать

Есть еще другие модули.

Источники

- argparse tutorial
- argparse документация

Какие бывают аргументы командной строки

Запустим команду **ls** и посмотрим, какие у нее есть аргументы:

```
$ ls
cpython  devguide  prog.py  pypy  rm-unused-function.patch
$ ls pypy
ctypes_configure  demo  dotviewer  include  lib_pypy  lib-python ...
$ ls -l
total 20
drwxr-xr-x 19 wena wena 4096 Feb 18 18:51 cpython
drwxr-xr-x  4 wena wena 4096 Feb  8 12:04 devguide
-rwxr-xr-x  1 wena wena  535 Feb 19 00:05 prog.py
drwxr-xr-x 14 wena wena 4096 Feb  7 00:59 pypy
-rw-r--r--  1 wena wena  741 Feb 18 01:01 rm-unused-function.patch
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
...
```

- Можно запускать команду без параметров, она покажет содержимое текущей директории.
- Позиционный аргумент - имя директории, которую мы просматриваем. Программа решает что делать с аргументом на основе того, *где* он появился в командной строке. Так в команде `cp src dst` первый аргумент - что копируем, второй аргумент - куда копируем.
- `-l` может появиться, а может и нет. Опциональный аргумент.

    - Возможна склейка аргументов, т.е. либо вызов `ls -l -a`, либо `ls -la`.
    - опциональные аргументы возможно записать в короткой `-s` или полной `--size` форме.
- есть хелп

Простой разбор аргументов

Почти ничего не сделали.

```
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

что получили?

```
$ python3 prog.py
$ python3 prog.py --help
usage: prog.py [-h]

optional arguments:
  -h, --help  show this help message and exit
$ python3 prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python3 prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

- Без аргументов ничего не делает.
- Есть хелп, который запускается по ключам -h или --help
- Если задаем аргументы, которых нет, получаем сообщение об ошибке + usage.

Позиционные аргументы

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
args = parser.parse_args()
print(args.echo)
```

запускаем:

```
$ python3 prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python3 prog.py --help
usage: prog.py [-h] echo

positional arguments:
  echo

optional arguments:
  -h, --help  show this help message and exit
$ python3 prog.py foo
foo
```

- функция `add_argument("echo")` добавила позиционный аргумент в список допустимых аргументов.
- `parse_args()` возвращает данные, в атрибуте *echo* этих данных находится позиционный аргумент.
- В хелп попадет без описания
- если он не задан, получаем сообщение об ошибке, какого именно аргумента нет.

Добавим хелп:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use here")
args = parser.parse_args()
print(args.echo)
```

получим:

```
$ python3 prog.py -h
usage: prog.py [-h] echo

positional arguments:
  echo        echo the string you use here

optional arguments:
  -h, --help  show this help message and exit
```

Пусть программа считает квадрат аргумента:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number")
args = parser.parse_args()
print(args.square**2)
```

получаем:

```
$ python3 prog.py 4
Traceback (most recent call last):
  File "prog.py", line 5, in <module>
    print(args.square**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Очевидно, что аргументы разбираются как строки. А нужно число.

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number",
                    type=int)
args = parser.parse_args()
print(args.square**2)
```

получим:

```
$ python3 prog.py 4
16
$ python3 prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four
```

Заметьте, что диагностика изменилась. Аргумент есть, но не того типа - тоже проведет к сообщению об ошибке.

TODO: описать все возможные типы из документации.

Опциональные аргументы

Добавляются той же функцией **add_argument**, начинаются с --.

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity")
args = parser.parse_args()
if args.verbosity:
    print("verbosity turned on")
```

получим:

```
$ python3 prog.py --verbosity 1
verbosity turned on
$ python3 prog.py
$ python3 prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]

optional arguments:
  -h, --help            show this help message and exit
  --verbosity VERBOSITY
                        increase output verbosity
$ python3 prog.py --verbosity
usage: prog.py [-h] [--verbosity VERBOSITY]
prog.py: error: argument --verbosity: expected one argument
```

- Программа написана так, чтобы что-то печатать, когда задано --verbosity и ничего не печатать, когда не задано.
- аргумент не обязательный, можно запускать без него.
- если запустили без него, то значение переменной args.verbosity выставляется в None и проверка if args.verbosity дает False.
- добавлено его описание в секцию optional arguments
- требует параметра после --verbosity (любого типа), иначе не работает.

Но такое поведение удобно, например, для опционального задания конфиг-файла, `-c myconfig.ini`, но не для переменной, которая либо включена (повышенный уровень логирования), либо выключена.

Сделаем ее булевской переменной:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

получим:

```
$ python3 prog.py --verbose
verbosity turned on
$ python3 prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python3 prog.py --help
usage: prog.py [-h] [--verbose]

optional arguments:
  -h, --help  show this help message and exit
  --verbose   increase output verbosity
```

Теперь опциональный аргумент должен запускаться без параметров и его значение либо True (если указан), либо False (если не указан).

Хелп изменился.

Короткие имена -v и --verbosity

Просто перечислите варианты задания аргумента в add_argument:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-v", "--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

получим:

```
$ python3 prog.py -v
verbosity turned on
$ python3 prog.py --help
usage: prog.py [-h] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  increase output verbosity
```

Позиционные и опциональные аргументы вместе

В программу вычисления квадрата числа добавим опцию --verbosity.

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbose", action="store_true",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose:
    print("the square of {} equals {}".format(args.square, answer))
else:
    print(answer)
```

Запустим программу:

```
$ python3 prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required: square
$ python3 prog.py 4
16
$ python3 prog.py 4 --verbose
the square of 4 equals 16
$ python3 prog.py --verbose 4
the square of 4 equals 16
```

- Вернули позиционный аргумент, значит получили ошибку, что его нет.
- порядок позиционного и опционального аргумента не имеет значения.

Опциональный аргумент с параметром + позиционный аргумент

Сделаем шаг назад. Пусть наш опциональный аргумент имеет параметр - какой уровень логирования будет использован (целое число, как и позиционный аргумент).

Как в этом случае задать и разобрать аргументы?

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity == 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

запускаем:

```
$ python3 prog.py 4
16
$ python3 prog.py 4 -v
usage: prog.py [-h] [-v VERBOSITY] square
prog.py: error: argument -v/--verbosity: expected one argument
$ python3 prog.py 4 -v 1
4^2 == 16
$ python3 prog.py 4 -v 2
the square of 4 equals 16
$ python3 prog.py 4 -v 3
16
```

выбор значения из заранее определенного списка

Не нравится, что при задании `--verbosity 3` мы получили 0 уровень. И никто не скажет, какие уровни доступны. Зададим допустимый набор аргументов:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int, choices=[0, 1, 2],
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity == 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

запустим:

```
$ python3 prog.py 4 -v 3
usage: prog.py [-h] [-v {0,1,2}] square
prog.py: error: argument -v/--verbosity: invalid choice: 3 (choose from 0, 1, 2)
$ python3 prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square

positional arguments:
  square                display a square of a given number

optional arguments:
  -h, --help            show this help message and exit
  -v {0,1,2}, --verbosity {0,1,2}
                        increase output verbosity
```

Заметьте, изменились и хелп, и сообщение ошибке.

Подсчет количества заданных аргументов -v action='count'

Хотим определять уровень логирования по тому, сколько раз задали опцию -v в командной строке: ничего, -v, -vv.

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display the square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity == 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

запускаем:

```
$ python3 prog.py 4
16
$ python3 prog.py 4 -v
4^2 == 16
$ python3 prog.py 4 -vv
the square of 4 equals 16
$ python3 prog.py 4 --verbosity --verbosity
the square of 4 equals 16
$ python3 prog.py 4 -v 1
usage: prog.py [-h] [-v] square
prog.py: error: unrecognized arguments: 1
$ python3 prog.py 4 -h
usage: prog.py [-h] [-v] square

positional arguments:
  square           display a square of a given number

optional arguments:
  -h, --help       show this help message and exit
  -v, --verbosity  increase output verbosity
$ python3 prog.py 4 -vvv
16
```

- Если не определен ни один флаг -v, то args.verbosity = None
- Без разницы - указыват короткую или полную форму флага -v или --verbosity.
- Слишком большое количество аргументов -vvv дает опять лаконичную форму логирования.

Поправим это, заменив `if args.verbosity == 2` на `if args.verbosity >= 2`.

Увы, если не задано -v, то переменная None и для нее не определено >=.

Добавим опцию **default**=0, чтобы когда аргумент не задан, значение переменной было не None, а 0.

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count", default=0,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity >= 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity >= 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

запустим:

```
$ python3 prog.py 4 -vvv
the square of 4 equals 16
$ python3 prog.py 4
16
```

Опциональный аргумент МОЖЕТ содержать параметр

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

Сложный пример

Напишем программу, которая возводит не в квадрат, а в указанную степень:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print("{} to the power {} equals {}".format(args.x, args.y, answer))
elif args.verbosity >= 1:
    print("{}^{} == {}".format(args.x, args.y, answer))
else:
    print(answer)
```

запустим:

```
$ python3 prog.py
usage: prog.py [-h] [-v] x y
prog.py: error: the following arguments are required: x, y
$ python3 prog.py -h
usage: prog.py [-h] [-v] x y

positional arguments:
  x                the base
  y                the exponent

optional arguments:
  -h, --help       show this help message and exit
  -v, --verbosity
$ python3 prog.py 4 2 -v
4^2 == 16
```

Взаимоисключающие аргументы

Зададим два аргумента: -v - повысить уровень логирования, -q - отключить логирование.

Добавим их в **add_mutually_exclusive_group()**

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print("{} to the power {} equals {}".format(args.x, args.y, answer))
else:
    print("{}^{} == {}".format(args.x, args.y, answer))
```

запустим:

```
$ python3 prog.py 4 2
4^2 == 16
$ python3 prog.py 4 2 -q
16
$ python3 prog.py 4 2 -v
4 to the power 2 equals 16
$ python3 prog.py 4 2 -vq
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
$ python3 prog.py 4 2 -v --quiet
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
```

Добавим описание, для чего нужна программа

argparse.ArgumentParser(description="calculate X to the power of Y**")**

```
import argparse

parser = argparse.ArgumentParser(description="calculate X to the power of Y")
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print("{} to the power {} equals {}".format(args.x, args.y, answer))
else:
    print("{}^{} == {}".format(args.x, args.y, answer))
```

запустим:

```
$ python3 prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose
  -q, --quiet
```

Переменное число позиционных аргументов

Напишем программу, которая складывает числа, которые заданы в виде аргументов.

TODO

TODO: документация по add_argument (action)

Файл конфигурации

Задать конфиграционный файл можно разными способами:

- написать python код и выполнить его;
- сохранить данные в json формате и читать их;
- написать .ini файл.

Для разбора *.ini* файла воспользуйтесь модулем configparser

Файл в human readable формате, данные разделены на секции. В разных секциях могут быть переменные с одинаковыми именами. Значения переменных разных типов. Есть комментарии.

Пример файла, который надо разобрать:

```
; config.ini
; Sample configuration file

[installation]
library=%(prefix)s/lib
include=%(prefix)s/include
bin=%(prefix)s/bin
prefix=/usr/local

# Setting related to debug configuration
[debug]
log_errors=true
show_warnings=False

[server]
port: 8080
nworkers: 32
pid-file=/tmp/spam.pid
root=/www/root
signature:
    =================================
    Brought to you by the Python Cookbook
    =================================
```

Код для разбора файла:

```
>>> from configparser import ConfigParser
>>> cfg = ConfigParser()
>>> cfg.read('config.ini')
['config.ini']
>>> cfg.sections()
['installation', 'debug', 'server']
>>> cfg.get('installation','library')
'/usr/local/lib'
>>> cfg.getboolean('debug','log_errors')
True
>>> cfg.getint('server','port')
8080
>>> cfg.getint('server','nworkers')
32
>>> print(cfg.get('server','signature'))
==================================
Brought to you by the Python Cookbook
==================================
>>>
```

Модификация конфиг-файла с помощью **cfg.write()**:

```
>>> cfg.set('server','port','9000')
>>> cfg.set('debug','log_errors','False')
>>> import sys
>>> cfg.write(sys.stdout)
[installation]
library = %(prefix)s/lib
include = %(prefix)s/include
bin = %(prefix)s/bin
prefix = /usr/local
[debug]
log_errors = False
show_warnings = False
[server]
port = 9000
nworkers = 32
pid-file = /tmp/spam.pid
root = /www/root
signature =
==================================
Brought to you by the Python Cookbook
==================================
>>>
```

Разница между *.ini* файлом и куском кода на питоне

Меньше ограничений на формат записи

Эти записи эквивалентны:

```
prefix=/usr/local
prefix: /usr/local
```

Имена переменных не зависят от регистра:

```
>>> cfg.get('installation','PREFIX')
'/usr/local'
>>> cfg.get('installation','prefix')
'/usr/local'
```

В булевские переменные можно писать разумные значения: (эти записи эквивалентны)

```
log_errors = true
log_errors = TRUE
log_errors = Yes
log_errors = 1
```

Конфиг-файл не обязан исполняться сверху вниз

Возможны подстановки переменных (переменная определена после использования):

```
[installation]
library=%(prefix)s/lib
include=%(prefix)s/include
bin=%(prefix)s/bin
prefix=/usr/local
```

Несколько конфигурационных файлов

Можно сливать прочитанные конфигурации в общую конфигурацию.

Пусть пользователь имеет свой конфиг-файл:

```
; ~/.config.ini
[installation]
prefix=/Users/beazley/test
[debug]
log_errors=False
```

Можно слить его с ранее прочитанной конфигурацией:

```
>>> cfg.get('installation', 'prefix')
'/usr/local'
>>> # Merge in user-specific configuration
>>> import os
>>> cfg.read(os.path.expanduser('~/.config.ini'))
['/Users/beazley/.config.ini']
>>> cfg.get('installation', 'prefix')
'/Users/beazley/test'
>>> cfg.get('installation', 'library')
'/Users/beazley/test/lib'
>>> cfg.getboolean('debug', 'log_errors')
False
```

Observe how the override of the prefix variable affects other related variables, such as the setting of library. This works because variable interpolation is performed as late as possible. You can see this by trying the following experiment:

```
>>> cfg.get('installation','library')
'/Users/beazley/test/lib'
>>> cfg.set('installation','prefix','/tmp/dir')
>>> cfg.get('installation','library')
'/tmp/dir/lib'
>>>
```

Finally, it’s important to note that Python does not support the full range of features you might find in an .ini file used by other programs (e.g., applications on Windows). Make sure you consult the configparser documentation for the finer details of the syntax and supported features.

Задача

Добавить в игру файл конфигурации. Определить в нем ключевые параметры игры: максимальное количество игроков, начальный размер руки, диалект правил и тп.

Логирование

Логирование - означает запись событий, которые случились во время работы программы. Программисты пишут логирующие вызовы, чтобы показать, что случились определенные события. Событие состоит из сообщения-описания и каких-то данных. Событию так же приписывают важность или уровень(severity или level).

Источники

- Документация
- Logging Cookbook
- PEP 282 -- A Logging System
- Logging HOWTO
- Python Cookbook

Логирование на коленке

Логирование - проще не бывает

```
import logging
logging.warning('Watch out!')  # will print a message to the console
logging.info('I told you so')  # will not print anything
```

На консоль будет напечатано `WARNING:root:Watch out!`. Второе сообщение не будет напечатано, потому что по умолчанию выставлен уровень WARNING.

Логируем в файл

Для логирования в файл определим имя этого файла. Например, `example.log`. Заодно установили уровень логирования.

```
import logging
logging.basicConfig(filename='example.log', level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
```

Логирование в простом скрипте

Нужно добавить простое логирование в скрипт. Самый простой вариант.

Используем модуль **logging**.

```
import logging
def main():
    # Configure the logging system
    logging.basicConfig(
        filename='app.log',
        level=logging.ERROR
    )

    # Variables (to make the calls that follow work)
    hostname = 'www.python.org'
    item = 'spam'
    filename = 'data.csv'
    mode = 'r'

    # Example logging calls (insert into your program)
    logging.critical('Host %s unknown', hostname)
    logging.error("Couldn't find %r", item)
    logging.warning('Feature is deprecated')
    logging.info('Opening file %r, mode=%r', filename, mode)
    logging.debug('Got here')

if __name__ == '__main__':
    main()
```

Функции **critical(), error(), warning(), info(), debug()** дают возможность логировать сообщения разного уровня важности (по убыванию).

При конфигурации логирования **basicConfig** указывается уровень (аргумент level), по которому делают фильтрацию. Сообщения ниже уровнем игнорируются.

Аргументы в функциях логирования - как в printf языка C. Первый аргумент - форматная строка с форматными символами, следующие аргументы разбираются по указанному формату.

Запустив код выше получим файл *app.log*:

```
CRITICAL:root:Host www.python.org unknown
ERROR:root:Could not find 'spam'
```

Изменим уровень логирования и формат логирующей строки:

```
logging.basicConfig(
    filename='app.log',
    level=logging.WARNING,
    format='%(levelname)s:%(asctime)s:%(message)s'
)
```

Содержимое логов изменится:

```
CRITICAL:2012-11-20 12:27:13,595:Host www.python.org unknown
ERROR:2012-11-20 12:27:13,595:Could not find 'spam'
WARNING:2012-11-20 12:27:13,595:Feature is deprecated
```

Можно не хардкодить конфигурацию логера в программе, а задавать ее в конфигурационном файле:

```
import logging
import logging.config
def main():
    # Configure the logging system
    logging.config.fileConfig('logconfig.ini')
...
```

Содержимое logconfig.ini файла:

```
[loggers]
keys=root

[handlers]
keys=defaultHandler

[formatters]
keys=defaultFormatter

[logger_root]
level=INFO
handlers=defaultHandler
qualname=root

[handler_defaultHandler]
class=FileHandler
formatter=defaultFormatter
args=('app.log', 'a')

[formatter_defaultFormatter]
format=%(levelname)s:%(name)s:%(message)s
```

Логгер должен быть сконфигурирован до записи в него

Для записи в `stderr` **вместо файла**, можно НЕ указывать файл, в который логируем при конфигурации:

```
logging.basicConfig(level=logging.INFO)
```

Для **изменения уровня логирования во время исполнения программы** укажите в конфигурации новый уровень:

```
logging.getLogger().level = logging.DEBUG
```

Если хотим, чтобы логфайл перезаписывался при каждом запуске

filemode - способ открытия логфайла. Откроем его не с 'a' (по умолчанию), а с 'w'.

```
logging.basicConfig(filename='example.log', filemode='w', level=logging.DEBUG)
```

Устанавливаем уровень логирования через командную строку

Пусть наша переменная `loglevel` связана с ключом `--log` в командной строке и может быть установлена, например в `--log=INFO`.

```
# assuming loglevel is bound to the string value obtained from the
# command line argument. Convert to upper case to allow the user to
# specify --log=DEBUG or --log=debug
numeric_level = getattr(logging, loglevel.upper(), None)
if not isinstance(numeric_level, int):
    raise ValueError('Invalid log level: %s' % loglevel)
logging.basicConfig(level=numeric_level, ...)
```

Логирование программы из нескольких файлов

Пусть наша программа написана в нескольких файлах (модулях) и мы хотим писать ее логи из разных файлов в единый лог-файл.

Программа: главный модуль myapp.py и еще один файл с кодом в mylib.py.

Конфигурация логера - до его первого использования

```
# myapp.py
import logging
import mylib

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logging.info('Started')
    mylib.do_something()
    logging.info('Finished')

if __name__ == '__main__':
    main()
```

в другом файле используем уже готовый логер:

```
# mylib.py
import logging

def do_something():
    logging.info('Doing something')
```

Получаем лог:

```
INFO:root:Started
INFO:root:Doing something
INFO:root:Finished
```

Изменение формата логирования

Вместо формата по умолчанию можно настроить свой формат логирования. Укажите параметр **format** в конфигурации.

```
import logging
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)
logging.debug('This message should appear on the console')
logging.info('So should this')
logging.warning('And this, too')
```

получим

```
DEBUG:This message should appear on the console
INFO:So should this
WARNING:And this, too
```

Таблица атрибутов формата

Attribute name	Format	Description
args	You shouldn't need to format this yourself.	The tuple of arguments merged into msg to produce message, or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
asctime	%(asctime)s	Human-readable time when the LogRecord was created. By default this is of the form '2003-07-08 16:49:45,896' (the numbers after the comma are millisecond portion of the time).
created	%(created)f	Time when the LogRecord was created (as returned by time.time()).
exc_info	You shouldn't need to format this yourself.	Exception tuple (a la sys.exc_info) or, if no exception has occurred, None.
filename	%(filename)s	Filename portion of pathname.
funcName	%(funcName)s	Name of function containing the logging call.

levelname	%(levelname)s	Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL').
levelno	%(levelno)s	Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL).
lineno	%(lineno)d	Source line number where the logging call was issued (if available).
message	%(message)s	The logged message, computed as msg % args. This is set when Formatter.format() is invoked.
module	%(module)s	Module (name portion of filename).
msecs	%(msecs)d	Millisecond portion of the time when the LogRecord was created.
msg	You shouldn't need to format this yourself.	The format string passed in the original logging call. Merged with args to produce message, or an arbitrary object (see Using arbitrary objects as messages).
name	%(name)s	Name of the logger used to log the call.
pathname	%(pathname)s	Full pathname of the source file where the logging call was issued (if available).
process	%(process)d	Process ID (if available).
processName	%(processName)s	Process name (if available).
relativeCreated	%(relativeCreated)d	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded.
stack_info	You shouldn't need to format this yourself.	Stack frame information (where available) from the bottom of the stack in the current thread, up to and including the stack frame of the logging call which resulted in the creation of this record.
thread	%(thread)d	Thread ID (if available).
threadName	%(threadName)s	Thread name (if available).

Дата и время в сообщении

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s')
logging.warning('is when this event was logged.')
```

получим

```
2010-12-12 11:41:42,612 is when this event was logged.
```

По умолчанию формат ISO8601, если нужен свой формат, укажите его в параметре **datefmt** при задании конфигурации.

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s', datefmt='%m/%d/%Y %I:%M:%S %p')
logging.warning('is when this event was logged.')
```

получим

```
12/12/2010 11:46:36 AM is when this event was logged.
```

Формат даты и времени такой же, как для функции time.strftime()

Продвинутое логирование

Как это работает?

Объекты:

- **Logger** (логер) - предоставляет интерфейс, которым пользуются в коде.
- **Handler** (обработчик) - посылает запись лога (созданную логером) куда нужно.
- **Filter** (фильтр) - фильтрует записи, которые нужно выводить.
- **Formatter** (форматер) - определяет в каком виде выводим записи.

Информация о событии логирования передается в виде записи (экземпляра LogRecord) между логерами, обработчиками, фильтрами и форматерами.

Для логирования вызваются методы экземпляра класса Logger (далее называем их логерами). Каждый экземпляр имеет имя, и они упорядочены в иерархии пространства имен с использованием точки как разделителя.

Например, логер *scan* - родитель логера *scan.txt*, *scan.html* и *scan.pdf*.

Обычно логер модуля называют по имени модуля:

```
logger = logging.getLogger(__name__)
```

Базовый логер **root**. Он - прародитель всех остальных логеров. Его имя `root`, именно его используют методы **critical(), error(), warning(), info(), debug()**. Функции и методы имеют одинаковые имена.

Можно выводить сообщения различными способами:

- в файл;
- HTTP GET/POST;
- email через SMTP;
- сокеты;
- очереди сообщений;
- ОС-зависимые механизмы: syslog или NT event log (Windows). Куда выводить определяют обработчики (handler). Можно создать свой обработчик, если вам не хватило встроенных обработчиков.

По умолчанию логер никуда не выводит. Чтобы определить куда выводить (на консоль или в файл), используют конфигурацию логера. В примере это был basicConfig(). Когда вы вызывает функции логирования debug(), info(), warning(), error() и critical(), они проверяют, установленно ли куда выводить, и если не установлено, то по умолчанию устанавливают в `sys.stderr` в формате по умолчанию, прежде чем передавать сообщение логеру root.

Формат, устанавливаемый basicConfig по умолчанию это

```
severity:logger name:message
```

его можно изменить, установив свой Formatter object

Logger Flow

()[../assets/logging_flow.png]

Loggers (логеры)

Logger нужен для:

- предоставить методы для прикладного кода для логирования.
- отфильтровать какие сообщения должны появиться согласно важности и фильтрам.
- доставить записи нужным обработчикам.

Основные методы логера - это установка конфигурации и собственно создание записей.

Методы конфигурации логера:

- **Logger.setLevel()** - определяет с какого уровня и выше будут записываться сообщения. Уровни логирования: DEBUG, INFO, WARNING, ERROR, CRITICAL. Если установлен уровень INFO, то сообщения уровня DEBUG выводиться НЕ будут, все остальные - будут.
- **Logger.addHandler()** и **Logger.removeHandler()** - добавляет и удаляет обработчик к этому логеру.
- **Logger.addFilter()** и **Logger.removeFilter()** - добавляет и удаляет фильтр к логеру. Фильтры обычно вам не нужны, используются для сложного логирования.

После конфигурации логера можно пользоваться его методами для создания сообщения.

- **Logger.debug(), Logger.info(), Logger.warning(), Logger.error(), Logger.critical()** создают запись (record), в которой определено сообщение (message) и уровень.
- **Logger.exception()** похож на **Logger.error()**. Разница в том, что Logger.exeption() посылает stacktrace. Используйте этот метод в try .. except блоке.
- **Logger.log()** получает уровень в виде аргумента.
- **getLogger()** возвращает ссылку на объект логера с указанным именем (если указано имя) или root (по умолчанию). Имена объеденены в иерархическую структуру, с разделителем точка. Множественные вызовы этого метода для одного и того же имени возвращают указатель на один и тот же объект.

Логеры foo.bar, foo.bar.baz, foo.bam - наследники логера foo.

У логеров есть эффективный уровень. Если в логере не выставлен явно уровень, то берется уровень родителя и используется как эффективный уровень этого ребенка. Если у родителя нет явно выставленного уровня, то ищется в родителе родителя и так далее вверх по иерархии, пока не найдется явно выставленный уровень. В логере root гарантированно есть явно выставленный уровень. По умолчанию он выставлен в WARNING. Эффективный уровень используют для принятия решения - передавать событие обработчику или нет (уровень события слишком низкий).

Дочерние логеры передают сообщения вверх по цепочке обработчиков, связанных с родительскими логерами. Поэтому можно не определять обработчики для каждого логера, а настроить обработчик на логере и создавать дочерние логеры по мере необходимости. (Можно отключить это распространение сообщений вверх по иерархии выставив атрибут логера propagate=False).

Handlers (обработчики)

Объект Handler отвечает за направление соответствующих сообщений (на основе их важности) туда, куда определено в этом обработчике. У объекта класса Logger может быть добавлено ноль или более обработчиков с помощью **addHandler()** метода. Например, приложение может посылать все сообщения в лог-файл, все error или выше на stdout, все critical - на email. В этом случае нужно 3 обработчика, где каждый обработчик посылает сообщения определенного уровня в определенное место.

В стандартной библиотеке есть набор обработчиков (см. Полезные обработчики); в тьюториалах показываются в основном StreamHandler и FileHandler.

Если вы пользуетесь стандартными обработчиками (а не пишете свой класс обработчика), то вам могут понадобиться методы:

- **setLevel()** - такой же, как в логере. Определяет, начиная с какой важности сообщения будут посылаться туда, куда определяет обработчик. Зачем два одинаковых метода - в логере и в обработчике? В логере определяется, посылается вообще сообщение или нет, а в обработчике - посылается ли оно (если послано) в handler destination.
- **setFormatter()** - определяет Formatter, которым пользуется этот обработчик.
- **addFilter()** и **removeFilter()** - сконфигурировать / деконфигурировать фильтр для обработчика.

Ваше приложение не должно использовать непосредственно объекты класса Handler, а пользоваться его наследниками.

Formatters (форматеры)

Форматеры определяют, какой будет порядок, структура и содержимое сообщения. В отличие от базового класса logging.Handler, ваше приложение может создавать свои (под)классы форматеров, если вам нужно особое поведение. У конструктора три параметра: message format string, a date format string and a style indicator.

```
logging.Formatter.__init__(fmt=None, datefmt=None, style='%')
```

Если нет форматной строки fmt, то используем строку сообщения как есть. Если нет формата даты datefmt, то по умолчанию формат даты `%Y-%m-%d %H:%M:%S` с милисекундами на конце. style (стиль) может быть %, '{' или '$' (по умолчанию %).

Если стиль %, то форматная строка fmt использует стиль `%(<dictionary key>)s` для подстановки строк; возможные ключи описаны в таблице атрибутов.

Если стиль '{', то форматная строка fmt предполагает стиль совместимый с str.format() (с использованием keyword arguments).

При стиле '$' форматная строка должна быть написана в стиле string.Template.substitute()

Этот формат сообщения показывает время в human-readable виде, важность и само сообщение в следующем порядке:

```
'%(asctime)s - %(levelname)s - %(message)s'
```

Из документации по Formatter: Formatters use a user-configurable function to convert the creation time of a record to a tuple. By default, time.localtime() is used; to change this for a particular formatter instance, set the converter attribute of the instance to a function with the same signature as time.localtime() or time.gmtime(). To change it for all formatters, for example if you want all logging times to be shown in GMT, set the `converter` attribute in the Formatter class (to time.gmtime for GMT display).

Конфигурируем логер

3 способа конфигурации логера:

- Создаем логеры, обработчики, форматеры в коде на питоне и вызываем методы конфигурации, которые были описаны выше.
- Создаем файл конфигурации логера и вызываем функцию **fileConfig()**
- Создаем словарь конфигурации логера и передаем его в функцию **dictConfig()**, схема словаря

```
logging.config.fileConfig(fname, defaults=None, disable_existing_loggers=True)
logging.config.dictConfig(config)
```

Приведем примеры всех трех способов.

Создаем логер и связанные с ним объекты в коде

```
import logging

# create logger
logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)

# create console handler and set level to debug
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

# create formatter
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# add formatter to ch
ch.setFormatter(formatter)

# add ch to logger
logger.addHandler(ch)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

получим:

```
$ python simple_logging_module.py
2005-03-19 15:10:26,618 - simple_example - DEBUG - debug message
2005-03-19 15:10:26,620 - simple_example - INFO - info message
2005-03-19 15:10:26,695 - simple_example - WARNING - warn message
2005-03-19 15:10:26,697 - simple_example - ERROR - error message
2005-03-19 15:10:26,773 - simple_example - CRITICAL - critical message
```

Конфиг-файл логера

Конфиг-файл `logging.conf`:

```
[loggers]
keys=root,simpleExample

[handlers]
keys=consoleHandler

[formatters]
keys=simpleFormatter

[logger_root]
level=DEBUG
handlers=consoleHandler

[logger_simpleExample]
level=DEBUG
handlers=consoleHandler
qualname=simpleExample
propagate=0

[handler_consoleHandler]
class=StreamHandler
level=DEBUG
formatter=simpleFormatter
args=(sys.stdout,)

[formatter_simpleFormatter]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
datefmt=
```

читаем его в программе:

```
import logging
import logging.config

logging.config.fileConfig('logging.conf')

# create logger
logger = logging.getLogger('simpleExample')

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

получаем:

```
$ python simple_logging_config.py
2005-03-19 15:38:55,977 - simpleExample - DEBUG - debug message
2005-03-19 15:38:55,979 - simpleExample - INFO - info message
2005-03-19 15:38:56,054 - simpleExample - WARNING - warn message
2005-03-19 15:38:56,055 - simpleExample - ERROR - error message
2005-03-19 15:38:56,130 - simpleExample - CRITICAL - critical message
```

При вызове функции по умолчанию параметр **disable_existing_loggers=True**.

Note that the class names referenced in config files need to be either relative to the logging module, or absolute values which can be resolved using normal import mechanisms. Thus, you could use either WatchedFileHandler (relative to the logging module) or mypackage.mymodule.MyHandler (for a class defined in package mypackage and module mymodule, where mypackage is available on the Python import path).

Словарь конфигурации

Словарь можно записывать в отдельный файл в JSON или YAML формате.

```
version: 1
formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: simple
    stream: ext://sys.stdout
loggers:
  simpleExample:
    level: DEBUG
    handlers: [console]
    propagate: no
root:
  level: DEBUG
  handlers: [console]
```

Что будет, если логер не сконфигурировать?

Если система логирования никак не была сконфигурирована, то возможна ситуация, что событие должно быть залогировано, но у нас нет ни одного обработчика для этого события. В этом случае поведение зависит от версиии Python.

- Python ДО 3.2:
  - If logging.raiseExceptions is False (production mode), the event is silently dropped.

    - If logging.raiseExceptions is True (development mode), a message 'No handlers could be found for logger X.Y.Z' is printed once.
- Python 3.2 или более свежие версии:

The event is output using a 'handler of last resort', stored in logging.lastResort. This internal handler is not associated with any logger, and acts like a StreamHandler which writes the event description message to the current value of sys.stderr (therefore respecting any redirections which may be in effect). No formatting is done on the message - just the bare event description message is printed. The handler's level is set to WARNING, so all events at this and greater severities will be output. To obtain the pre-3.2 behaviour, logging.lastResort can be set to None.

Логирование в библиотеке

Вы пишите модуль, логирование в котором хочется держать отдельно от логирования основной программы, где он используется.

Создайте свой логгер и пишите в него.

logging.NullHandler - обработчик, который "ничего не делает"

```
# somelib.py
import logging

log = logging.getLogger(__name__)           # во время import нашего модуля выполнится
 этот код
log.addHandler(logging.NullHandler())

# Example function (for testing)
def func():
    log.critical('A Critical Error!')
    log.debug('A debug message')
```

В этой конфигурации по умолчанию нет ни одного логера (и не будет логирования!):

```
>>> import somelib
>>> somelib.func()
>>>
```

Однако, как только логирующая система будет сконфигурирована, начнут появляться сообщения для лога:

```
>>> import logging
>>> logging.basicConfig()
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
>>>
```

Библиотеки - это особый случай логирования, потому что неизвестно, в каком окружении будет работать библиотека. В общем случае, не стоит писать библиотеку так, чтобы она сама конфигурировала логирующую систему, или делала какие-то предположения о существующей конфигурации логера. Лучше позаботьтесь об изоляции.

Вызов **getLogger(__name__)** создает объект логера с тем же именем, что и вызывающий модуль. Так как все модули имеют уникальные имена, получаем специальный объект логера, отделенный от других логеров.

Операция **log.addHandler(logging.NullHandler())** - привязывает null handler к только что созданному объекту логера. Этот null handler по умолчанию игнорирует все логирующие сообщения. Таким образом, если библиотека используется, но логер не был сконфигурирован, никакие сообщения или предупреждения никуда не будут записаны.

Стоит конфигурировать логер отдельной библиотеке независимо от других настроек логирования. Например:

```
>>> import logging
>>> logging.basicConfig(level=logging.ERROR)
>>> import somelib
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
>>> # Change the logging level for 'somelib' only
>>> logging.getLogger('somelib').level=logging.DEBUG
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
DEBUG:somelib:A debug message
```

Здесь корневой логер был настроен только чтобы пропускать сообщения с уровнем ERROR или выше. Однако, у логера модуля somelib был установлен уровень DEBUG. Локальные настройки более приоритетны, чем глобальные.

Эту особенность можно использовать для отладки отдельной библиотеки, не влючая отладку других частей.

Useful Handlers

In addition to the base Handler class, many useful subclasses are provided:

- StreamHandler instances send messages to streams (file-like objects).
- FileHandler instances send messages to disk files.
- BaseRotatingHandler is the base class for handlers that rotate log files at a certain point. It is not meant to be instantiated directly. Instead, use RotatingFileHandler or TimedRotatingFileHandler.
- RotatingFileHandler instances send messages to disk files, with support for maximum log file sizes and log file rotation.
- TimedRotatingFileHandler instances send messages to disk files, rotating the log file at certain timed intervals.
- SocketHandler instances send messages to TCP/IP sockets. Since 3.4, Unix domain sockets are also supported.
- DatagramHandler instances send messages to UDP sockets. Since 3.4, Unix domain sockets are also supported.
- SMTPHandler instances send messages to a designated email address.
- SysLogHandler instances send messages to a Unix syslog daemon, possibly on a remote machine.
- NTEventLogHandler instances send messages to a Windows NT/2000/XP event log.
- MemoryHandler instances send messages to a buffer in memory, which is flushed whenever specific criteria are met.
- HTTPHandler instances send messages to an HTTP server using either GET or POST semantics.
- WatchedFileHandler instances watch the file they are logging to. If the file changes, it is closed and reopened using the file name. This handler is only useful on Unix-like systems; Windows does not support the underlying mechanism used.
- QueueHandler instances send messages to a queue, such as those implemented in the queue or multiprocessing modules.
- NullHandler instances do nothing with error messages. They are used by library developers who want to use logging, but want to avoid the 'No handlers could be found for logger XXX' message which can be displayed if the library user has not configured logging. See Configuring Logging for a Library for more information.

Рецепты использования логера

Перевод Logging cookbook

Содержание:

Логирование в нескольких модулях

Если несколько раз вызвать `logging.getLogger('someLogger')` получим ссылку на один и тот же объект, даже если вызываем из разных модулей, но в рамках одного процесса интерпретатора питона.

Прикладной код может определять и конфигурировать родительский логер в одном модуле и создавать (но не конфигурировать) дочерний логер в отдельном модуле, и все вызовы в дочернем логере будут переданы родительскому логеру.

Главный модуль:

```
import logging
import auxiliary_module

# create logger with 'spam_application'
logger = logging.getLogger('spam_application')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
# add the handlers to the logger
logger.addHandler(fh)
logger.addHandler(ch)

logger.info('creating an instance of auxiliary_module.Auxiliary')
a = auxiliary_module.Auxiliary()
logger.info('created an instance of auxiliary_module.Auxiliary')
logger.info('calling auxiliary_module.Auxiliary.do_something')
a.do_something()
logger.info('finished auxiliary_module.Auxiliary.do_something')
logger.info('calling auxiliary_module.some_function()')
auxiliary_module.some_function()
logger.info('done with auxiliary_module.some_function()')
```

Модуль auxiliary:

```
import logging

# create logger
module_logger = logging.getLogger('spam_application.auxiliary')

class Auxiliary:
    def __init__(self):
        self.logger = logging.getLogger('spam_application.auxiliary.Auxiliary')
        self.logger.info('creating an instance of Auxiliary')

    def do_something(self):
        self.logger.info('doing something')
        a = 1 + 1
        self.logger.info('done doing something')

def some_function():
    module_logger.info('received a call to "some_function"')
```

получим:

```
2005-03-23 23:47:11,663 - spam_application - INFO -
   creating an instance of auxiliary_module.Auxiliary
2005-03-23 23:47:11,665 - spam_application.auxiliary.Auxiliary - INFO -
   creating an instance of Auxiliary
2005-03-23 23:47:11,665 - spam_application - INFO -
   created an instance of auxiliary_module.Auxiliary
2005-03-23 23:47:11,668 - spam_application - INFO -
   calling auxiliary_module.Auxiliary.do_something
2005-03-23 23:47:11,668 - spam_application.auxiliary.Auxiliary - INFO -
   doing something
2005-03-23 23:47:11,669 - spam_application.auxiliary.Auxiliary - INFO -
   done doing something
2005-03-23 23:47:11,670 - spam_application - INFO -
   finished auxiliary_module.Auxiliary.do_something
2005-03-23 23:47:11,671 - spam_application - INFO -
   calling auxiliary_module.some_function()
2005-03-23 23:47:11,672 - spam_application.auxiliary - INFO -
   received a call to 'some_function'
2005-03-23 23:47:11,673 - spam_application - INFO -
   done with auxiliary_module.some_function()
```

Логирование в нескольких threads

Никаких дополнительных усилий. Все работает. Работает и с большим количеством thread.

```
import logging
import threading
import time

def worker(arg):
    while not arg['stop']:
        logging.debug('Hi from myfunc')
        time.sleep(0.5)

def main():
    logging.basicConfig(level=logging.DEBUG, format='%(relativeCreated)6d %(threadName
)s %(message)s')
    info = {'stop': False}
    thread = threading.Thread(target=worker, args=(info,))
    thread.start()
    while True:
        try:
            logging.debug('Hello from main')
            time.sleep(0.75)
        except KeyboardInterrupt:
            info['stop'] = True
            break
    thread.join()

if __name__ == '__main__':
    main()
```

получим (пока не напечатаем stop):

```
   0 Thread-1 Hi from myfunc
   3 MainThread Hello from main
 505 Thread-1 Hi from myfunc
 755 MainThread Hello from main
1007 Thread-1 Hi from myfunc
1507 MainThread Hello from main
1508 Thread-1 Hi from myfunc
2010 Thread-1 Hi from myfunc
2258 MainThread Hello from main
2512 Thread-1 Hi from myfunc
3009 MainThread Hello from main
3013 Thread-1 Hi from myfunc
3515 Thread-1 Hi from myfunc
3761 MainThread Hello from main
4017 Thread-1 Hi from myfunc
4513 MainThread Hello from main
4518 Thread-1 Hi from myfunc
```

Много handler и formatter

Добавляйте сколько хотите:

```
import logging

logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
ch.setFormatter(formatter)
fh.setFormatter(formatter)
# add the handlers to logger
logger.addHandler(ch)
logger.addHandler(fh)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

Logging to multiple destinations

Хотим логировать и в файл, и на консоль. При этом с разными уровнями важности и в разном формате.

```
import logging

# set up logging to file - see previous section for more details
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s',
                    datefmt='%m-%d %H:%M',
                    filename='/temp/myapp.log',
                    filemode='w')
# define a Handler which writes INFO messages or higher to the sys.stderr
console = logging.StreamHandler()
console.setLevel(logging.INFO)
# set a format which is simpler for console use
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
# tell the handler to use this format
console.setFormatter(formatter)
# add the handler to the root logger
logging.getLogger('').addHandler(console)

# Now, we can log to the root logger, or any other logger. First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas in your
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

получим на консоль:

```
root        : INFO     Jackdaws love my big sphinx of quartz.
myapp.area1 : INFO     How quickly daft jumping zebras vex.
myapp.area2 : WARNING  Jail zesty vixen who grabbed pay from quack.
myapp.area2 : ERROR    The five boxing wizards jump quickly.
```

получим в файл (примерно):

```
10-22 22:19 root         INFO     Jackdaws love my big sphinx of quartz.
10-22 22:19 myapp.area1  DEBUG    Quick zephyrs blow, vexing daft Jim.
10-22 22:19 myapp.area1  INFO     How quickly daft jumping zebras vex.
10-22 22:19 myapp.area2  WARNING  Jail zesty vixen who grabbed pay from quack.
10-22 22:19 myapp.area2  ERROR    The five boxing wizards jump quickly.
```

Заметим, DEBUG сообщения попали только в файл. Остальные и на консоль, и в файл.

Вы можете использовать другие обработчики, а не только FileHandler и StreamHandler.

Configuration server example

```
import logging
import logging.config
import time
import os

# read initial config file
logging.config.fileConfig('logging.conf')

# create and start listener on port 9999
t = logging.config.listen(9999)
t.start()

logger = logging.getLogger('simpleExample')

try:
    # loop through logging calls to see the difference
    # new configurations make, until Ctrl+C is pressed
    while True:
        logger.debug('debug message')
        logger.info('info message')
        logger.warn('warn message')
        logger.error('error message')
        logger.critical('critical message')
        time.sleep(5)
except KeyboardInterrupt:
    # cleanup
    logging.config.stopListening()
    t.join()
```

Далее скрипт получает имя файла и посылает этот файл на сервер, чтобы этот файл стал новой конфигурацией логера. Перед посылкой файла посылается его длина.

```
#!/usr/bin/env python
import socket, sys, struct

with open(sys.argv[1], 'rb') as f:
    data_to_send = f.read()

HOST = 'localhost'
PORT = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print('connecting...')
s.connect((HOST, PORT))
print('sending config...')
s.send(struct.pack('>L', len(data_to_send)))
s.send(data_to_send)
s.close()
print('complete')
```

Dealing with handlers that block

Иногда вам нужно, чтобы обработчик не блокировал поток, в котором вы создаете сообщение. Пример такого сценария - веб приложение.

Например, SMTPHandler может долго обрабатывать сообщение по не зависящим от разработчика причинам. Любой сетевой хендлер может может слишком долго выполнять DNS запрос и это вы тоже не можете контролировать.

Решение состоит из двух частей.

В первой части используйте QueueHandler для тех логеров, в которые пишут из тредов с, чья производительность критична. Сообщения просто попадают в очередь (которую стоит сделать достаточность большой емкости или вообще без ограничения на ее размер). Запись в очередь обычно делается достаточно быстро. Однако, стоит ловить случай переполнения очереди.

Если вы разработчик библиотеки с тредами, чувствительными к производительности, удостоверьтесь, что эта особенность задокументирована (вместе с предложением присоединять только QueueHandler к вашим логерам), чтобы помочь другим программистам, которые будут использовать ваш код.

Второй частью решения является QueueListener, который разработан как соотвествующая часть к QueueHandler. QueueListener очень простой: он передает очередь и некоторые обработчики и стартует внутренний thread, который слушает его очередь, получает LogRecords, посланные из QueueHandler (или быть может из другого источника LogRecords). Очередные LogRecords удаляются из очереди и передаются обработчикам для обработки.

Выгода в отдельном QueueListener классе в том, что вы можете использовать один и тот же экземпляр для обслуживания множества QueueHandler. Это более resource-friendly, чем, скажем иметь версию QueueHandler в отдельном треде.

Пример использования этих двух классов. (import опущены):

```
que = queue.Queue(-1)  # no limit on size
queue_handler = QueueHandler(que)
handler = logging.StreamHandler()
listener = QueueListener(que, handler)
root = logging.getLogger()
root.addHandler(queue_handler)
formatter = logging.Formatter('%(threadName)s: %(message)s')
handler.setFormatter(formatter)
listener.start()
# The log output will display the thread which generated
# the event (the main thread) rather than the internal
# thread which monitors the internal queue. This is what
# you want to happen.
root.warning('Look out!')
listener.stop()
```

получим:

```
MainThread: Look out!
```

Использование 1 QueueListener и множества QueueHandler смотрите в примере Logging to a single file from multiple processes

Sending and receiving logging events across a network

Допустим, вы хотите посылать логирующие события по сети и обрабатывать их при получении. Проще всего добавить SocketHandler к root логеру в тех местах, где события посылаются:

Тут посылаем:

```
import logging, logging.handlers

rootLogger = logging.getLogger('')
rootLogger.setLevel(logging.DEBUG)
socketHandler = logging.handlers.SocketHandler('localhost',
                    logging.handlers.DEFAULT_TCP_LOGGING_PORT)
# don't bother with a formatter, since a socket handler sends the event as
# an unformatted pickle
rootLogger.addHandler(socketHandler)

# Now, we can log to the root logger, or any other logger. First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas in your
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

Тут получаем и обрабатываем с использованием модуля `socketserver`:

```
import pickle
import logging
import logging.handlers
import socketserver
import struct


class LogRecordStreamHandler(socketserver.StreamRequestHandler):
    """Handler for a streaming logging request.

    This basically logs the record using whatever logging policy is
    configured locally.
    """

    def handle(self):
        """
        Handle multiple requests - each expected to be a 4-byte length,
        followed by the LogRecord in pickle format. Logs the record
        according to whatever policy is configured locally.
        """
        while True:
            chunk = self.connection.recv(4)
            if len(chunk) < 4:
                break
            slen = struct.unpack('>L', chunk)[0]
```

```
            chunk = self.connection.recv(slen)
            while len(chunk) < slen:
                chunk = chunk + self.connection.recv(slen - len(chunk))
            obj = self.unPickle(chunk)
            record = logging.makeLogRecord(obj)
            self.handleLogRecord(record)

    def unPickle(self, data):
        return pickle.loads(data)

    def handleLogRecord(self, record):
        # if a name is specified, we use the named logger rather than the one
        # implied by the record.
        if self.server.logname is not None:
            name = self.server.logname
        else:
            name = record.name
        logger = logging.getLogger(name)
        # N.B. EVERY record gets logged. This is because Logger.handle
        # is normally called AFTER logger-level filtering. If you want
        # to do filtering, do it at the client end to save wasting
        # cycles and network bandwidth!
        logger.handle(record)

class LogRecordSocketReceiver(socketserver.ThreadingTCPServer):
    """
    Simple TCP socket-based logging receiver suitable for testing.
    """

    allow_reuse_address = True

    def __init__(self, host='localhost',
                 port=logging.handlers.DEFAULT_TCP_LOGGING_PORT,
                 handler=LogRecordStreamHandler):
        socketserver.ThreadingTCPServer.__init__(self, (host, port), handler)
        self.abort = 0
        self.timeout = 1
        self.logname = None

    def serve_until_stopped(self):
        import select
        abort = 0
        while not abort:
            rd, wr, ex = select.select([self.socket.fileno()],
                                       [], [],
                                       self.timeout)
            if rd:
                self.handle_request()
            abort = self.abort

def main():
    logging.basicConfig(
        format='%(relativeCreated)5d %(name)-15s %(levelname)-8s %(message)s')
```

```
    tcpserver = LogRecordSocketReceiver()
    print('About to start TCP server...')
    tcpserver.serve_until_stopped()

if __name__ == '__main__':
    main()
```

Сначала должен стартовать сервер, потом клиенты.

На стороне клиентов ничего в консоль не пишется.

На стороне сервера получаем:

```
About to start TCP server...
   59 root            INFO     Jackdaws love my big sphinx of quartz.
   59 myapp.area1     DEBUG    Quick zephyrs blow, vexing daft Jim.
   69 myapp.area1     INFO     How quickly daft jumping zebras vex.
   69 myapp.area2     WARNING  Jail zesty vixen who grabbed pay from quack.
   69 myapp.area2     ERROR    The five boxing wizards jump quickly.
```

Note that there are some security issues with pickle in some scenarios. If these affect you, you can use an alternative serialization scheme by overriding the makePickle() method and implementing your alternative there, as well as adapting the above script to use your alternative serialization.

Adding contextual information to your logging output

Sometimes you want logging output to contain contextual information in addition to the parameters passed to the logging call. For example, in a networked application, it may be desirable to log client-specific information in the log (e.g. remote client's username, or IP address). Although you could use the extra parameter to achieve this, it's not always convenient to pass the information in this way. While it might be tempting to create Logger instances on a per-connection basis, this is not a good idea because these instances are not garbage collected. While this is not a problem in practice, when the number of Logger instances is dependent on the level of granularity you want to use in logging an application, it could be hard to manage if the number of Logger instances becomes effectively unbounded.

Логирование из нескольких потоков

Although logging is thread-safe, and logging to a single file from multiple threads in a single process is supported, logging to a single file from multiple processes is not supported, because there is no standard way to serialize access to a single file across multiple processes in Python. If you need to log to a single file from multiple processes, one way of doing this is to have all the processes log to a SocketHandler, and have a separate process which implements a socket server which reads from the socket and logs to file. (If you prefer, you can dedicate one thread in one of the existing processes to perform this function.) This section documents this approach in more detail and includes a working socket receiver which can be used as a starting point for you to adapt in your own applications.

If you are using a recent version of Python which includes the multiprocessing module, you could write your own handler which uses the Lock class from this module to serialize access to the file from your processes. The existing FileHandler and subclasses do not make use of multiprocessing at present, though they may do so in the future. Note that at present, the multiprocessing module does not provide working lock functionality on all platforms (see https://bugs.python.org/issue3770).

Alternatively, you can use a Queue and a QueueHandler to send all logging events to one of the processes in your multi-process application. The following example script demonstrates how you can do this; in the example a separate listener process listens for events sent by other processes and logs them according to its own logging configuration. Although the example only demonstrates one way of doing it (for example, you may want to use a listener thread rather than a separate listener process – the implementation would be analogous) it does allow for completely different logging configurations for the listener and the other processes in your application, and can be used as the basis for code meeting your own specific requirements:

```
# You'll need these imports in your own code
import logging
import logging.handlers
import multiprocessing

# Next two import lines for this demo only
from random import choice, random
import time

#
# Because you'll want to define the logging configurations for listener and workers, t
he
# listener and worker process functions take a configurer parameter which is a callable

# for configuring logging for that process. These functions are also passed the queue,
# which they use for communication.
#
# In practice, you can configure the listener however you want, but note that in this
# simple example, the listener does not apply level or filter logic to received record
```

```
s.
# In practice, you would probably want to do this logic in the worker processes, to av
oid
# sending events which would be filtered out between processes.
#
# The size of the rotated files is made small so you can see the results easily.
def listener_configurer():
    root = logging.getLogger()
    h = logging.handlers.RotatingFileHandler('mptest.log', 'a', 300, 10)
    f = logging.Formatter('%(asctime)s %(processName)-10s %(name)s %(levelname)-8s %(m
essage)s')
    h.setFormatter(f)
    root.addHandler(h)

# This is the listener process top-level loop: wait for logging events
# (LogRecords)on the queue and handle them, quit when you get a None for a
# LogRecord.
def listener_process(queue, configurer):
    configurer()
    while True:
        try:
            record = queue.get()
            if record is None:  # We send this as a sentinel to tell the listener to q
uit.
                break
            logger = logging.getLogger(record.name)
            logger.handle(record)  # No level or filter logic applied - just do it!
        except Exception:
            import sys, traceback
            print('Whoops! Problem:', file=sys.stderr)
            traceback.print_exc(file=sys.stderr)

# Arrays used for random selections in this demo

LEVELS = [logging.DEBUG, logging.INFO, logging.WARNING,
          logging.ERROR, logging.CRITICAL]

LOGGERS = ['a.b.c', 'd.e.f']

MESSAGES = [
    'Random message #1',
    'Random message #2',
    'Random message #3',
]

# The worker configuration is done at the start of the worker process run.
# Note that on Windows you can't rely on fork semantics, so each process
# will run the logging configuration code when it starts.
def worker_configurer(queue):
    h = logging.handlers.QueueHandler(queue)  # Just the one handler needed
    root = logging.getLogger()
    root.addHandler(h)
    # send all messages, for demo; no other level or filter logic applied.
```

```
    root.setLevel(logging.DEBUG)

# This is the worker process top-level loop, which just logs ten events with
# random intervening delays before terminating.
# The print messages are just so you know it's doing something!
def worker_process(queue, configurer):
    configurer(queue)
    name = multiprocessing.current_process().name
    print('Worker started: %s' % name)
    for i in range(10):
        time.sleep(random())
        logger = logging.getLogger(choice(LOGGERS))
        level = choice(LEVELS)
        message = choice(MESSAGES)
        logger.log(level, message)
    print('Worker finished: %s' % name)

# Here's where the demo gets orchestrated. Create the queue, create and start
# the listener, create ten workers and start them, wait for them to finish,
# then send a None to the queue to tell the listener to finish.
def main():
    queue = multiprocessing.Queue(-1)
    listener = multiprocessing.Process(target=listener_process,
                                       args=(queue, listener_configurer))
    listener.start()
    workers = []
    for i in range(10):
        worker = multiprocessing.Process(target=worker_process,
                                         args=(queue, worker_configurer))
        workers.append(worker)
        worker.start()
    for w in workers:
        w.join()
    queue.put_nowait(None)
    listener.join()

if __name__ == '__main__':
    main()
```

A variant of the above script keeps the logging in the main process, in a separate thread:

```
import logging
import logging.config
import logging.handlers
from multiprocessing import Process, Queue
import random
import threading
import time

def logger_thread(q):
    while True:
```

```
        record = q.get()
        if record is None:
            break
        logger = logging.getLogger(record.name)
        logger.handle(record)


def worker_process(q):
    qh = logging.handlers.QueueHandler(q)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(qh)
    levels = [logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
              logging.CRITICAL]
    loggers = ['foo', 'foo.bar', 'foo.bar.baz',
               'spam', 'spam.ham', 'spam.ham.eggs']
    for i in range(100):
        lvl = random.choice(levels)
        logger = logging.getLogger(random.choice(loggers))
        logger.log(lvl, 'Message no. %d', i)

if __name__ == '__main__':
    q = Queue()
    d = {
        'version': 1,
        'formatters': {
            'detailed': {
                'class': 'logging.Formatter',
                'format': '%(asctime)s %(name)-15s %(levelname)-8s %(processName)-10s
%(message)s'
            }
        },
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO',
            },
            'file': {
                'class': 'logging.FileHandler',
                'filename': 'mplog.log',
                'mode': 'w',
                'formatter': 'detailed',
            },
            'foofile': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-foo.log',
                'mode': 'w',
                'formatter': 'detailed',
            },
            'errors': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-errors.log',
                'mode': 'w',
```

```
                'level': 'ERROR',
                'formatter': 'detailed',
            },
        },
        'loggers': {
            'foo': {
                'handlers': ['foofile']
            }
        },
        'root': {
            'level': 'DEBUG',
            'handlers': ['console', 'file', 'errors']
        },
    }
    workers = []
    for i in range(5):
        wp = Process(target=worker_process, name='worker %d' % (i + 1), args=(q,))
        workers.append(wp)
        wp.start()
    logging.config.dictConfig(d)
    lp = threading.Thread(target=logger_thread, args=(q,))
    lp.start()
    # At this point, the main process could do some useful work of its own
    # Once it's done that, it can wait for the workers to terminate...
    for wp in workers:
        wp.join()
    # And now tell the logging thread to finish up, too
    q.put(None)
    lp.join()
```

This variant shows how you can e.g. apply configuration for particular loggers - e.g. the foo logger has a special handler which stores all events in the foo subsystem in a file mplog-foo.log. This will be used by the logging machinery in the main process (even though the logging events are generated in the worker processes) to direct the messages to the appropriate destinations.

Задачи

1. Аргументы командной строки и конфиг-файл

Написать программу add.py, которая принимает в командной строке 2 аргумента (числа) и производит над ними одну из арифметических операций: +, -, *, а затем печатает выражение и результат. Например:

```
python add.py 2 3
2 + 3 = 5
```

- По умолчанию это операция +.
- в аргументах командной строки с ключом -c может быть задан конфигуационный файл, и в нем определена операция (формат конфиг-файла - на усмотрение программиста).
- В аргументах может быть задана операция --add, --sub, --mul. Что будет если задать несколько операций - на усмотрение программиста.

Способы задания перечислены по возрастанию приоритета операции. Т.е. если в аргументах указано --sub, то все остальное игнорируем.

2. Простое логирование

Добавить логирование на консоль, которое будет писать:

- Программа начала работу
- прочитан и применен конфиг-файл
- разобран и применен аргумент командной строки
- выполнена операция
- программа сейчас закончит работу

2.a Формат логирования времени

Писать время логирования операции в формате yyyy/mm/dd hh:mm:ss

3. Деление

Писать лог в файл.

Добавить в список операций деление --div. При попытке делить на ноль логировать ошибку вместе со стектрейсом.

4. Ошибки на экран

Сделать так, чтобы все ошибки и выше еще и печатались на stderr.

C - расширения

Источники

- документация
- tutorialspoint
- ctypes документация
- SWIG tutorial
  - SWIG on Windows
- C/Python API
- Intermediate python
- SciPy cookbook
- Using C and Fortran code with Python
- Numba vs. Cython: Take 2
- Python cookbook, chapter 15, C extensions
- (Python Cookbook by David Ascher, Alex Martelli) [https://www.safaribooksonline.com/library/view/python-cookbook/0596001673/ch16.html] Chapter 16. Extending and Embedding

Зачем использовать вставки другого кода

- C работает быстрее;
- нужна конкретная библиотека на C и не хочется переписывать ее на питон;
- нужен низкоуровневый интерфейс управления ресурсами для работы с памятью и файлами.

Что использовать для вызова C-функции

Один из механизмов:

- ctypes
- SWIG
- Python/C API

Как сделать файл-библиотеку на языке C или C++

SWIG FAQ

C-код, который мы будем дальше использовать

```
/* sample.c */
#include <math.h>
/* Compute the greatest common divisor */
int gcd(int x, int y) {
    int g = y;
    while (x > 0) {
        g = x;
        x = y % x;
        y = g;
    }
    return g;
}
/* Test if (x0,y0) is in the Mandelbrot set or not */
int in_mandel(double x0, double y0, int n) {
    double x=0,y=0,xtemp;
    while (n > 0) {
        xtemp = x*x - y*y + x0;
        y = 2*x*y + y0;
        x = xtemp;
        n -= 1;
        if (x*x + y*y > 4) return 0;
    }
    return 1;
}
/* Divide two numbers */
int divide(int a, int b, int *remainder) {
    int quot = a / b;
    *remainder = a % b;
    return quot;
}
/* Average values in an array */
double avg(double *a, int n) {
    int i;
    double total = 0.0;
    for (i = 0; i < n; i++) {
        total += a[i];
    }
    return total / n;
}
/*A C data structure */
typedef struct Point {
    double x,y;
} Point;
/* Function involving a C data structure */
double distance(Point *p1, Point *p2) {
    return hypot(p1->x - p2->x, p1->y - p2->y);
}
```

- *gdc* и *is_mandel* - простые функции от int и double, которые возвращают значения;
- *divide* - возвращает, по сути, 2 числа - частное и записывает остаток от деления по указанному адресу;

- *avg* - перебирает массив, переданный как указатель;
- *Point* и *distance* - работают со структурами.

Пусть прототипы функций файла `sample.c` заданы в `sample.h` и сам файл собран в библиотеку `libsample.so` .

Диагностика segmentation faults

Что-то пошло не так и программа упала. Хочется получить информативный trace падения. Воспользуемся модулем **faulthandler**

Из кода питона:

```
import faulthandler
faulthandler.enable()
```

или запустим питон с опцией `-Xfaulthandler` :

```
bash % python3 -Xfaulthandler program.py
```

или определите переменную окружения **PYTHONFAULTHANDLER**.

Если ваша программа с использованием C-кода упала, вы получите сообщение вида:

```
Fatal Python error: Segmentation fault
Current thread 0x00007fff71106cc0:
File "example.py", line 6 in foo
File "example.py", line 10 in bar
File "example.py", line 14 in spam
File "example.py", line 19 in <module>
Segmentation fault
```

Далее воспользуйтесь питон дебагером **pdb** и C-дебагером (например, gdb) для исследования C-части.

It should be noted that certain kinds of errors in C may not be easily recoverable. For example, if a C extension trashes the stack or program heap, it may render faulthan dler inoperable and you'll simply get no output at all (other than a crash). Obviously, your mileage may vary.

CTypes

Самый простой способ - ctypes

- Плюс: не нужно изменять C-код.

Проверьте, что библиотека и ваш интерпретатор питона собраны для одной и той же архитектуры, у них совпадает размер машинного слова и тп.

C-совместимые типы данных и функции для загрузки DLL.

Таблица типов

ctypes type	C type	Python type
c_bool	_Bool	bool
c_char	char	1-character bytes object
c_wchar	wchar_t	1-character string
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 or long long	int
c_ulonglong	unsigned __int64 or unsigned long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t or Py_ssize_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL terminated)	bytes object or None
c_wchar_p	wchar_t * (NUL terminated)	string or None
c_void_p	void *	int or None

Пример: сумма 2 чисел на языке C:

```
// Простой C-файл - суммируем целые и действительные числа

int add_int(int, int);
float add_float(float, float);

int add_int(int num1, int num2){
    return num1 + num2;
}

float add_float(float num1, float num2){
    return num1 + num2;
}
```

Скомпилируем файл в библиотеку .so (.dll под Windows). Получим adder.so.

```
# Для Linux
$  gcc -shared -Wl,-soname,adder -o adder.so -fPIC add.c

# Для Mac
$ gcc -shared -Wl,-install_name,adder.so -o adder.so -fPIC add.c
```

Код на питоне:

```
from ctypes import *

# Загружаем библиотеку
adder = CDLL('./adder.so')

# Находим сумму целых чисел
# Самый простой случай - аргументы по умолчанию int и возвращается по умолчанию int
res_int = adder.add_int(4,5)
print("Сумма 4 и 5 = " + str(res_int))

# Находим сумму действительных чисел
# нужно описать типы аргументов и возвращаемого значения
a = c_float(5.5)
b = c_float(4.1)

add_float = adder.add_float
add_float.restype = c_float
print("Сумма 5.5 и 4.1 = " + str(add_float(a, b)))
```

запускаем и получаем:

```
Сумма 4 и 5 = 9
Сумма 5.5 и 4.1 = 9.600000381469727
```

Пример sample.c

Пусть shared library собрана и помещена в той же директории, что и питоновский файл.

Напишем питоновский модуль-обертку для этой библиотеки:

```
# sample.py
import ctypes
import os

# Try to locate the .so file in the same directory as this file
_file = 'libsample.so'
_path = os.path.join(*(os.path.split(__file__)[:-1] + (_file,)))
_mod = ctypes.cdll.LoadLibrary(_path)

# int gcd(int, int)
gcd = _mod.gcd
gcd.argtypes = (ctypes.c_int, ctypes.c_int)
gcd.restype = ctypes.c_int

# int in_mandel(double, double, int)
in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_int)
in_mandel.restype = ctypes.c_int

# int divide(int, int, int *)
# напишем функцию-обертку, которая возвращает два значения
_divide = _mod.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
_divide.restype = ctypes.c_int
def divide(x, y):
    rem = ctypes.c_int()
    quot = _divide(x, y, rem)
    return quot,rem.value

# void avg(double *, int n)
# Define a special type for the 'double *' argument
class DoubleArrayType:
    def from_param(self, param):
        typename = type(param).__name__
        if hasattr(self, 'from_' + typename):
            return getattr(self, 'from_' + typename)(param)
        elif isinstance(param, ctypes.Array):
            return param
        else:
            raise TypeError("Can't convert %s" % typename)

    # Cast from array.array objects
    def from_array(self, param):
        if param.typecode != 'd':
            raise TypeError('must be an array of doubles')
        ptr, _ = param.buffer_info()
```

```
        return ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))

    # Cast from lists/tuples
    def from_list(self, param):
        val = ((ctypes.c_double)*len(param))(*param)
        return val

    from_tuple = from_list

    # Cast from a numpy array
    def from_ndarray(self, param):
        return param.ctypes.data_as(ctypes.POINTER(ctypes.c_double))

DoubleArray = DoubleArrayType()

_avg = _mod.avg
_avg.argtypes = (DoubleArray, ctypes.c_int)
_avg.restype = ctypes.c_double
def avg(values):
    return _avg(values, len(values))

# struct Point { }
class Point(ctypes.Structure):
    _fields_ = [
        ('x', ctypes.c_double),
        ('y', ctypes.c_double)
    ]
# double distance(Point *, Point *)
distance = _mod.distance
distance.argtypes = (ctypes.POINTER(Point), ctypes.POINTER(Point))
distance.restype = ctypes.c_double
```

Теперь можно использовать эти питоновские функции, импортируя модуль-обертку:

```
>>> import sample
>>> sample.gcd(35,42)
7
>>> sample.in_mandel(0,0,500)
1
>>> sample.in_mandel(2.0,1.0,500)
0
>>> sample.divide(42,8)
(5, 2)
>>> sample.avg([1,2,3])
2.0
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
4.242640687119285
```

Где расположена библиотека?

Библиотека должна лежать в том месте, где питоновский код может ее найти. Как вариант, можем положить ее в той же директории. В примере эта директория добывается из переменной `__file__` .

Если в другом месте - настраивайте пути поиска.

Если вы хотите взять стандартную библиотеку, то можно использовать функцию **ctypes.util.find_library()**

```
>>> from ctypes.util import find_library
>>> find_library('m')
'/usr/lib/libm.dylib'
>>> find_library('pthread')
'/usr/lib/libpthread.dylib'
>>> find_library('sample')
'/usr/local/lib/libsample.so'
```

Библиотека загружается функцией **ctypes.cdll.LoadLibrary**(*path*), которой передают путь к библиотеке.

Спецификация типов аргументов и возвращаемого значения

Некоторые функции можно определить один-в-один:

```
# int in_mandel(double, double, int)
in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_int)
in_mandel.restype = ctypes.c_int
```

Передача целочисленных переменных по указателю (изменение нескольких значений)

Некоторые содержат приемы, которые не работают в питоновском коде и "питонично" писать по-другому.

В функции divide() частное возвращается, а остаток записывается в переменную, адрес которой передан последним аргументом. Если мы переведем слово-в-слово, то код не будет работать:

```
>>> divide = _mod.divide
>>> divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
>>> x = 0
>>> divide(10, 3, x)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 3: <class 'TypeError'>: expected LP_c_int instance inst
ead of int
```

int - это неизменяемые объекты в питоне, поэтому по адресу не удается поменять значение.

Чтобы код заработал, нужно сделать x нужного типа.

```
>>> x = ctypes.c_int()
>>> divide(10, 3, x)
3
>>> x.value
1
```

В функцию передается ссылка на объект типа c_int, который содержит мутабельные целые числа. Полученное число лежит в поле value.

Но лучше написать "питоническую" функцию-обертку, которая в питоновском стиле вернет кортеж из частного и остатка. И далее использовать эту функцию divide.

```
# int divide(int, int, int *)
_divide = _mod.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
_divide.restype = ctypes.c_int

def divide(x, y):
    rem = ctypes.c_int()
    quot = _divide(x,y,rem)
    return quot, rem.value
```

Массивы

Функция avr хочет получить массив дробных чисел. Но что есть "массив" в питоне? Список? Кортеж? Массив numpy? Массив из модуля array?

Для обработки всех этих возможностей сделан класс DoubleArrayType. Метод **from_param**() превращает питоновский тип в подходящий объект ctypes (например, в указатель на ctypes.c_double). Вы пишите его так, как вам хочется.

Цитата из документации по ctypes:

If you have defined your own classes which you pass to function calls, you have to implement a from_param() class method for them to be able to use them in the argtypes sequence. The from_param() class method receives the Python object passed to the function call, it should do a typecheck or whatever is needed to make sure this object is acceptable, and then return the object itself, its _as_parameter_ attribute, or whatever you want to pass as the C function argument in this case. Again, the result should be an integer, string, bytes, a ctypes instance, or an object with an _as_parameter_ attribute.

Для этого определяется тип передаваемого параметра и вызывается метод, который обрабатывает именно этот тип. Так для **списков и кортежей** питона вызывается метод from_list (обратите внимание, как реализуется, что для кортежей вызывается тот же метод). from_list() преобразует param в массив ctypes.

Пример преобразования списка в массив ctypes:

```
>>> nums = [1, 2, 3]
>>> a = (ctypes.c_double * len(nums))(*nums)
>>> a
<__main__.c_double_Array_3 object at 0x10069cd40>
>>> a[0]
1.0
>>> a[1]
2.0
>>> a[2]
3.0
>>>
```

Для массивов из пакета **array** метод извлекает хранящийся внутри объекта адрес массива чисел. Смотрим, как можно его извлечь:

```
>>> import array
>>> a = array.array('d',[1,2,3])
>>> a
array('d', [1.0, 2.0, 3.0])
>>> ptr, _ = a.buffer_info()
>>> ptr
4298687200
>>> ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))
<__main__.LP_c_double object at 0x10069cd40>
```

Из **numpy.array** - как показано в from_ndarray().

Протестируем работу функции avr на разных типах массивов:

```
>>> import sample
>>> sample.avg([1,2,3])
2.0
>>> sample.avg((1,2,3))
2.0
>>> import array
>>> sample.avg(array.array('d',[1,2,3]))
2.0
>>> import numpy
>>> sample.avg(numpy.array([1.0,2.0,3.0]))
2.0
```

Структуры

Для передачи структуры языка C сделаем класс, наследующий классу **ctypes.Structure** и определим соответствующие поля нужных типов:

```
class Point(ctypes.Structure):
    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]
```

Далее используем этот класс везде, где нужен экземпляр этой структуры:

```
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> p1.x
1.0
>>> p1.y
2.0
>>> sample.distance(p1,p2)
4.242640687119285
```

Подключение стандартных библиотека

Стандартная MS C библиотека:

```
>>> from ctypes import *
>>> print(windll.kernel32)
<WinDLL 'kernel32', handle ... at ...>
>>> print(cdll.msvcrt)
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Note: Accessing the standard C library through cdll.msvcrt will use an outdated version of the library that may be incompatible with the one being used by Python. Where possible, use native Python functionality, or else import and use the msvcrt module.

На Linux:

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

Возвращаемое значение

Вызовем стандартную функцию strchr.

```
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p    # c_char_p is a pointer to a string
>>> strchr(b"abcdef", ord("d"))
b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
>>>
```

Заметьте, когда нужно вернуть NULL, возвращается None.

Можно определить типы аргументов функции:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
'def'
>>>
```

Объект питона в виде аргумента

Определите в классе поле **_as_parameter_**. В нем должен быть один из приемлемых типов: число, строка или байты.

If you don’t want to store the instance’s data in the `_as_parameter_` instance variable, you could define a property which makes the attribute available on request.

```
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

Итого о ctypes

Удобен для маленьких вставок. Для больших библиотек придется много времени тратить на написание как вызывать фукции (функции-обертки, классы).

Придется вникать в содержимое кода на C. Легко ошибиться с указателями и выйти за границы памяти.

SWIG

SWIG on Windows

В Simplified Wrapper and Interface Generator (SWIG) нужно написать отдельный файл, который описывает интерфейс. Этот файл будет передаваться в утилиту командной строки SWIG.

Обычно его не используют, ибо сложно. Но если у вас много языков, откуда нужно доступаться к C-коду, то используйте SWIG.

Пример из SWIG tutorial

C-файл `example.c` содержит разные функции и переменные:

```
#include <time.h>
double My_variable = 3.0;

int fact(int n) {
    if (n <= 1) return 1;
    else return n*fact(n-1);
}

int my_mod(int x, int y) {
    return (x%y);
}

char *get_time()
{
    time_t ltime;
    time(&ltime);
    return ctime(&ltime);
}
```

Файл, описывающий интерфейс `example.i`. Он не будет изменяться в зависимости от языка, на который вы хотите портировать свой C-код:

```
/* example.i */
%module example
%{
/* Помещаем сюда заголовочные файлы или объявления функций */
extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
%}

extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
```

Компиляция (заметьте, питон 2.1, указывайте правильную директорию к вашему актуальному питону):

```
unix % swig -python example.i
unix % gcc -c example.c example_wrap.c -I/usr/local/include/python2.1
unix % ld -shared example.o example_wrap.o -o _example.so
```

Python:

```
>>> import example
>>> example.fact(5)
120
>>> example.my_mod(7,3)
1
>>> example.get_time()
'Sun Feb 11 23:01:07 1996'
>>>
```

C/Python API

C/Python API

Работает с объектами

Пишем специальный C-код для работы с питоном.

Все объекты Python представляются как структуры PyObject и заголовочный файл `Python.h` предоставляет различные функции для работы с объектами. Например, если PyObject одновременно PyListType (список), то мы можем использовать функцию PyList_Size(), чтобы получить длину списка. Это эквивалентно коду len(some_list) в Python. Большинство основных функций/операторов для стандартных Python объектов доступны в C через Python.h.

Пример

Давайте напишем C-библиотеку для суммирования всех элементов списка Python (все элементы являются числами). Начнем с интерфейса, который мы хотим иметь в итоге. Вот Python-файл, использующий пока отсутствующую C-библиотеку:

```
# Это не простой Python import addList это C-библиотека
import addList

l = [1,2,3,4,5]
print("Сумма элементов списка - " + str(l) + " = " +  str(addList.add(l)))
```

C-код `adder.c`:

```
// Python.h содержит все необходимые функции, для работы с объектами Python
#include <Python.h>

// Эту функцию мы вызываем из Python кода
static PyObject* addList_add(PyObject* self, PyObject* args){

  PyObject * listObj;

  // Входящие аргументы находятся в кортеже
  // В нашем случае есть только один аргумент - список, на который мы будем
  // ссылаться как listObj
  if (! PyArg_ParseTuple( args, "O", &listObj))
    return NULL;

  // Длина списка
  long length = PyList_Size(listObj);
```

```
  // Проходимся по всем элементам
  int i, sum =0;
  for(i = 0; i < length; i++){
    // Получаем элемент из списка - он также Python-объект
    PyObject* temp = PyList_GetItem(listObj, i);
    // Мы знаем, что элемент это целое число - приводим его к типу C long
    long elem = PyInt_AsLong(temp);
    sum += elem;
  }

  // Возвращаемое в Python-код значение также Python-объект
  // Приводим C long к Python integer
  return Py_BuildValue("i", sum);
}

// Немного документации для 'add'
static char addList_docs[] =
    "add( ): add all elements of the list\n";

/*
Эта таблица содержит необходимую информацию о функциях модуля
<имя функции в модуле Python>, <фактическая функция>,
<ожидаемые типы аргументов функции>, <документация функции>
*/
static PyMethodDef addList_funcs[] = {
    {"add", (PyCFunction)addList_add, METH_VARARGS, addList_docs},
    {NULL, NULL, 0, NULL}
};

/*
addList имя модуля и это блок его инициализации.
<желаемое имя модуля>, <таблица информации>, <документация модуля>
*/
PyMODINIT_FUNC initaddList(void){
    Py_InitModule3("addList", addList_funcs,
                   "Add all the lists");
}
```

- Заголовочный файл `Python.h` содержит все требуемые типы (для представления типов объектов в Python) и определения функций (для работы с Python-объектами).
- Дальше мы пишем функцию, которую собираемся вызывать из Python. По соглашению, имя функции принимается {module-name}_{function-name}, которое в нашем случае - addList_add. Подробнее об этой функции будет дальше.
- Затем заполняем таблицу, которая содержит всю необходимую информацию о функциях, которые мы хотим иметь в модуле. Каждая строка относится к функции, последняя - контрольное значение (строка из null элементов). Затем идёт блок инициализации модуля - PyMODINIT_FUNC init{module-name}.

Функция addList_add принимает аргументы типа PyObject (args также является кортежем, но поскольку в Python всё является объектами, мы используем унифицированный тип PyObject). Мы парсим входные аргументы (фактически, разбиваем кортеж на отдельные элементы) при помощи PyArg_ParseTuple(). Первый параметр является аргументом для парсинга. Второй аргумент - строка, регламентирующая процесс парсинга элементов кортежа args. Знак на N-ой позиции строки сообщает нам тип N-ого элемента кортежа args, например - 'i' значит integer, 's' - строка и 'O' - Python-объект. Затем следует несколько аргументов, где мы хотели бы хранить выходные элементы PyArg_ParseTuple(). Число этих аргументов равно числу аргументов, которые планируется передавать в функцию модуля и их позиционность должна соблюдаться. Например, если мы ожидаем строку, целое число и список в таком порядке, сигнатура функции будет следующего вида:

```
int n;
char *s;
PyObject* list;
PyArg_ParseTuple(args, "isO", &n, &s, &list);
```

В данном случае, нам нужно извлечь только объект списка и сохранить его в переменной listObj. Затем мы используем функцию PyList_Size() чтобы получить длину списка. Логика совпадает с len(some_list) в Python.

Теперь мы итерируем по списку, получая элементы при помощи функции PyLint_GetItem(list, index). Так мы получаем PyObject*. Однако, поскольку мы знаем, что Python-объекты еще и PyIntType, то используем функцию PyInt_AsLong(PyObj *) для получения значения. Выполняем процедуру для каждого элемента и получаем сумму. Сумма преобразуется в Python-объект и возвращается в Python-код при помощи Py_BuildValue(). Аргумент "i" означает, что возвращаемое значение имеет тип integer.

В заключение мы собираем C-модуль. Сохраните следующий код как файл setup.py:

```
# Собираем модули

from distutils.core import setup, Extension

setup(name='addList', version='1.0',\
      ext_modules=[Extension('addList', ['adder.c'])])
```

Запускаем:

```
python setup.py install
```

Это соберёт и установит C-файл в Python-модуль, который нам требуется.

Теперь осталось только протестировать работоспособность:

```
# Модуль, вызывающий C-код
import addList

l = [1,2,3,4,5]
print("Сумма элементов списка - " + str(l) + " = " +  str(addList.add(l)))
```

запускаем:

```
Сумма элементов списка - [1, 2, 3, 4, 5] = 15
```

В итоге, как вы можете видеть, мы получили наше первое C-расширение, использующее Python.h API. Этот метод может показаться сложным, однако с практикой вы поймёте его удобство.

Cython

- cython.org

Concurrency: процессы (process) и потоки (thread)

Источники

- (threading)[https://docs.python.org/3/library/threading.html] - документация
- (subprocess)[https://docs.python.org/3/library/subprocess.html] - документация
- Саммерфильд, глава 9
- Корутины:
  - (intermediate python)[https://lancelote.gitbooks.io/intermediate-python/content/book/coroutines.html]
  - (Coroutines)[http://www.dabeaz.com/coroutines/Coroutines.pdf] - презентация David Beazley http://www.dabeaz.com Presented at PyCon'2009, Chicago, Illinois
  - (Combinatorial Generation Using Coroutines With Examples in Python) [https://sahandsaba.com/combinatorial-generation-using-coroutines-in-python.html]
- асинхронность:
  - (Understanding Asynchronous IO With Python 3.4's Asyncio And Node.js) [https://sahandsaba.com/understanding-asyncio-node-js-python-3-4.html]
- GIL
  - (New GIL)[http://www.dabeaz.com/python/NewGIL.pdf] - презентация David Beazley
- (pymotw)[https://pymotw.com/3/concurrency.html]
  - (pymotw)[https://pymotw.com/3/threading/index.html]
  - (pymotw)[https://pymotw.com/3/subprocess/index.html]
- Python cookbook, chapter 12, Concurrency
- (Python Cookbook by David Ascher, Alex Martelli) [https://www.safaribooksonline.com/library/view/python-cookbook/0596001673/ch06.html] Chapter 6. Threads, Processes, and Synchronization
- (И еще раз о GIL в Python)[https://habr.com/post/238703/] - обзор производительности разных решений для параллельных вычислений
- (Учимся писать многопоточно)[https://habr.com/post/149420/]
- (Синхронизация потоков в Python)[http://www.quizful.net/post/thread-synchronization-in-python] перевод (Thread Synchronization Mechanisms in Python) [http://effbot.org/zone/thread-synchronization.htm]

Зачем нужна concurrency

- Эффективнее используем многоядерную архитектуру.
- Разделяем логику программы на полностью или частично асинхронные секции (пингуем несколько серверов одновременно).

Синхронизация доступа к ресурсам

Если можно разбить задачу на независимые подзадачи БЕЗ синхронизации - сделайте это.

Проблема producer-consumer

Один (thread, process) producer производит числа от 0 до 9. Другой такой же объект - consumer - "потребляет" эти числа.

Если нет синхронизации, то могут возникнуть проблемы.

```
import threading

x = 0

def producer():
    global x
    for i in range(5):
        x = i
        print('producer', x)

def consumer():
    global x
    for i in range(5):
        print('consumer', x)


# init threads
t1 = threading.Thread(target=producer, args=())
t2 = threading.Thread(target=consumer, args=())

# start threads
t1.start()
t2.start()

# join threads to the main thread
t1.join()
t2.join()
print('Main', x)
```

Продьюсер может очень быстро производить числа, а консьюмер тупит.

```
producer 0
producer 1
producer 2
producer 3
producer 4
consumer 4
consumer 4
consumer 4
consumer 4
consumer 4
Main 4
```

Или наоборот, продьюсер еще не успел произвести число (потому что в него добавили в начало sleep(1), например), а консьюмер уже его хочет обработать.

```
consumer 0
consumer 0
consumer 0
consumer 0
consumer 0
producer 0
producer 1
producer 2
producer 3
producer 4
Main 4
```

Атомарность операций

Пусть id вычисляет какой-то номер нашего объекта, например, денежной купюры, карты или номер паспорта. Очевидно, что они должны быть уникальными и идти один за другим.

Напишем для этого код:

```
id = 0

def get_id():
    global id
    ... do something with id ...
    id += 1
```

Если запустить в один тред, то все работает хорошо. Если запустить в несколько тредов, то получаем, например, дублирование номеров. Почему?

Операция `id += 1` - НЕ **атомарная**.

Атомарной называют операцию, которая выполняется за 1 шаг без возможности ее прерывания.

`id += 1` состоит из 3 операций: чтения id, вычисления числа на 1 больше, чем прочитанное и запись результата в id. В любой момент времени между операциями может произойти переключение на другой поток.

Атомарные операции:

- чтение или изменение одного атрибута объекта
- чтение или изменение одной глобальной переменной
- выборка элемента из списка
- модификация списка "на месте" (т.е. с помощью метода append)
- выборка элемента из словаря
- модификация словаря "на месте" (т.е. добавление элемента, или вызов метода clear)

Заметьте, или чтение, или изменение атрибута - атомарны, но чтение и последующее изменение атрибута - НЕ атомарная операция.

Критическая секция

участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком (процессом) исполнения. При нахождении в критической секции двух (или более) процессов возникает состояние «гонки» («состязания»).

Атомарность операции не гарантирует, что она не является критической секцией. Очистка словаря - атомарная операция. Но если в это время этот же словарь кто-то перебирает в цикле, то очевидно, что и очистка, и перебор общего словаря - это критические секции.

Для корректной работы критических секций программы используют разные механизмы синхронизации.

Потоки thread

Напишем программу в 2 потока, каждый из которых *по очереди* печатает свой номер (0 или 1) и далее числа от 0 до 9.

Синхронизацию сделаем на Event.

```
import threading

def writer(x, event_for_wait, event_for_set):
    for i in range(10):
        event_for_wait.wait() # wait for event
        event_for_wait.clear() # clean event for future
        print(x, i)
        event_for_set.set() # set event for neighbor thread

# init events
e1 = threading.Event()
e2 = threading.Event()

# init threads
t1 = threading.Thread(target=writer, args=(0, e1, e2))
t2 = threading.Thread(target=writer, args=(1, e2, e1))

# start threads
t1.start()
t2.start()

e1.set() # initiate the first event

# join threads to the main thread
t1.join()
t2.join()
```

Методы класса Thread

Метод	Описание
start()	запускает thread; вызывается 1 раз, перед run()
run()	то, что должен делать thread. Thread.run вызывает метод, переданный в аргументах конструктора с переданными там же аргументами.
join(*timeout*=None)	Ждет, пока этот тред закончится или закончится таймаут. Возвращает всегда None. Вызывайте is_alive(), чтобы узнать жив тред или нет. RuntimeError при попытке join текущий тред (дедлок).
threading.enumerate()	список всех живых тредов
name	имя треда, используется для идентификации, устанавливается в конструкторе. Может совпадать у разных тредов.
iden	идентификатор треда, целое число != 0, может быть повторно использован для нового треда после окончания старого. get_ident()
daemon	для треда-демона нужно установить в True до вызова функции start()

Синхронизация на Event

Объект Event содержит внутренний флаг (изначально false), который устанавливается в true методом **set()**, и сбрасывается методом **clear()**. Метод **wait(*timeout*=None)** блокирует, пока флаг не станет true или не истечет timeout (дробное число секунд).

Метод wait вернет True, если флаг был/стал true. В противном случае (истек таймаут) он вернет False. До 3.1 всегда возвращалось None.

Код легко масштабируется. Добавим еще один thread:

```
import threading

def writer(x, event_for_wait, event_for_set):
    for i in range(10):
        event_for_wait.wait() # wait for event
        event_for_wait.clear() # clean event for future
        print(x, i)
        event_for_set.set() # set event for neighbor thread

# init events
e1 = threading.Event()
e2 = threading.Event()
e3 = threading.Event()  # ++++

# init threads
t1 = threading.Thread(target=writer, args=(0, e1, e2))
t2 = threading.Thread(target=writer, args=(1, e2, e3))  # модифицирована
t2 = threading.Thread(target=writer, args=(3, e3, e1))  # ++++

# start threads
t1.start()
t2.start()
t3.start()              # +++++

e1.set() # initiate the first event

# join threads to the main thread
t1.join()
t2.join()
t3.join()
```

Global Interpreter Lock (GIL)

Концепция Global Interpreter Lock - в каждый момент времени только 1 поток (thread) может исполняться 1 процессором.

- Плюс: разные потоки могут легко использовать одни и те же переменные. Исполняемый поток получает доступ ко всему окружению. Получаем thread safety (потокобезопасность).
- Минус: накладные расходы на переключение потоков.

Напишем миллион строк в файл в одном потоке и измерим время выполнения программы. Получим 0.35 сек.

```
with open('test1.txt', 'w') as fout:
    for i in range(1000000):
        print(1, file = fout)
```

Теперь перепишем код в 2 потока по полмиллиона записей в каждом (без синхронизации потоков):

```
from threading import Thread

def writer(filename, n):
    with open(filename, 'w') as fout:
        for i in range(n):
            print(1, file=fout)

t1 = Thread(target=writer, args=('test2.txt', 500000,))
t2 = Thread(target=writer, args=('test3.txt', 500000,))

t1.start()
t2.start()
t1.join()
t2.join()
```

По сути работы столько же, сколько у предыдущей программы. И ожидаем, что работать она будет примерно столько же. А по факту работает она от 0.7 до 7 секунд. Почему?

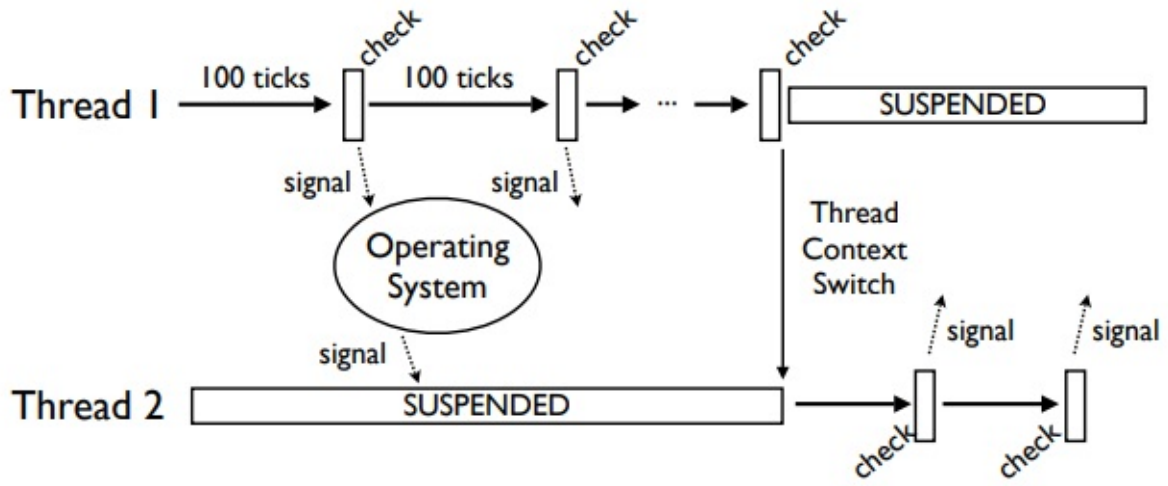
Старый GIL (до 3.1)

В старом GIL обычный цикл изменения одной переменной миллион раз давал следующие результаты на Dual-Core 2Ghz Macbook, OS-X 10.5.6

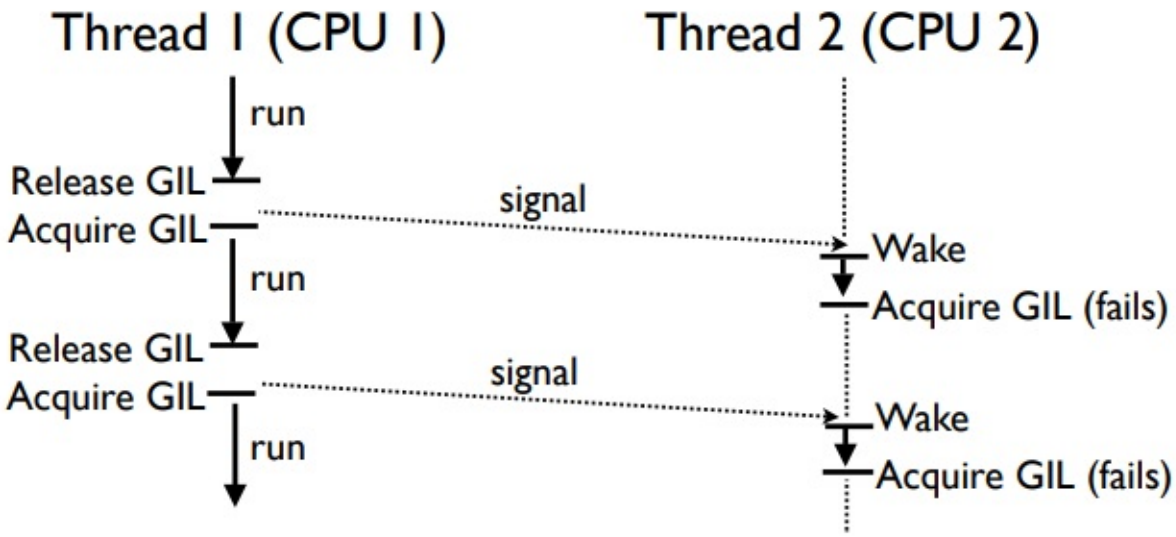
Вариант	Время
Последовательный запуск	24.6 sec
2-thread запуск	45.5 sec
2-thread запуск, отключен 1 CPU core	38.0 sec

В чем дело?

Старый GIL основывался на тиках интерпретатора и повторяющихся сигналах на условной переменной (cond. var)



В случае 2 core должны тоже выполняться по очереди.



Второй thread может прождать 100 сек в безуспешных попытках получить GIL, пока одна из них не закончися успехом.

Посмотрим, как меняется время при Failed GIL Acquire при выключении 1 core:

Новый GIL

Все так же основывается на сигналах и условных переменных, но считает не тики, а секунды, которые устанавливаются в `sys.setcheckinterval()`.

Решение о переключении потоков связано с глобальной переменной

```
static volatile int gil_drop_request = 0;    /* Python/ceval.c */
```

Thread работает в интерпретаторе до тех пор, пока значение этой переменной не станет 1. В этот момент тред обязан отпустить GIL.

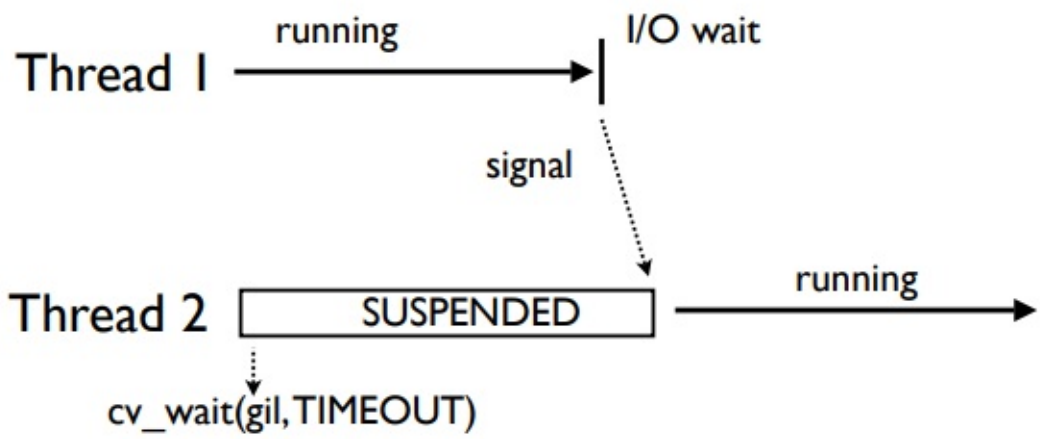
Пока 1 тред, он работает бесконечно (не прерываясь), не отпуская GIL и не посылая сигналов.

Когда появляется второй тред, он приостановлен (suspended), потому что у него нет GIL. Каким то образом должен получить его от первого треда.

Второй тред выполняет `cv_wait` на GIL с ограничением по времени.

Тред 2 будет ждать, когда тред 1 добровольно освободит GIL (например, во время I/O или когда соберется спать).

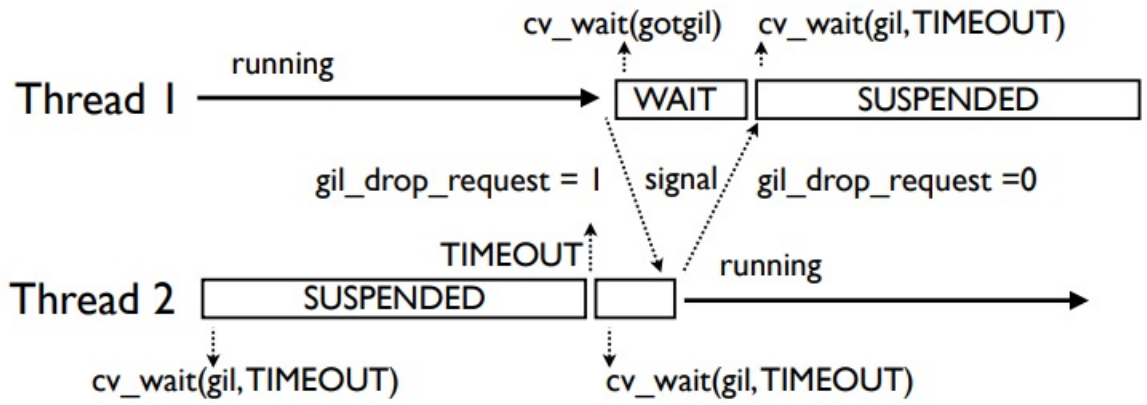
Если GIL отдается добровольно, то треду 2 будет послан сигнал от треда 1, когда тред 1 заснет. Тогда начнет выполняться тред 2.



Если же до окончания таймаута в cv_wait тред 2 не получил GIL, то он посылает gil_drop_request, выставляя переменную `gil_drop_request=1`. После этого он опять ждет в cv_wait.

Тред 1 вынужден отдать GIL: он заканчивает выполнение текущей инструкции, отпускает GIL и посылает сигнал, что GIL свободен. Далее он начинает ждать сигнал (что GIL взят). Тред 2 захватывает GIL и посылает сигнал, что GIL взят. Т.е. "битва за GIL" исключена.

Далее повторяется ситуация, но теперь тред 2 выполняется, а тред 1 - подвешен.



По умолчанию таймаут для переключения треда 5 мс. Для сравнения - по умолчанию таймаут для переключения контекста в большинстве систем 10 мс. Настраивается в `sys.setswitchinterval()` .

Аналогично система работает для многих тредов. По окончанию таймаута тред может установить gil_drop_request=1, только если во время таймаута не было переключения тредов (добровольного или принудительного). Если у нас много соревнующихся тредов, только один из них может установить gil_drop_request за время таймаута.

Тред 3 и тред 4 после первого таймаута не инициируют установку gil_drop_request=1, потому что пока был их таймаут, тред 2 получил GIL (т.е. был thread switch).

После второго таймаута один из них (кто быстрее) сможет инициировать выставление gil_drop_request.

Не факт, что именно тот тред, который выставил gil_drop_request, получит GIL. Это больше зависит от ОС. На рисунке тред 2 выполнил gil_drop_request, а ОС решила, что выполняться будет тред 3.

Теперь по производительности многотредовая программа не намного хуже, чем программа в 1 тред. (До 100 тредов измеряли и получали приемлемые результаты).

Новый GIL позволяет треду выполняться 5 мс, не зависимо от других тредов или приоритетов I/O. Это значит, что треды, требующие ЦПУ могут на некоторое время блокировать треды, требующие I/O.

Таким образом мы исключили избыточное переключение контестов. Если вас не устраивает время ответа в 5 мс, его можно настроить, но...

Обратите внимание

- долгие вычисления и C/C++ расширения могут блокировать переключение тредов;
- переключение тредов не preemptive (кооперативное, а не вытесняющее);
- таким образом, если операция в C extension занимает 5 секунд, вы будете должны подождать это время, пока GIL не отпустят (в старом GIL было такое же поведение).

Вывод: **Не стоит использовать многотредовость для распараллеливания вычислений.** Многотредовость - для обработки событий.

Назад, в историю:

- модуль **thread** - устарел, переименован в **_thread** (для обратной совместимости)
- модуль **threading** - пользуемся им.

Вариант создания нити - наследуемся от Thread

Выше пример не требовал создания новых классов, новый тред создавался при вызовые конструктора класса Thread, которому передавалась функция и ее аргументы.

Можно пойти другим путем - написать класс-наследник от Thread.

Методы класса	Метод Описание
run():	Это функция точки входа для любого потока.
start():	Способ start() запускает поток, когда вызывается метод run.
join([time]):	Метод join() позволяет программе ожидать оканчиваются.
isAlive():	Метод isAlive() проверяет активную нить.
getName():	Метод getName() возвращает имя потока.
setName():	Метод setName() обновляет имя потока.

```
# Многопоточность в Python: пример кода для отображения текущей даты в потоке.
#1. Определить подкласс, используя класс thread.
#2. Создать экземпляр подкласса и вызвет поток.

import threading
import datetime

class myThread (threading.Thread):
    def __init__(self, name, counter):
        threading.Thread.__init__(self)
        self.threadID = counter
        self.name = name
        self.counter = counter
    def run(self):
        print "Запуск " + self.name
        print_date(self.name, self.counter)
        print "Выход " + self.name

def print_date(threadName, counter):
    datefields = []
    today = datetime.date.today()
    datefields.append(today)
    print "%s[%d]: %s" % ( threadName, counter, datefields[0] )

# Создание новой нити
thread1 = myThread("Нить", 1)
thread2 = myThread("Нить", 2)

# Запуск новой нити
thread1.start()
thread2.start()

thread1.join()
thread2.join()
print "Выход из программы!!!"
```

Синхронизация тредов

Синхронизация тредов через Lock

У объекта Lock 2 метода:

- **acquire(blocking=True, timeout=-1)** - ждать, когда лок освободят, не более timeout секунд;
- **release()** - освобождает лок.

Для каждого разделяемого ресурса пишем свой собственый лок

Для избежания взаимных блокировок (deadlock) важно освобождать лок и в случае ошибок. Т.е. пишем его в **try-finally** блоке:

```
lock.acquire()
try:
    ... доступ к разделяемому ресурсу
finally:
    lock.release() # освободить блокировку, что бы ни произошло
```

Начиная с питона 2.5 есть возможность писать гарантированно освобождаемый лок (Lock, RLock, Condition, Semaphore, BoundedSemaphore) с конструкцией **with**. То же самое:

```
with lock:
    ... доступ к разделяемому ресурсу
```

Если хочется избежать блокировки потока методом acquire, если лок уже кем-то захвачен, то нужно вызывать его с аргументом blocking=False, тогда вызов этого метода не блокирует и сразу же возвращает False, если блокировка кем-то захвачена:

```
if not lock.acquire(False):
    ... не удалось заблокировать ресурс
else:
    try:
        ... доступ к разделяемому ресурсу
    finally:
        lock.release()
```

Этот код **НЕ** гарантирует, что мы быстро получим лок в acquire:

```
if not lock.locked():
    # другой поток может начать выполняться перед тем как мы перейдём к следующему опе
ратору
    lock.acquire() # всё равно может заблокировать выполнение
```

Проблемы

Обычной блокировке (объекту типа Lock) всё равно, кто её захватил; если блокировка захвачена, любой поток при попытке её захватить **будет заблокирован, даже если этот поток уже владеет этой блокировкой** в данный момент.

Все работает, даже если эти функции (допустим, они обрабатывают две независимые части объекта) выполняются двумя разными потоками:

```
lock = threading.Lock()
def get_first_part():
    lock.acquire()
    try:
        ... получить данные первой части разделяемого объекта
    finally:
        lock.release()
    return data

def get_second_part():
    lock.acquire()
    try:
        ... получить данные второй части разделяемого объекта
    finally:
        lock.release()
    return data
```

НО! Если хотим добавить функцию, которая обрабатывает сразу весь объект. Этот код работать не будет:

```
def get_both_parts():
    first = get_first_part()
    # тут может вклиниться другой поток, изменить объект и он станет неконсистентным
    second = get_second_part()
    return first, second
```

Нужно локать весь объект целиком! Но этот код тоже не работает:

```
def get_both_parts():
    lock.acquire()                  # держим блокировку
    try:
        first = get_first_part()    # тут не можем получить блокировку, потому что ее
держит тот же поток!
        second = get_second_part()
    finally:
        lock.release()
    return first, second
```

Можно сделать отдельный лок на блокировку объекта целиком:

```
lock = threading.Lock()
lock_whole = threading.Lock()

def get_both_parts():
    with lock_whole:
        first = get_first_part()
        second = get_second_part()
        return first, second

def get_first_part():
    lock.acquire()
    try:
        ... получить данные первой части разделяемого объекта
    finally:
        lock.release()
    return data

def get_second_part():
    lock.acquire()
    try:
        ... получить данные второй части разделяемого объекта
    finally:
        lock.release()
    return data
```

Но лучше использовать лок, который позволяет повторно себя брать тому же потоку, так называемый reenterable lock (**RLock**).

RLock

```
lock = threading.Lock()
lock.acquire()
lock.acquire() # вызов заблокирует выполнение

lock = threading.RLock()
lock.acquire()
lock.acquire() # вызов не заблокирует выполнение
```

Если в предыдущем примере наш лок будет r-локом, то функции get_first_part и get_second_part оставляем без изменений, но переписываем get_both_parts

```
lock = RLock()
def get_first_part():... так же ....
def get_second_part():... так же ....
def get_both_parts():
    lock.acquire()                      # берем лок, глубина вызова +1
    try:
        first = get_first_part()        # повторный лок, глубина вызова +1; освобожден
ие, глубина вызова -1
        second = get_second_part()      # повторный лок, глубина вызова +1; освобожден
ие, глубина вызова -1
    finally:
        lock.release()                  # освобождение, глубина вызова -1, лок освобож
ден полностью!
    return first, second
```

Еще механизмы синхронизации

Аналогичны синхронизации потоков. Давайте вторую половину рассмотрим там.

Данные, специфичные для одного треда

```
mydata = threading.local()
mydata.x = 1
```

Запуск сторонней программы - модуль subprocess

Модуль создает отдельные процессы. Сделан для удобного запуска сторонних команд.

Запускаем функцией **(subprocess.run())** **[https://docs.python.org/3/library/subprocess.html]** (начиная с версии 3.5). Если нужно что-то специфическое, используем непосредственно внутренний интерфейс **Popen**.

В старых версиях были функции **call**, **check**, **check_output**.

Не забываем **import subprocess**

```
subprocess.run(args, *, stdin=None, input=None, stdout=None, stderr=None, shell=False,
 cwd=None, timeout=None, check=False, encoding=None, errors=None)
```

Запускает то, что передано в аргументах, ждет когда исполнится и возвращает экземпляр CompletedProcess.

```
>>> subprocess.run(["ls", "-l"])  # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], stdout=subprocess.PIPE)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n')
```

При shell=True можно не только задать одной строкой `'ls -l /dev/null'`, но и использовать переменные `'cd $HOME'`

Законченный код:

```
try:
    completed = subprocess.run(
        'echo to stdout; echo to stderr 1>&2; exit 1',
        check=True,
        shell=True,
        stdout=subprocess.PIPE,
    )
except subprocess.CalledProcessError as err:
    print('ERROR:', err)
else:
    print('returncode:', completed.returncode)
    print('Have {} bytes in stdout: {!r}'.format(
        len(completed.stdout),
        completed.stdout.decode('utf-8'))
    )
# OUTPUT:
# to stderr
# ERROR: Command 'echo to stdout; echo to stderr 1>&2; exit 1'
# returned non-zero exit status 1
```

отключим check=True:

```
import subprocess

try:
    completed = subprocess.run(
        'echo to stdout; echo to stderr 1>&2; exit 1',
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
    )
except subprocess.CalledProcessError as err:
    print('ERROR:', err)
else:
    print('returncode:', completed.returncode)
    print('Have {} bytes in stdout: {!r}'.format(
        len(completed.stdout),
        completed.stdout.decode('utf-8'))
    )
    print('Have {} bytes in stderr: {!r}'.format(
        len(completed.stderr),
        completed.stderr.decode('utf-8'))
    )
# OUTPUT:
# returncode: 1
# Have 10 bytes in stdout: 'to stdout\n'
# Have 10 bytes in stderr: 'to stderr\n'
```

run() с check=True эквивалентно check_call()

Работа непосредственно с PIPE

Можно использовать непосредственно Popen, чтобы проконтролировать как выполняется команда и как обрабатываются ее input и output. Например, передавая различные аргументы в stdin, stdout и stderr можно имитировать изменения в os.popen().

```
import subprocess

print('read:')
proc = subprocess.Popen(
    ['echo', '"to stdout"'],
    stdout=subprocess.PIPE,
)
stdout_value = proc.communicate()[0].decode('utf-8')
print('stdout:', repr(stdout_value))

# OUTPUT:
# read:
# stdout: '"to stdout"\n'
```

Подадим данные на stdin.

Для одноразовой посылки на stdin используйте communicate(). Аналогично использованию os.popen() с модой 'w'.

```
print('write:')
proc = subprocess.Popen(
    ['cat', '-'],
    stdin=subprocess.PIPE,
)
proc.communicate('stdin: to stdin\n'.encode('utf-8'))

# OUTPUT:
# write:
# stdin: to stdin
```

Заменяем shell pipe line

Если вам хочется получить (отфильтровать, подменить) передаваемые данные: Заменяем shell pipe line `dmesg | grep hda` :

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
output = p2.communicate()[0]
```

Если передать `stdout=subprocess.DEVNULL`, то на `completed.stdout` попадет None.

Сразу и stdin, и stdout:

```
import subprocess

print('popen2:')

proc = subprocess.Popen(
    ['cat', '-'],
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
)
msg = 'through stdin to stdout'.encode('utf-8')
stdout_value = proc.communicate(msg)[0].decode('utf-8')
print('pass through:', repr(stdout_value))

# OUTPUT:
# popen2:
# pass through: 'through stdin to stdout'
```

Перехватываем stderr

```
import subprocess

print('popen3:')
proc = subprocess.Popen(
    'cat -; echo "to stderr" 1>&2',
    shell=True,
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE,
)
msg = 'through stdin to stdout'.encode('utf-8')
stdout_value, stderr_value = proc.communicate(msg)
print('pass through:', repr(stdout_value.decode('utf-8')))
print('stderr      :', repr(stderr_value.decode('utf-8')))

# OUTPUT:
# popen3:
# pass through: 'through stdin to stdout'
# stderr      : 'to stderr\n'
```

Перенаправляем stderr на стандартный stdout

`stderr = subprocess.STDOUT` вместо PIPE:

```
import subprocess

print('popen4:')
proc = subprocess.Popen(
    'cat -; echo "to stderr" 1>&2',
    shell=True,
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.STDOUT,
)
msg = 'through stdin to stdout\n'.encode('utf-8')
stdout_value, stderr_value = proc.communicate(msg)
print('combined output:', repr(stdout_value.decode('utf-8')))
print('stderr value   :', repr(stderr_value))

# OUTPUT:
# popen4:
# combined output: 'through stdin to stdout\nto stderr\n'
# stderr value   : None
```

Цепочка вызовов

Одна команда перенаправляет выход на вход другой. Для иллюстрации напишем код, выполняющий `cat index.rst | grep ".. literalinclude" | cut -f 3 -d:` (заметим, его можно выполнить непосредственно subprocess.run, указав shell=True):

```
import subprocess

cat = subprocess.Popen(
    ['cat', 'index.rst'],
    stdout=subprocess.PIPE,
)

grep = subprocess.Popen(
    ['grep', '.. literalinclude::'],
    stdin=cat.stdout,
    stdout=subprocess.PIPE,
)

cut = subprocess.Popen(
    ['cut', '-f', '3', '-d:'],
    stdin=grep.stdout,
    stdout=subprocess.PIPE,
)

end_of_pipe = cut.stdout

print('Included files:')
for line in end_of_pipe:
    print(line.decode('utf-8').strip())
```

TODO

Посмотреть, нужно ли добавлять примеры из:

https://pymotw.com/3/subprocess/

Interacting with Another Command и далее

Посмотреть кукбуки

Модуль muliprocessing - работа с процессами

В первом приближении программа на похожа на программу с тредами:

```
import threading


def worker(num):
    """thread worker function"""
    print('Worker: %s' % num)


threads = []
for i in range(5):
    t = threading.Thread(target=worker, args=(i,))
    threads.append(t)
    t.start()
```

То же самое для процессов, а не тредов:

```
import multiprocessing


def worker(num):
    """thread worker function"""
    print('Worker:', num)


if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker, args=(i,))
        jobs.append(p)
        p.start()
```

Методы класса Process

Метод	Описание
start()	запускает процесс; вызывается 1 раз, перед run()
run()	то, что должен делать процесс. Process.run вызывает метод, переданный в аргументах конструктора с переданными там же аргументами.
join(*timeout*=None)	Ждет, пока этот процесс закончится или закончится таймаут. Возвращает всегда None. Вызывайте is_alive(), чтобы узнать жив процесс или нет. RuntimeError при попытке join текущий процесс (дедлок).
name	имя процесса, используется для идентификации, устанавливается в конструкторе. Может совпадать у разных процессов.
pid	pid, до реального создания вернет None
daemon	для процесса-демона нужно установить в True до вызова функции start(). **Демон не может иметь детей**
exitcode	код завершения ребенка. None, если он еще не завершился. -N (отрицательное число), если завершился по сигналу N.
terminate()	Завершает процесс. В UNIX посылает SIGTERM, в Windows вызывается TerminateProcess()

Note that the start(), join(), is_alive(), terminate() and exitcode methods should only be called by the process that created the process object.

Подробнее о старте процесса. В модуле реализовано 3 метода старта нового процесса:

- **spawn** The parent process starts a fresh python interpreter process. The child process will only inherit those resources necessary to run the process objects run() method. In particular, unnecessary file descriptors and handles from the parent process will not be inherited. Starting a process using this method is rather slow compared to using fork or forkserver.

Available on Unix and Windows. The default on Windows.

- **fork** The parent process uses os.fork() to fork the Python interpreter. The child process, when it begins, is effectively identical to the parent process. All resources of the parent are inherited by the child process. Note that safely forking a multithreaded process is problematic.

Available on Unix only. The default on Unix.

- **forkserver** When the program starts and selects the forkserver start method, a server process is started. From then on, whenever a new process is needed, the parent

process connects to the server and requests that it fork a new process. The fork server process is single threaded so it is safe for it to use os.fork(). No unnecessary resources are inherited.

Available on Unix platforms which support passing file descriptors over Unix pipes.

Вывод: если ваша программа работала на UNIX, до старта дочерних процессов читая конфиг-файлы и открывая файлы и сетевые соединения, то на Windows этой информации нет - "пустой" новый процесс. Передавайте нужную информацию в конструкторе (dict конфига), shared memory и тп.

exitcode

Выход из дочернего процесса	exitcode этого процесса
`return n`	0
`return`	0
`sys.exit(n)`	число n
исключение	1
окончен послыкой сигнала N	-1

```
# multiprocessing_exitcode.py
import multiprocessing
import sys
import time

def exit_error():
    sys.exit(1)

def exit_ok():
    return

def return_value():
    return 1

def raises():
    raise RuntimeError('There was an error!')

def terminated():
    time.sleep(3)

if __name__ == '__main__':
    jobs = []
    funcs = [
        exit_error,
        exit_ok,
        return_value,
        raises,
        terminated,
    ]
    for f in funcs:
        print('Starting process for', f.__name__)
        j = multiprocessing.Process(target=f, name=f.__name__)
        jobs.append(j)
        j.start()

    jobs[-1].terminate()

    for j in jobs:
        j.join()
        print('{:>15}.exitcode = {}'.format(j.name, j.exitcode))
```

Получим:

```
$ python3 multiprocessing_exitcode.py

Starting process for exit_error
Starting process for exit_ok
Starting process for return_value
Starting process for raises
Starting process for terminated
Process raises:
Traceback (most recent call last):
  File ".../lib/python3.6/multiprocessing/process.py", line 258,
in _bootstrap
    self.run()
  File ".../lib/python3.6/multiprocessing/process.py", line 93,
in run
    self._target(*self._args, **self._kwargs)
  File "multiprocessing_exitcode.py", line 28, in raises
    raise RuntimeError('There was an error!')
RuntimeError: There was an error!
     exit_error.exitcode = 1
        exit_ok.exitcode = 0
   return_value.exitcode = 0
         raises.exitcode = 1
     terminated.exitcode = -15
```

Логирование

Подробнее смотри рецепты логирования.

Самый простой случай - все сообщения печатаем на stderr. Для этого в модуле muliprocessing есть уже сконфигурированный логгер **log_to_stderr**.

По умолчанию этот логер установлен в уровень NOTSET, когда нет никаких сообщений. Поэтому единственное, что вам нужно - установить нужный уровень:

```
multiprocessing_log_to_stderr.py
import multiprocessing
import logging
import sys


def worker():
    print('Doing some work')
    sys.stdout.flush()


if __name__ == '__main__':
    multiprocessing.log_to_stderr(logging.DEBUG)
    p = multiprocessing.Process(target=worker)
    p.start()
    p.join()
```

получим:

```
$ python3 multiprocessing_log_to_stderr.py

[INFO/Process-1] child process calling self.run()
Doing some work
[INFO/Process-1] process shutting down
[DEBUG/Process-1] running all "atexit" finalizers with priority
>= 0
[DEBUG/Process-1] running the remaining "atexit" finalizers
[INFO/Process-1] process exiting with exitcode 0
[INFO/MainProcess] process shutting down
[DEBUG/MainProcess] running all "atexit" finalizers with
priority >= 0
[DEBUG/MainProcess] running the remaining "atexit" finalizers
```

Если нужно менять уровень логирования, то получим сначала ссылку на логер **get_logger()**:

```
import multiprocessing
import logging
import sys


def worker():
    print('Doing some work')
    sys.stdout.flush()


if __name__ == '__main__':
    multiprocessing.log_to_stderr()
    logger = multiprocessing.get_logger()
    logger.setLevel(logging.INFO)
    p = multiprocessing.Process(target=worker)
    p.start()
    p.join()
```

Наследуемся от класса Process

Как и в случает thread, можно не только передавать функцию в конструктор, и наследоваться от класса muliprocessing.Process, переопределяя функцию run().

```
import multiprocessing

class Worker(multiprocessing.Process):

    def run(self):
        print('In {}'.format(self.name))
        return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = Worker()
        jobs.append(p)
        p.start()
    for j in jobs:
        j.join()
```

Передача сообщений между процессами - Queue

Типичный сценарий использования процессов - разделить всю работу между несколькими работниками. Потом нужно будет собрать результаты этой работы.

Простейший способ обмениваться информацией между процессами - очередь сообщений **Queue**, а объекты сериализуем модулем **pickle**.

```
import multiprocessing

class MyFancyClass:

    def __init__(self, name):
        self.name = name

    def do_something(self):
        proc_name = multiprocessing.current_process().name
        print('Doing something fancy in {} for {}!'.format(
            proc_name, self.name))

def worker(q):
    obj = q.get()
    obj.do_something()

if __name__ == '__main__':
    queue = multiprocessing.Queue()

    p = multiprocessing.Process(target=worker, args=(queue,))
    p.start()

    queue.put(MyFancyClass('Fancy Dan'))

    # Wait for the worker to finish
    queue.close()
    queue.join_thread()
    p.join()
```

получим:

```
Doing something fancy in Process-1 for Fancy Dan!
```

Теперь несколько процессов получают данные из **JoinableQueue** и пересылают результаты обратно родительскому процессу. Для остановки работников используется техника 'poison pill': после посылки реальных данных главный процесс посылает специальные "стоп" значения (которые точно не могут быть реальными данными). После получения этого значения, работник прекращает свой цикл обработки данных. Посылается по 1 стоп-сигналу на каждого работника. Родительский процесс использует join() очереди, чтобы дождаться, что все дети закончили посылать ему данные, и далее обрабатывает присланные результаты.

```
import multiprocessing
import time

class Consumer(multiprocessing.Process):
```

```
    def __init__(self, task_queue, result_queue):
        multiprocessing.Process.__init__(self)
        self.task_queue = task_queue
        self.result_queue = result_queue

    def run(self):
        proc_name = self.name
        while True:
            next_task = self.task_queue.get()
            if next_task is None:
                # Poison pill means shutdown
                print('{}: Exiting'.format(proc_name))
                self.task_queue.task_done()
                break
            print('{}: {}'.format(proc_name, next_task))
            answer = next_task()
            self.task_queue.task_done()
            self.result_queue.put(answer)


class Task:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __call__(self):
        time.sleep(0.1)  # pretend to take time to do the work
        return '{self.a} * {self.b} = {product}'.format(
            self=self, product=self.a * self.b)

    def __str__(self):
        return '{self.a} * {self.b}'.format(self=self)


if __name__ == '__main__':
    # Establish communication queues
    tasks = multiprocessing.JoinableQueue()
    results = multiprocessing.Queue()

    # Start consumers
    num_consumers = multiprocessing.cpu_count() * 2
    print('Creating {} consumers'.format(num_consumers))
    consumers = [
        Consumer(tasks, results)
        for i in range(num_consumers)
    ]
    for w in consumers:
        w.start()

    # Enqueue jobs
    num_jobs = 10
    for i in range(num_jobs):
```

```
            tasks.put(Task(i, i))

    # Add a poison pill for each consumer
    for i in range(num_consumers):
        tasks.put(None)

    # Wait for all of the tasks to finish
    tasks.join()

    # Start printing results
    while num_jobs:
        result = results.get()
        print('Result:', result)
        num_jobs -= 1
```

получим:

```
Creating 8 consumers
Consumer-1: 0 * 0
Consumer-2: 1 * 1
Consumer-3: 2 * 2
Consumer-4: 3 * 3
Consumer-5: 4 * 4
Consumer-6: 5 * 5
Consumer-7: 6 * 6
Consumer-8: 7 * 7
Consumer-3: 8 * 8
Consumer-7: 9 * 9
Consumer-4: Exiting
Consumer-1: Exiting
Consumer-2: Exiting
Consumer-5: Exiting
Consumer-6: Exiting
Consumer-8: Exiting
Consumer-7: Exiting
Consumer-3: Exiting
Result: 6 * 6 = 36
Result: 2 * 2 = 4
Result: 3 * 3 = 9
Result: 0 * 0 = 0
Result: 1 * 1 = 1
Result: 7 * 7 = 49
Result: 4 * 4 = 16
Result: 5 * 5 = 25
Result: 8 * 8 = 64
Result: 9 * 9 = 81
```

muliprocessing.Event

Аналогичен threading.Event.

Event может быть установлен или нет. Можно ждать event (быть может с ограничением по таймауту).

```
import multiprocessing
import time


def wait_for_event(e):
    """Wait for the event to be set before doing anything"""
    print('wait_for_event: starting')
    e.wait()
    print('wait_for_event: e.is_set()->', e.is_set())


def wait_for_event_timeout(e, t):
    """Wait t seconds and then timeout"""
    print('wait_for_event_timeout: starting')
    e.wait(t)
    print('wait_for_event_timeout: e.is_set()->', e.is_set())


if __name__ == '__main__':
    e = multiprocessing.Event()
    w1 = multiprocessing.Process(
        name='block',
        target=wait_for_event,
        args=(e,),
    )
    w1.start()

    w2 = multiprocessing.Process(
        name='nonblock',
        target=wait_for_event_timeout,
        args=(e, 2),
    )
    w2.start()

    print('main: waiting before calling Event.set()')
    time.sleep(3)
    e.set()
    print('main: event is set')
```

получено:

```
main: waiting before calling Event.set()
wait_for_event: starting
wait_for_event_timeout: starting
wait_for_event_timeout: e.is_set()-> False
main: event is set
wait_for_event: e.is_set()-> True
```

muliprocessing.Lock

Аналогичен threading.Lock

acquire() - запрашивает лок. **release()** - освобождает лок.

Обратите внимание, что правильно обрернуть взятие-отпускание лога в try-finally блок:

```
    lock.acquire()
    try:
        stream.write('Lock acquired directly\n')
    finally:
        lock.release()
```

или воспользуйтесь конструкцией **with**:

```
def worker_with(lock, stream):
    with lock:
        stream.write('Lock acquired via with\n')
```

Полный код:

```
import multiprocessing
import sys


def worker_with(lock, stream):
    with lock:
        stream.write('Lock acquired via with\n')


def worker_no_with(lock, stream):
    lock.acquire()
    try:
        stream.write('Lock acquired directly\n')
    finally:
        lock.release()


lock = multiprocessing.Lock()
w = multiprocessing.Process(
    target=worker_with,
    args=(lock, sys.stdout),
)
nw = multiprocessing.Process(
    target=worker_no_with,
    args=(lock, sys.stdout),
)

w.start()
nw.start()

w.join()
nw.join()
```

получим:

```
Lock acquired via with
Lock acquired directly
```

muliprocessing.Condition

Аналогичен threading.Condition

Процесс s1 отрабатывает stage_1. Процессы s2 отрабатывают stage_2.

Два потока работают на второй стадии в параллель, но только после того, как закончила работу первая стадия.

```
import multiprocessing
import time


def stage_1(cond):
    """perform first stage of work,
    then notify stage_2 to continue
    """
    name = multiprocessing.current_process().name
    print('Starting', name)
    with cond:
        print('{} done and ready for stage 2'.format(name))
        cond.notify_all()


def stage_2(cond):
    """wait for the condition telling us stage_1 is done"""
    name = multiprocessing.current_process().name
    print('Starting', name)
    with cond:
        cond.wait()
        print('{} running'.format(name))


if __name__ == '__main__':
    condition = multiprocessing.Condition()
    s1 = multiprocessing.Process(name='s1',
                                 target=stage_1,
                                 args=(condition,))
    s2_clients = [
        multiprocessing.Process(
            name='stage_2[{}]'.format(i),
            target=stage_2,
            args=(condition,),
        )
        for i in range(1, 3)
    ]

    for c in s2_clients:
        c.start()
        time.sleep(1)
    s1.start()

    s1.join()
    for c in s2_clients:
        c.join()
```

получим:

```
Starting stage_2[1]
Starting stage_2[2]
Starting s1
s1 done and ready for stage 2
stage_2[1] running
stage_2[2] running
```

multiprocessing.Semaphore

Иногда полезно обеспечить одновременный достп к ресурсу многим процессам, но ограничить количество этих процессов.

Например, пул соединений может поддерживать ограниченное количество одновременных соединений, или сетевое приложение может поддерживать фиксированное количество одновременных скачиваний.

В этих случаях удобно использовать семафоры Semaphore. В отличие от Lock, где 1 значение - взведено (1) или нет (0), семафор поддерживает значения от 0 до n.

В этом примере класс ActivePool просто служит удобным способом отслеживания процессов, выполняемых в данный момент. Реальный пул ресурсов, вероятно, выделит соединение или другое значение для нового активного процесса и вернет значение, когда задача будет выполнена. Здесь пул просто используется для хранения имен активных процессов, чтобы показать, что одновременно работают только три.

In this example, the ActivePool class simply serves as a convenient way to track which processes are running at a given moment. A real resource pool would probably allocate a connection or some other value to the newly active process, and reclaim the value when the task is done. Here, the pool is just used to hold the names of the active processes to show that only three are running concurrently.

```
import random
import multiprocessing
import time

class ActivePool:

    def __init__(self):
        super(ActivePool, self).__init__()
        self.mgr = multiprocessing.Manager()
        self.active = self.mgr.list()
        self.lock = multiprocessing.Lock()

    def makeActive(self, name):
        with self.lock:
            self.active.append(name)
```

```
    def makeInactive(self, name):
        with self.lock:
            self.active.remove(name)

    def __str__(self):
        with self.lock:
            return str(self.active)


def worker(s, pool):
    name = multiprocessing.current_process().name
    with s:
        pool.makeActive(name)
        print('Activating {} now running {}'.format(
            name, pool))
        time.sleep(random.random())
        pool.makeInactive(name)


if __name__ == '__main__':
    pool = ActivePool()
    s = multiprocessing.Semaphore(3)
    jobs = [
        multiprocessing.Process(
            target=worker,
            name=str(i),
            args=(s, pool),
        )
        for i in range(10)
    ]

    for j in jobs:
        j.start()

    while True:
        alive = 0
        for j in jobs:
            if j.is_alive():
                alive += 1
                j.join(timeout=0.1)
                print('Now running {}'.format(pool))
        if alive == 0:
            # all done
            break
```

получим:

```
Activating 0 now running ['0', '1', '2']
Activating 1 now running ['0', '1', '2']
Activating 2 now running ['0', '1', '2']
Now running ['0', '1', '2']
Now running ['0', '1', '2']
Now running ['0', '1', '2']
Now running ['0', '1', '2']
Activating 3 now running ['0', '1', '3']
Activating 4 now running ['1', '3', '4']
Activating 6 now running ['1', '4', '6']
Now running ['1', '4', '6']
Now running ['1', '4', '6']
Activating 5 now running ['1', '4', '5']
Now running ['1', '4', '5']
Now running ['1', '4', '5']
Now running ['1', '4', '5']
Activating 8 now running ['4', '5', '8']
Now running ['4', '5', '8']
Now running ['4', '5', '8']
Now running ['4', '5', '8']
Now running ['4', '5', '8']
Now running ['4', '5', '8']
Activating 7 now running ['5', '8', '7']
Now running ['5', '8', '7']
Activating 9 now running ['8', '7', '9']
Now running ['8', '7', '9']
Now running ['8', '9']
Now running ['8', '9']
Now running ['9']
Now running ['9']
Now running ['9']
Now running ['9']
Now running []
```

Managing Shared State

В предыдущем примере список активных процессов поддерживался централизованно в экземпляре ActivePool через специальный тип list() в классе muliprocessing.Manager. Manager координирует доступ к совместо используемым ресурсам между всеми его пользователями.

В примере ниже все процессы будут писать в единый (совместно используемый) словарь.

```
import multiprocessing
import pprint


def worker(d, key, value):
    d[key] = value


if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    d = mgr.dict()
    jobs = [
        multiprocessing.Process(
            target=worker,
            args=(d, i, i * 2),
        )
        for i in range(10)
    ]
    for j in jobs:
        j.start()
    for j in jobs:
        j.join()
    print('Results:', d)
```

получаем:

```
Results: {0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14,
8: 16, 9: 18}
```

Shared Namespace

В добавок к list и dict класс Manager может создавать совместно используемые namespace.

Любое значение имени, добавленное в Namespace, видно во всех клиентах, у которых есть этот экземпляр Namespace.

```
import multiprocessing


def producer(ns, event):
    ns.value = 'This is the value'
    event.set()


def consumer(ns, event):
    try:
        print('Before event: {}'.format(ns.value))
    except Exception as err:
        print('Before event, error:', str(err))
    event.wait()
    print('After event:', ns.value)


if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    namespace = mgr.Namespace()
    event = multiprocessing.Event()
    p = multiprocessing.Process(
        target=producer,
        args=(namespace, event),
    )
    c = multiprocessing.Process(
        target=consumer,
        args=(namespace, event),
    )

    c.start()
    p.start()

    c.join()
    p.join()
```

получаем:

```
Before event, error: 'Namespace' object has no attribute 'value'
After event: This is the value
```

Модификация изменяемых переменных НЕ поддерживается автоматически

В примере сохраняем в namespace ссылку на list (изменяемый тип). В одном процессе изменяем по ссылке его содержимое, в другом процессе ничего не знают об изменениях.

```
import multiprocessing


def producer(ns, event):
    # DOES NOT UPDATE GLOBAL VALUE!
    ns.my_list.append('This is the value')
    event.set()


def consumer(ns, event):
    print('Before event:', ns.my_list)
    event.wait()
    print('After event :', ns.my_list)


if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    namespace = mgr.Namespace()
    namespace.my_list = []

    event = multiprocessing.Event()
    p = multiprocessing.Process(
        target=producer,
        args=(namespace, event),
    )
    c = multiprocessing.Process(
        target=consumer,
        args=(namespace, event),
    )

    c.start()
    p.start()

    c.join()
    p.join()
```

получилось: (да ничего не получилось! консьюмер так и не узнал об изменениях в продьюсере!)

```
Before event: []
After event : []
```

Для обновления содержимого изменяемой переменной свяжите заново объект namespace с этой переменной.

(To update the list, attach it to the namespace object again.)

А для списка лучше используйте специальный тип из Manager.list(), а не из Manager.namespace()

Пул процессов

Если задача может быть разбита на независимые подзадачи, которые можно распределить между исполнителями, то для этого использовать класс Pool. return values работ собираются в список, который будет возвращен как результат работы всего пула.

Аргументы конструктора Pool включает количество процессов и функцию, которая будет выполняться, когда стартует дочерний процесс (1 функция на 1 ребенка).

Результат метода map() эквивалентен встроенной функции map(), за исключением того, что отдельные задачи выполняются параллельно. Так как пул обрабатывает его входные значения параллельно, close() и join() используются для того, чтобы синхронизировать родительский и дочерние процессы для необходимой зачистки (cleanup).

```
import multiprocessing


def do_calculation(data):
    return data * 2


def start_process():
    print('Starting', multiprocessing.current_process().name)


if __name__ == '__main__':
    inputs = list(range(10))
    print('Input   :', inputs)

    builtin_outputs = map(do_calculation, inputs)
    print('Built-in:', builtin_outputs)

    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(
        processes=pool_size,
        initializer=start_process,
    )
    pool_outputs = pool.map(do_calculation, inputs)
    pool.close()  # no more tasks
    pool.join()  # wrap up current tasks

    print('Pool    :', pool_outputs)
```

получим:

```
Input   : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Built-in: <map object at 0x1007b2be0>
Starting ForkPoolWorker-3
Starting ForkPoolWorker-4
Starting ForkPoolWorker-5
Starting ForkPoolWorker-6
Starting ForkPoolWorker-1
Starting ForkPoolWorker-7
Starting ForkPoolWorker-2
Starting ForkPoolWorker-8
Pool    : [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

По умолчанию Pool создает фиксированное количество рабочих процессов и передает им задачи до тех пор, пока задачи не закончатся. Установка параметра `maxtasksperchild` заставит пул рестартовать дочерний процесс после того, как он выполнит указанное число заданий, предотвращая долго выполняющиеся дочерние процессы от расхода системных ресурсов.

By default, Pool creates a fixed number of worker processes and passes jobs to them until there are no more jobs. Setting the maxtasksperchild parameter tells the pool to restart a worker process after it has finished a few tasks, preventing long-running workers from consuming ever more system resources.

Изменим только вызов конструктора Pool:

```
    pool = multiprocessing.Pool(
        processes=pool_size,
        initializer=start_process,
        maxtasksperchild=2,
    )
```

The pool restarts the workers when they have completed their allotted tasks, even if there is no more work. In this output, eight workers are created, even though there are only 10 tasks, and each worker can complete two of them at a time.

```
Input   : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Built-in: <map object at 0x1007b21d0>
Starting ForkPoolWorker-1
Starting ForkPoolWorker-2
Starting ForkPoolWorker-4
Starting ForkPoolWorker-5
Starting ForkPoolWorker-6
Starting ForkPoolWorker-3
Starting ForkPoolWorker-7
Starting ForkPoolWorker-8
Pool    : [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Модуль asyncio

Фреймворк, представляемый asyncio основан на event loop (цикле обработки событий), первоклассном объекте, ответственном за обработку событий ввода-вывода, системных событий и изменения контекста приложения. У этого цикла есть несколько реализаций. Обычно по умолчанию берется подходящая реализация, но вы можете выбрать реализацию сами. Это может быть полезно, например, под Windows, где некторые классы могут дать выигрыш в сетевом I/O.

Приложение взаимодействует с event loop *неявно*, регистрируя код, который будет выполнен и позволяя циклу обработки событий делать необходимые вызовы в коде приложения когда появляются события.

Например, сервер открывает сокеты и регистрирует их, чтобы сказали, когда придет input event. Event loop уведомляет сервер, когда возникает новое входящее соединиение или когда есть данные для чтения.

Ожидается, что приложение снова получит контроль после обработки события. То есть если нет больше данных для чтения, сервер должен вернуть управление циклу обработки.

Правильный возврат управления обеспечивают корутины. О них чуть позже.

future - это структура данных, представляющая результат работы, которая еще не совсем закончена. Event loop может следить за тем, что объект Future заверщается, позволяя одной части приложения ожидать другую для завершения некоторой работы. Кроме future модуль asyncio содержит другие concurrency примитивы, такие как lock и semaphore.

Task - это подкласс Future, который знает как обернуть и управлять исполнением корутин. Task может быть scheduled (запланирована) циклом обработки на выполнение, когда будут доступны необходимые ей ресурсы, и передаст произведенные данные (результат) в другие корутины.

Корутины

Корутины похожи на генераторы за исключением нескольких отличий, основные из которых:

- генераторы возвращают данные
- корутины потребляют данные

Создадим генератор для чисел Фибоначчи:

```
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

Этот генератор можно использовать в цикле for:

```
for i in fib():
    print(i)
```

Такой подход отличается скоростью и отсутствием повышенной нагрузки на память, поскольку значения генерируются "на лету" и не хранятся в списке. Теперь, если мы используем yield, то получим корутину.

Корутины потребляют данные, которые им передают

Пример реализации grep через корутину:

```
def grep(pattern):
    print("Searching for", pattern)
    while True:
        line = (yield)
        if pattern in line:
            print(line)
```

Код `line = (yield)` - место, куда передаются данные из внешнего источника.

Сначала данных нет. Потом они передаются метдом **send()**:

```
search = grep('coroutine')
next(search)                          # нужно, чтобы корутина заработала
# Вывод: Searching for coroutine
search.send("I love you")
search.send("Don't you love me?")
search.send("I love coroutines instead!")
# Вывод: I love coroutines instead!
```

Корутины, как и генераторы, не запускаются сразу. Для запуска нужен метод __next__() и send().

Корутины можно закрыть:

```
search.close()
```

Кооперативная многозадачность с корутинами

Стартовать корутину event loop модуля asyncio может разными способами. Самый простой - вызвать **run_until_complete()**, передавая в него саму корутину.

Заметьте, написание корутины стало синтаксически проще:

```
import asyncio

async def coroutine():
    print('in coroutine')

event_loop = asyncio.get_event_loop()
try:
    print('starting coroutine')
    coro = coroutine()                  # это НЕ приводит к вызову корутины
    print('entering event loop')
    event_loop.run_until_complete(coro)
finally:
    print('closing event loop')
    event_loop.close()
```

получаем:

```
starting coroutine
entering event loop
in coroutine
closing event loop
```

Возвращаем результат из корутины

Из корутины возвращается результат. Код ожидает получения этого результата.

```
import asyncio

async def coroutine():
    print('in coroutine')
    return 'result'

event_loop = asyncio.get_event_loop()
try:
    return_value = event_loop.run_until_complete(
        coroutine()
    )
    print('it returned: {!r}'.format(return_value))
finally:
    event_loop.close()
```

получаем:

```
in coroutine
it returned: 'result'
```

Цепь выполнения корутин

Одна корутина может стартовать другую корутину и ожидать от нее результата. Это позволяет запросто сделать декомпозицию задачи на части, которые можно повторно использовать.

Пример описывает 2 фазы, которые должны выполняться в нужном порядке, но могут выполняться параллельно с другими операциями.

```
import asyncio

async def outer():
    print('in outer')
    print('waiting for result1')
    result1 = await phase1()
    print('waiting for result2')
    result2 = await phase2(result1)
    return (result1, result2)

async def phase1():
    print('in phase1')
    return 'result1'

async def phase2(arg):
    print('in phase2')
    return 'result2 derived from {}'.format(arg)

event_loop = asyncio.get_event_loop()
try:
    return_value = event_loop.run_until_complete(outer())
    print('return value: {!r}'.format(return_value))
finally:
    event_loop.close()
```

получаем:

```
in outer
waiting for result1
in phase1
waiting for result2
in phase2
return value: ('result1', 'result2 derived from result1')
```

Генераторы вместо корутин

Корутины - это ключевой компонент дизайна asyncio. Они дают конструкцию языка для того, чтобы остановить выполнение программы, при этом сохраняя состояние, в котором была программа и входя в это состояние через некоторое время. Что является очень важным для поддержания данного фреймворка.

В питоне 3.5 была введена новая возможность определять такие корутины естественно, используя **async def** и **await** для ожидания.

Ранняя версия питона 3.0 использует генераторы, обернутые в декоратор **asyncio.coroutine()** и **yield** для достижения того же эффекта.

```
# asyncio_generator.py
import asyncio

@asyncio.coroutine
def outer():
    print('in outer')
    print('waiting for result1')
    result1 = yield from phase1()
    print('waiting for result2')
    result2 = yield from phase2(result1)
    return (result1, result2)

@asyncio.coroutine
def phase1():
    print('in phase1')
    return 'result1'

@asyncio.coroutine
def phase2(arg):
    print('in phase2')
    return 'result2 derived from {}'.format(arg)

event_loop = asyncio.get_event_loop()
try:
    return_value = event_loop.run_until_complete(outer())
    print('return value: {!r}'.format(return_value))
finally:
    event_loop.close()
```

получим то же самое, что и в предыдущем примере.

Планирование запуска обычных функций

TODO - написать потом, запуск функций по таймеру (задержка, планирование на определенное время) https://pymotw.com/3/asyncio/scheduling.html

Scheduling a Callback "Soon"

Если время вызова callback не имеет значения, то можно использовать **call_soon()** для планирования вызова следующей итерации цикла. Любые дополнительные позиционные аргумента после функции передаются в callback во время его вызова. Для передачи в callback аргументов по ключу, используйте **partial()** из модуля functools.

Callback-и запускаются в том порядке, в котором были переданы.

```
# asyncio_call_soon.py
import asyncio
import functools


def callback(arg, *, kwarg='default'):
    print('callback invoked with {} and {}'.format(arg, kwarg))


async def main(loop):
    print('registering callbacks')
    loop.call_soon(callback, 1)
    wrapped = functools.partial(callback, kwarg='not default')
    loop.call_soon(wrapped, 2)

    await asyncio.sleep(0.1)


event_loop = asyncio.get_event_loop()
try:
    print('entering event loop')
    event_loop.run_until_complete(main(event_loop))
finally:
    print('closing event loop')
    event_loop.close()
```

получим

```
entering event loop
registering callbacks
callback invoked with 1 and default
callback invoked with 2 and not default
closing event loop
```

Асинхронное получение результатов

future - это структура данных, представляющая результат работы, которая еще не совсем закончена. Event loop может следить за тем, что объект Future заверщается, позволяя одной части приложения ожидать другую для завершения некоторой работы.

Ожидание Future

Future работает как корутина, то есть приемы ожидания корутины так же можно использовать для ожидания, когда future будет помечена как завершенная. В примере future передается в метод **run_until_complete()** цикла обработки.

Состояние Future изменяется в "завершено", когда вызывается **set_result()** и экземпляр Future сохраняет результат, переданный в метод, для дальнейшего поиска.

```
import asyncio

def mark_done(future, result):
    print('setting future result to {!r}'.format(result))
    future.set_result(result)

event_loop = asyncio.get_event_loop()
try:
    all_done = asyncio.Future()

    print('scheduling mark_done')
    event_loop.call_soon(mark_done, all_done, 'the result')

    print('entering event loop')
    result = event_loop.run_until_complete(all_done)
    print('returned result: {!r}'.format(result))
finally:
    print('closing event loop')
    event_loop.close()

print('future result: {!r}'.format(all_done.result()))
```

получаем:

```
scheduling mark_done
entering event loop
setting future result to 'the result'
returned result: 'the result'
closing event loop
future result: 'the result'
```

Future так же может использовать ключевое слово **await**. Вместо

```
    result = event_loop.run_until_complete(all_done)
```

можно писать:

```
    result = await all_done
```

Результат Future возвращается, используя await, таким образом зачастую у нас одинаковый код для обычной корутины и экземпляра Future:

```
# asyncio_future_await.py
import asyncio

def mark_done(future, result):
    print('setting future result to {!r}'.format(result))
    future.set_result(result)

async def main(loop):
    all_done = asyncio.Future()

    print('scheduling mark_done')
    loop.call_soon(mark_done, all_done, 'the result')

    result = await all_done
    print('returned result: {!r}'.format(result))

event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

получим:

```
scheduling mark_done
setting future result to 'the result'
returned result: 'the result'
```

Future Callbacks

В добавок к тому, что Future работает подобно корутине, она может вызывать callbacks во время своего завершения. Callbacks включаются в том же порядке, в каком были зарегистрированы.

callback должен ожидать один аргумент, экземпляр Future. Для передачи дополнительных аргументов в колбеки, так же используйте функцию functools.partial() для написания функции-обертки.

The callback should expect one argument, the Future instance. To pass additional arguments to the callbacks, use functools.partial() to create a wrapper.

```
# asyncio_future_callback.py
import asyncio
import functools

def callback(future, n):
    print('{}: future done: {}'.format(n, future.result()))

async def register_callbacks(all_done):
    print('registering callbacks on future')
    all_done.add_done_callback(functools.partial(callback, n=1))
    all_done.add_done_callback(functools.partial(callback, n=2))


async def main(all_done):
    await register_callbacks(all_done)
    print('setting result of future')
    all_done.set_result('the result')


event_loop = asyncio.get_event_loop()
try:
    all_done = asyncio.Future()
    event_loop.run_until_complete(main(all_done))
finally:
    event_loop.close()
```

получим:

```
registering callbacks on future
setting result of future
1: future done: the result
2: future done: the result
```

Одновременное выполнение задач. Класс Task

Задачи - один из основных путей взаимодействия с циклом обработки. Tasks wrap coroutines and track when they are complete. Задачи - это подклассы Future, таким образом другие корутины могут ожидать их и каждая имеет результат, который можно взять у задачи после ее завершения.

Запуск задачи

Чтобы запустить задачу, нужно сначала сделать экземпляр задачи функцией **create_task()**. Эта задача будет выполняться как часть параллельных операций, управляемых event loop, до тех пор, пока цикл работет и корутина не закончилась.

В примере ожидаем с помощью **await**, пока задача не закончится и из нее можно будет взять результат.

```
asyncio_create_task.py
import asyncio

async def task_func():
    print('in task_func')
    return 'the result'

async def main(loop):
    print('creating task')
    task = loop.create_task(task_func())
    print('waiting for {!r}'.format(task))
    return_value = await task
    print('task completed {!r}'.format(task))
    print('return value: {!r}'.format(return_value))

event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

получим:

```
creating task
waiting for <Task pending coro=<task_func() running at
asyncio_create_task.py:12>>
in task_func
task completed <Task finished coro=<task_func() done, defined at
asyncio_create_task.py:12> result='the result'>
return value: 'the result'
```

Cancelling Task

По ссылке на экземпляр Task, которую вернул create_task(), можно отменить задачу до ее завершения метдом **calcel()**.

В примере создается и потом отменяется задача до старта event loop. В результате в функции run_until_complete() возникает исключение CancelledError.

```
asyncio_cancel_task.py
import asyncio


async def task_func():
    print('in task_func')
    return 'the result'


async def main(loop):
    print('creating task')
    task = loop.create_task(task_func())

    print('canceling task')
    task.cancel()

    print('canceled task {!r}'.format(task))
    try:
        await task
    except asyncio.CancelledError:
        print('caught error from canceled task')
    else:
        print('task result: {!r}'.format(task.result()))


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

получаем:

```
creating task
canceling task
canceled task <Task cancelling coro=<task_func() running at
asyncio_cancel_task.py:12>>
caught error from canceled task
```

Если задача была отменена, когда она ожидала другую concurrent операцию, эту задачу уведомляют об отмене возникновением исключения CancelledError в том месте, где происходило ожидание.

Обработка исключения позволяет, если необходимо, зачистить уже выполненную работу.

Catching the exception provides an opportunity to clean up work already done, if necessary.

```
asyncio_cancel_task2.py
import asyncio


async def task_func():
    print('in task_func, sleeping')
    try:
        await asyncio.sleep(1)
    except asyncio.CancelledError:
        print('task_func was canceled')
        raise
    return 'the result'


def task_canceller(t):
    print('in task_canceller')
    t.cancel()
    print('canceled the task')


async def main(loop):
    print('creating task')
    task = loop.create_task(task_func())
    loop.call_soon(task_canceller, task)
    try:
        await task
    except asyncio.CancelledError:
        print('main() also sees task as canceled')


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

получили:

```
creating task
in task_func, sleeping
in task_canceller
canceled the task
task_func was canceled
main() also sees task as canceled
```

Создание Task из корутины

Функция **ensure_future()** возвращает объект Task, связанную с выполнением корутины. Его можно передать в другой код, который может ожидать его, ничего не зная о том, как была сделана или вызвана оригинальная корутина.

Заметьте, что корутина, переданная в ensure_future() не стартует до тех пор, пока что-то не использует await, чтобы разрешить ее выполнение.

```
# asyncio_ensure_future.py
import asyncio


async def wrapped():
    print('wrapped')
    return 'result'


async def inner(task):
    print('inner: starting')
    print('inner: waiting for {!r}'.format(task))
    result = await task
    print('inner: task returned {!r}'.format(result))


async def starter():
    print('starter: creating task')
    task = asyncio.ensure_future(wrapped())
    print('starter: waiting for inner')
    await inner(task)
    print('starter: inner returned')


event_loop = asyncio.get_event_loop()
try:
    print('entering event loop')
    result = event_loop.run_until_complete(starter())
finally:
    event_loop.close()
```

получим:

```
entering event loop
starter: creating task
starter: waiting for inner
inner: starting
inner: waiting for <Task pending coro=<wrapped() running at asyncio_ensure_future.py:12
>>
wrapped
inner: task returned 'result'
starter: inner returned
```

Управление выполнением корутин

Линейный поток выполнения (control flow) ряда корутин легко организуется с помощью встроенного ключевого слова **await**. Для более сложной стуктуры выполнения корутин, позволяющей одной корутине ожидать завершения нескольких других параллельных задач, можно использовать инструменты asyncio.

Ожидание нескольких корутин

Часто полезно разбить одну операцию на много частей и выполнять их независимо. Например, даунлоад нескольких удаленных ресурсов или querying remote APIs. Если порядок выполнения этих задач не важен, и количество операций может быть любым, можно использовать **wait()** для приостановки одной корутины до завершения других фоновых (background) операций.

Внутри **wait()** использует set для хранения созданных экземпляров Task, что приводит к их запуску и завершению в произвольном порядке. Из wait() возвращается кортеж из двух множеств - завершенные и подвешенные (pending) задачи.

```
# asyncio_wait.py
import asyncio

async def phase(i):
    print('in phase {}'.format(i))
    await asyncio.sleep(0.1 * i)
    print('done with phase {}'.format(i))
    return 'phase {} result'.format(i)

async def main(num_phases):
    print('starting main')
    phases = [
        phase(i)
        for i in range(num_phases)
    ]
    print('waiting for phases to complete')
    completed, pending = await asyncio.wait(phases)
    results = [t.result() for t in completed]
    print('results: {!r}'.format(results))

event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(3))
finally:
    event_loop.close()
```

получим:

```
starting main
waiting for phases to complete
in phase 0
in phase 1
in phase 2
done with phase 0
done with phase 1
done with phase 2
results: ['phase 1 result', 'phase 0 result', 'phase 2 result']
```

Подвешенные (pending) операции будут возвращены, только если wait используют, задавая таймаут.

Эти оставшиеся фоновые задачи должны быть или отменены или закончены by waiting for them.

Оставляя эти задачи в подвешенном состоянии пока event loop продолжает работать, мы позволяем этим задачам выполниться в будущем, что нежелательно, если мы посчитали общую операцию прерванной. Если в конце процесса будут подвешенные операции, то получим warnings.

```
asyncio_wait_timeout.py
import asyncio


async def phase(i):
    print('in phase {}'.format(i))
    try:
        await asyncio.sleep(0.1 * i)
    except asyncio.CancelledError:
        print('phase {} canceled'.format(i))
        raise
    else:
        print('done with phase {}'.format(i))
        return 'phase {} result'.format(i)


async def main(num_phases):
    print('starting main')
    phases = [
        phase(i)
        for i in range(num_phases)
    ]
    print('waiting 0.1 for phases to complete')
    completed, pending = await asyncio.wait(phases, timeout=0.1)
    print('{} completed and {} pending'.format(
        len(completed), len(pending),
    ))
    # Cancel remaining tasks so they do not generate errors
    # as we exit without finishing them.
    if pending:
        print('canceling tasks')
        for t in pending:
            t.cancel()
    print('exiting main')


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(3))
finally:
    event_loop.close()
```

получим:

```
starting main
waiting 0.1 for phases to complete
in phase 1
in phase 0
in phase 2
done with phase 0
1 completed and 2 pending
cancelling tasks
exiting main
phase 1 cancelled
phase 2 cancelled
```

Сбор результатов с корутин

Если фоновые фазы хорошо определены, и только результаты этих фаз имеют значение, то для ожидания множественных операций можно использовать функцию **gather()**

Задачи, созданные функцией gather() не видны, то есть их нельзя отменить. Возвращается список результатов в том же порядке, в котором аргументы были переданы в gather(), вне зависимости от того, в каком порядке задачи на самом деле выполнялись и завершились.

```
# asyncio_gather.py
import asyncio

async def phase1():
    print('in phase1')
    await asyncio.sleep(2)
    print('done with phase1')
    return 'phase1 result'

async def phase2():
    print('in phase2')
    await asyncio.sleep(1)
    print('done with phase2')
    return 'phase2 result'

async def main():
    print('starting main')
    print('waiting for phases to complete')
    results = await asyncio.gather(
        phase1(),
        phase2(),
    )
    print('results: {!r}'.format(results))

event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main())
finally:
    event_loop.close()
```

получим:

```
starting main
waiting for phases to complete
in phase2
in phase1
done with phase2
done with phase1
results: ['phase1 result', 'phase2 result']
```

Handling Background Operations as They Finish

as_completed() - это генератор, который управляет выполнением предоставленных ему корутин и выдает их результаты по одному, не дожидаясь завершения работы других корутин. Так же как и в wait(), порядок не гарантирован, то не надо ждать завершения всех задач, чтобы начать обработку результатов.

В примере стартует несколько фоновых фаз, которые завершаются в обратном порядке. По мере использования генератора, цикл ожидает результата корутины, используя await.

```
# asyncio_as_completed.py
import asyncio


async def phase(i):
    print('in phase {}'.format(i))
    await asyncio.sleep(0.5 - (0.1 * i))
    print('done with phase {}'.format(i))
    return 'phase {} result'.format(i)


async def main(num_phases):
    print('starting main')
    phases = [
        phase(i)
        for i in range(num_phases)
    ]
    print('waiting for phases to complete')
    results = []
    for next_to_complete in asyncio.as_completed(phases):
        answer = await next_to_complete
        print('received answer {!r}'.format(answer))
        results.append(answer)
    print('results: {!r}'.format(results))
    return results


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(3))
finally:
    event_loop.close()
```

получим:

```
starting main
waiting for phases to complete
in phase 0
in phase 2
in phase 1
done with phase 2
received answer 'phase 2 result'
done with phase 1
received answer 'phase 1 result'
done with phase 0
received answer 'phase 0 result'
results: ['phase 2 result', 'phase 1 result', 'phase 0 result']
```

Примитивы синхронизации

Хотя приложения asyncio обычно запускаются как single-threaded process, они по-прежнему сделаны как параллельные приложения. Каждая корутина или задача может быть исполнена в произольном порядке, основываясь на прерываниях от I/O и других внешних событиях. Для поддержки безопасной concurrency, asyncio включает реалзицию таких же низкоуровневых примитивов, что и в модулях threading и multiprocessing.

- asyncio.Lock
- asyncio.Event
- asyncio.Condition
- asyncio.Queue

Приведем примеры для Lock и Queue.

Lock

A Lock can be used to guard access to a shared resource. Only the holder of the lock can use the resource. Multiple attempts to acquire the lock will block so that there is only one holder at a time.

A lock can be invoked directly, using await to acquire it and calling the release() method when done as in coro2() in this example. They also can be used as asynchronous context managers with the with await keywords, as in coro1().

```
# asyncio_lock.py
import asyncio
import functools


def unlock(lock):
    print('callback releasing lock')
    lock.release()


async def coro1(lock):
    print('coro1 waiting for the lock')
    with await lock:
        print('coro1 acquired lock')
    print('coro1 released lock')


async def coro2(lock):
    print('coro2 waiting for the lock')
    await lock
    try:
        print('coro2 acquired lock')
    finally:
        print('coro2 released lock')
        lock.release()


async def main(loop):
    # Create and acquire a shared lock.
    lock = asyncio.Lock()
    print('acquiring the lock before starting coroutines')
    await lock.acquire()
    print('lock acquired: {}'.format(lock.locked()))

    # Schedule a callback to unlock the lock.
    loop.call_later(0.1, functools.partial(unlock, lock))

    # Run the coroutines that want to use the lock.
    print('waiting for coroutines')
    await asyncio.wait([coro1(lock), coro2(lock)]),


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop))
finally:
    event_loop.close()
```

получим:

```
acquiring the lock before starting coroutines
lock acquired: True
waiting for coroutines
coro1 waiting for the lock
coro2 waiting for the lock
callback releasing lock
coro1 acquired lock
coro1 released lock
coro2 acquired lock
coro2 released lock
```

Queue

An asyncio.Queue provides a first-in, first-out data structure for coroutines like a queue.Queue does for threads or a multiprocessing.Queue does for processes.

Adding items with put() or removing items with get() are both asynchronous operations, since the queue size might be fixed (blocking an addition) or the queue might be empty (blocking a call to fetch an item).

```
asyncio_queue.py
import asyncio


async def consumer(n, q):
    print('consumer {}: starting'.format(n))
    while True:
        print('consumer {}: waiting for item'.format(n))
        item = await q.get()
        print('consumer {}: has item {}'.format(n, item))
        if item is None:
            # None is the signal to stop.
            q.task_done()
            break
        else:
            await asyncio.sleep(0.01 * item)
            q.task_done()
    print('consumer {}: ending'.format(n))


async def producer(q, num_workers):
    print('producer: starting')
    # Add some numbers to the queue to simulate jobs
    for i in range(num_workers * 3):
        await q.put(i)
        print('producer: added task {} to the queue'.format(i))
    # Add None entries in the queue
    # to signal the consumers to exit
    print('producer: adding stop signals to the queue')
```

```
    for i in range(num_workers):
        await q.put(None)
    print('producer: waiting for queue to empty')
    await q.join()
    print('producer: ending')


async def main(loop, num_consumers):
    # Create the queue with a fixed size so the producer
    # will block until the consumers pull some items out.
    q = asyncio.Queue(maxsize=num_consumers)

    # Scheduled the consumer tasks.
    consumers = [
        loop.create_task(consumer(i, q))
        for i in range(num_consumers)
    ]

    # Schedule the producer task.
    prod = loop.create_task(producer(q, num_consumers))

    # Wait for all of the coroutines to finish.
    await asyncio.wait(consumers + [prod])


event_loop = asyncio.get_event_loop()
try:
    event_loop.run_until_complete(main(event_loop, 2))
finally:
    event_loop.close()
```

получим:

```
consumer 0: starting
consumer 0: waiting for item
consumer 1: starting
consumer 1: waiting for item
producer: starting
producer: added task 0 to the queue
producer: added task 1 to the queue
consumer 0: has item 0
consumer 1: has item 1
producer: added task 2 to the queue
producer: added task 3 to the queue
consumer 0: waiting for item
consumer 0: has item 2
producer: added task 4 to the queue
consumer 1: waiting for item
consumer 1: has item 3
producer: added task 5 to the queue
producer: adding stop signals to the queue
consumer 0: waiting for item
consumer 0: has item 4
consumer 1: waiting for item
consumer 1: has item 5
producer: waiting for queue to empty
consumer 0: waiting for item
consumer 0: has item None
consumer 0: ending
consumer 1: waiting for item
consumer 1: has item None
consumer 1: ending
producer: ending
```

Задачи

Во всех случаях подвисающие соединения, неожиданно отвалившиеся клиенты или их обилие не вешают сервер и не мешают другим клиентам получать данные. И всё это в один поток и без сложной ручной работы с select/poll.

0. Эхо-сервер

1. Сервер времени

Каждому подключенному раз в секунду посылают время.

1. IRC чат

Чат на TCP, где каждый может писать всем (типа IRC, только без комманд). В качестве клиента - telnet (или putty в режиме telnet или raw).