Последовательности и цикл for

Для перехода к последовательностям сначала немного глобальной информации. Всё наше пространство имён в питончике представляет из себя 7 основных групп объектов (которые тоже сами по себе объекты), которые показаны с помощью операции *dir.* Объекты этих групп хранятся в globals() - словарике глобальных переменных

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__']
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class
'_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {},
'__builtins__': <module 'builtins' (built-in)>}
```

Основной и более всего ами используемый объект - модуль встроенных объектов: все возможные виды ошибок, базовые объекты типа True False None, базовые функции

```
>>> __builtins__
<module 'builtins' (built-in)>
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BaseExceptionGroup', 'BlockingIOError', 'BrokenPipeError', 'BufferError',
'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError',
'Ellipsis', 'EncodingWarning', 'EnvironmentError', 'Exception', 'ExceptionGroup',
'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning',
'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError',
'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'PermissionError',
'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning',
'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True',
'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__', '__debug__',
'__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs',
'aiter', 'all', 'anext', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable',
'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir',
'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr',
'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter',
'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set',
'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

В питоне, как нигде, наглядно простроена инкапсуляция всего. Любые объекты - это контейнеры других имён Таблицы Индексных Дескрипторов (ОПЯТЬ ОНА? Да, опять она), всё во всё вложено, любые операции с объектами есть по сути своей методы объектов. На их вызовы у нас, конечно, есть синтаксический сахар

```
>>> a, b = 4, 5
>>> a + b
9
>>> a.__add__(b)
9

>>> dir(a) # Методы классов, для которых синтаксический сахар существует, обозначаются конструкциями вида __операция__
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'as_integer_ratio', 'bit_count', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']

>>> (1237).__add__(123)
1360
>>> (1237).__divmod__(123)
(10, 7)

>>> len("qweqwe")
6
>>> s = "qweqwe"
>>> len(s)
6
>>> s.__len__()
6
>>> s.__len__() is len(s)
True
```

От понимания объектности всего переходим к понятию Протокола. Протокол это набор правил и\или критериев, которым должен удовлетворять объект \ алгоритм

```
>>> a, b = "!@#", 4
>>> a.__mul__(b)
'!@#!@#!@#!@#'
>>> str.__mul__(a, b)
'!@#!@#!@#!@#'
>>> b * a
'!@#!@#!@#!@#'
>>> b.__mul__(a)
NotImplemented
>>>
>>> # Строка умеет себя умножать на число, НО число не умеет себя умножать на строку
>>> # Числовой протокол: Если при операции появляется обЪект типа NotImplementedType, то запускает rmul для элементов в обратном порядке
>>> a.__rmul__(b)
'!@#!@#!@#!@#'
```

Последовательность - тоже своего рода протокол. Просто объект такого рода должен поддерживать соответствующие методы, типа уметь итерироваться (выдавать следующий элемент)

Теперь про Цикл for. Это не привычный нам for из Си и плюсов (в плюсах есть аналогичная штука - range-based for). Здесь каждый элемент последовательности поочерёдно связывается с именем локальной переменной, после чего с ним проводятся операции

```
>>> for var in (1, 2, 3, "QQ"):
...     # каждый элемент поочерёдно связывается именем var
...     print(var)
...
1
2
3
QQ
```

Можно и много элементов распаковывать, т.е. каждый объект воспримется, как ещё одна последовательность\набор значений и множественно присвоится

```
>>> for a, b in ["QW", "WE", (0, 9)]:
...     print(a, b)
...
Q W
W E
0 9
>>> for a, *b, c in ["QW", "WertE", (0,8.5, 9)]:
...     print(a, b, c)
...
Q [] W
W ['e', 'r', 't'] E
0 [8.5] 9
```

break, continue, else для for - аналогично while

```
>>> for n in 1, 2, 13, 4, 5:
...     if n == 13:
...         print("OOPS")
...         break
... else:
...     print("no 13")
OOPS

>>> for n in 1, 2, 3, 4, 5:
...     if n == 13:
...         print("OOPS")
...         break
... else:
...     print("no 13")
no 13
```

Для последовательностей существуют логические функции all и any, работающие как всеобщий and и or соответственно

>>> all((True, False, True))
False
>>> any([True, False, True])
True

У последовательностей есть методы принадлежности in | not in, а также индексирование(это операция [], метод __getitem__())

>>> s = "qwertyui"
>>> s[0]
'q'
>>> s[1]
'w'
>>> s[-1]
'i'

>>> l = 1, 4, 3456, 2, 3
>>> len(l)
5
>>> max(l)
3456

В последовательностях существует селекционирование - возможность выделить отдельным объектом. Операция для этого действия - конструкция вида [start:stop:step]. По умолчанию start = 0; end = len(последовательность); step = 1. Допускается шаг в обратную сторону (отрицательным числом), допускается указание элементов считая с конца (отрицательными числами {н-р, -1 -ый элемент это первый с конца, т.е. последний}). Селекционирование работает от start включительно до stop невключительно. Единственное исключение: с отрицательным шагом для включения нулевого элемента надо писать умолчание

```
>>> s
'qwertyui'
>>> s[1:4]
'wer'
>>> s[0:8:3]
'qru'
>>> s[::3]
'qru'
>>> s[:]
'qwertyui'
>>> # Это была копия последовательности

>>> s[-1:1:-1]
'iuytre'
>>> s[-1:0:-1]
'iuytrew'
>>> s[-1::-1] # Исключение
'iuytrewq'

>>> l
(1, 4, 3456, 2, 3)
>>> l.index(4) # метод возвращает первую позицию, на которой встречается объект
1
>>> l.index(12345)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: tuple.index(x): x not in tuple
>>> l.count(4)
1
>>> l.count(12345)
0
```

Создадим класс, который будет уметь обрабатывать последовательности

```
>>> class C:
...     def __getitem__(self, arg):
...             return arg
...
>>> c = C()
>>> c[123]
123
>>> c[123:456]
slice(123, 456, None)
```

Получили объект slice - объект, описывающий характеристики последовательности. При этом, по сути, это просто кортеж с именованными параметрами, запихивать туда можно всё, что угодно

```
>>> res = c[123:456]
>>>
>>> res.start
123
>>> res.stop
456
>>> res.step

>>> res = c[1:5:100]
>>> res.step
100

>>> c["ASR"]
'ASR'
>>> c["ASR":4]
slice('ASR', 4, None)

>>> c["ASR":4:(1, 2)]
slice('ASR', 4, (1, 2))
>>> c["ASR":4:(1, 2)].step
(1, 2)
```

Вспоминаем приколы со строками: Индексирование в строке - вычисляемая величина, строки состоят из строк, поэтому индексы можно спамить сколько хотим

```
>>> s
'qwertyui'
>>> s[3]
'r'
>>> s[3][0]
'r'
>>> s[3][0][-1]
'r'
>>> s[3][0][-1][0]
'r'
>>> s[3][0][-1][0][0]
'r'
```

list comprehension
Генерировать последовательности можно прямо с ходу, задавая формульно описание последовательности

```
>>> l = [1,2,3,4,5]
>>> l
[1, 2, 3, 4, 5]

>>> l = [i*2 + 1 for i in range(2, 6)]
>>> l
[5, 7, 9, 11]

>>> l = [i*2 + 1 for i in range(2, 11)]
>>> l
[5, 7, 9, 11, 13, 15, 17, 19, 21]

>>> l = [i*2 + 1 for i in range(2, 11) if i != 6]
>>> l
[5, 7, 9, 11, 15, 17, 19, 21]
```

Почему не надо делать linked list в питоне

```
>>>
>>> l = [1, None]
>>> l[1] = [2, None]
>>> l
[1, [2, None]]
>>> # извращунство
```

Range-генераторы (тут для красивого вывода используем спец.функцию из библиотеки pprint)

```
>>> import pprint
>>> a = [list(range(i, i + 12)) for i in range(10, 18)]
>>> pprint.pprint(a)
[[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21],
 [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22],
 [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23],
 [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24],
 [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25],
 [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26],
 [16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27],
 [17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]]
```

Подстава и решение по генерации

```
>>> a = [[0]* 12] * 8
>>> pprint.pprint(a)
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
>>> a[5][7] = 12345
>>> pprint.pprint(a)
[[0, 0, 0, 0, 0, 0, 0, 12345, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 12345, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 12345, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 12345, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 12345, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 12345, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 12345, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 12345, 0, 0, 0, 0]]
```

т.к. операция умножения просто делать множество тех же самых объектов по одной ссылке на изначальный, мы просто наспамили себе одного и того же объекта.

Избегаем это через range, где заданное количество раз создаём *разные, но равные* объекты

```
>>> a = [[0]* 12 for i in range(8)]
>>> pprint.pprint(a)
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
>>> a[5][7] = 12345
>>> pprint.pprint(a)
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 12345, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

range - индексируемый объект

```
>>> r = range(1, 200, 4)
>>> r[24]
97
>>> r[-3]
189
```