1. Объекты, переменные.

В питоне все есть объект.

Каждый объект имеет:

- Идентификатор. Это числовое значение, адрес в памяти, по которому располагается объект. Можно узнать с помощью встроенной функции id() (id не меняется) У None id меняется!
- Тип. Функция type()
- Значение.
- Счетчик ссылок. Он показывает, сколько раз другие объекты ссылаются на данный. Как только счетчик становится =0, объект уничтожается.

Выделяют два типа объектов:

1) Immutable. В течение времени жизни их значение не меняется. (string, tuple)
   Например, при присвоении строковой переменной нового значения, создается новый объект, и переменная начинает на него ссылаться. Изменится и ее идентификатор.
   Плюсы неизменяемых последовательностей: из за меньшего набора операций возникает меньше расходов по памяти и по времени, их можно использовать во множествах и словарях.
2) Mutable. Значение может быть изменено. (list)

Создание объекта: Выделяется область в памяти, а затем инициализируется. Потом в пространстве имен заводится переменная и в нее добавляется ссылка на объект.
Объекты везде передаются по ссылкам.
Чтобы узнать, какие переменные доступны в локальном пространстве имен, есть функция locals(). Она возвращает словарь.
dir - словарь всех методов/полей/и иже с ними для данного объекта
Для копирования можно использовать модуль copy.
copy.copy() – поверхностное копирование (shallow copy)
copy.deepcopy() – полное копирование.

Любой объект - словарь, доступ к нему осуществляется через obj.__dict__. Таким образом, можно “на лету” править список методов, ловя дикое нарушение типа безопасности (но если делать аккуратно - то все норм):

>>> t.__dict__

{'_is_stopped': False, '_args': (), '_daemonic': True, '_started': <threading.Event object at 0x02E1CBB0>, '_ident': None, '_target': <function <lambda> at 0x005284F8>, '_name': 'Thread-1', '_tstate_lock': None, '_kwargs': {}, '_initialized': True, '_stderr': <idlelib.PyShell.PseudoOutputFile object at 0x02A33730>}

>>> t.__dict__['test'] = 100500

>>> t.test

100500

>>> type(t)

<class 'threading.Thread'>

Создание экземпляра класса:

1) выделяем память new
2) создаём точную копию объекта класса
3) вызываем init

То есть если init пуст, то dir(ClassName) и dir(ObjectOfClassName) идентичны!

ЛЮБОЙ ТИП ТОЖЕ ОБЪЕКТ!

2. Функции. Аргументы функции.

Функция - объект, у которого определен метод __call__ - он вызовется при применении оператора “круглые скобки”

По pep8 – в именах функций только маленькие буквы, слова разделяются _

Начинается объявление с ключевого слова def. После него идет имя функции. После имени в круглых скобках задается список параметров. Тело функции пишется с отступом со следующей строки. Благодаря механизму кортежей, функции в питоне могут возвращать одновременно множество объектов. Если в функции отсутствует оператор return, или же он вызван без параметров, то функция возвращает специальное значение None.

Позиционные аргументы – сопоставление: первый переданный аргумент – это первый аргумент, который принимает функция и т.д. * - распаковывается список или любая другая последовательность. Args - tuple

Именованные аргументы. ** - так распаковывается словарь. Kwargs – словарь.

Можно использовать значения по умолчанию.

Несколько полезных примеров:

```
>>> def f(a, b, c):
        print(a, b, c)
>>> f(1, 2, b=3)
TypeError: f() got multiple values for argument 'b'

>>> def f(*args, b):
        print(args, b)
>>> f(1, 2, 3)
TypeError: f() missing 1 required keyword-only argument: 'b'
>>> f(1, 2, 3, b=3)
(1, 2, 3) 3

>>> def f(*args, **kwargs, c):
        print(args, kwargs, c)
SyntaxError: invalid syntax
```

Попытка очистить словарь приводит к странному поведению: словарь очищается, но некоторые функции (__call__, __type__, e.t.c.) продолжают работать. Все остальные же кидают RunTimeError

Анонимные функции

Для их создания используется lambda. Содержат одно выражение.

3. Базовые структуры данных: tuple, list, set, frozenset, dict.

Tuple – неизменяемый тип. Произвольная последовательность объектов. Чтобы создать тапл, надо перечислить элементы через запятую либо для создания пустого тапла воспользоваться конструктором tuple() или просто (). Можно обращаться по индексу, вкладывать одни таплы в другие.

List – похож на тапл, но изменяемый. Чтобы создать лист, надо перечислить элементы через запятую в квадратных скобках либо для создания пустого листа воспользоваться конструктором list() или просто []. Методы:

append() - добавление новых элементов в конец списка

insert(i) - добавление элементов в произвольное место списка

pop(i) - удаление элемента с конца списка (по умолчанию). Это позволяет удобно пользоваться списком, например как стеком

Set – изменяемый тип. множества. не предполагает порядка элементов. Элементы в нем не повторяются. Можно поместить любые из неизменяемых типов (например, лист нельзя). Чтобы создать множество, надо перечислить элементы через запятую в фигурных скобках {}. Для создания пустого множества используется конструктор set(). Методы:

add() - добавление нового элемента в множество

remove() - удаление элемента из множества

difference() - разность множеств

intersection() - пересечение множеств

union() - объединение множеств

issubset() - принадлежит ли одно множество другому

issuperset() - является ли множество надмножества другого

Frozenset – неизменяемый тип. Frozenset можно класть в другие сеты и фрозенсеты, т.к. он хешируемый. Создаётся с помощью конструктора frozenset().

Dict – изменяемый тип. Для создания словаря используется конструктор dict() или просто {}. Структура данных, которая хранит пары ключ-значение. Ключи являются уникальными. Словарь – изменяемый тип. Ключи – неизменяемые (хешируемые) типы. Словари устроены как хеш-таблицы. Методы:

items() - возвращает список пар, хранимых в словаре

keys() - возвращает список ключей

values() - возвращает список значений

4. Строки: str, bytes, bytearray. Кодировки.

Str – неизменяемый тип. упорядоченная последовательность символов, можно обращаться к символам по индексу. Вместо s[len(s) - 1] можно писать s[-1]. Одинарные и двойные кавычки равноправны. Каждый символ строки – тоже строка. Чтобы взять подстроку, используются срезы - s[a:b:step] вернёт каждый step-ый символ строки s, начиная с a-ого символа по b-ый символ, не включая. У объекта типа str есть методы:

find() - находит подстроку в строке

format() - используется для удобного создания строк

Оказывается, есть куча крутых фич, про которые рассказывают на лекциях:

```
>>> "{a.real} + {a.imag} j ".format(a = 1 + 2j)
```

'1.0 + 2.0 j '
>>> a = 1
>>> b = 2
>>> "{a} + {b} = {}".format(a + b, **locals())
'1 + 2 = 3'
Т.е. можно передавать словарь с бОльшим числом элементов, чем нужно, подставляются только нужные (в данном случае - первый элемент словаря).

join() - объединение нескольких строк в одну
replace() - используется для замены подстроки в строке
split() - из одной строки получает набор строк по разделителю
strip() - удаляет пробелы в начале и конце строки
encode() - из строки получает последовательность байтов

Bytes – неизменяемый тип. b'строка из аски-символов'. Элементы – байты. s[i] выведет аски-код i-ого символа. Последовательность байтов очень схожа со строками. Список методов схож с методами строк.

decode() - из последовательности байтов получает строку

Bytearray – изменяемый тип. Для создания нужно пользоваться конструктором bytearray(b'строка'), передавать конструктору строку байт. Также как и bytes, список методов схож с методами строк. Как и в листе, есть методы append(), insert(), pop().

Кодировки:

Между строками и объектами типа bytes и str существует взаимосвязь:
Для получения набора байт, кодирующих строку в определённой кодировке существует метод encode.

Encode:

>>>"Привет".encode('utf-8')
b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'
>>>"Привет".encode('cp1251')
b'\xcf\xf0\xe8\xe2\xe5\xf2'

Аналогичным образом, из строки байт можно получить обычную строку, раскодировав её в некоторой кодировке:

Decode:

>>>b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'.decode()
'Привет'

Как вы заметили, если не передавать кодировку - по умолчанию используется utf-8 (так же и в случае с encode).

>>>b'\xcf\xf0\xe8\xe2\xe5\xf2'.decode()
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xcf in position 0: invalid continuation byte

Если передать неверную кодировку - может возникнуть ошибка.
Связана она с тем, что данная последовательность байт не соответствует никакой последовательности символов в переданной кодировке.

5. Синтаксис регулярных выражений.

Синтаксис

Внутри регулярного выражения символы ., ^, $, *, +, ?, {, }, [,], \, |, (,) имеют специальное значение. Их следует экранировать с помощью слэша.

- .(точка) - любая буква, за исключением перевода строки. Но можно добавить дополнительный флаг re.DOTALL, тогда символ \n тоже будет включён.
- * - повторение буквы (набора букв) сколько угодно раз (возможно, 0)
- + - повторение буквы хотя бы 1 раз (эквивалентно ss*)
- ? - символ перед вопросом может быть, а может отсутствовать

  Операции *, +, ? являются жадными, то есть захватывают все символы по максимуму, жадно. Чтобы отключить жадность, после жадной операции надо поставить ?
- [...] - в квадратных скобках можно указать символы, которые могут встречаться на этом месте в строке. Можно перечислять символы подряд или указать диапазон через тире.
  - Например, [а-яА-ЯёЁ] соответствует любой русской букве («ё» и «Ё» не входят в диапазоны [а-я] и [А-Я] соответственно.)
  - Можно указать символы, которых не должно быть на этом месте в строке. Для этого используется ^ сразу после открывающейся квадратной скобки (например, [^0-9] — любые символы, кроме цифр от 0 до 9).
- ^ - привязка к началу строки или подстроки.
- $ - привязка к концу строки или подстроки.
- {...} - квантификаторы - количество вхождений символа в строку.
  - {n} — n вхождений символа в строку.
  - {n,} - n или более вхождений символа в строку.
  - {,m} - не более m вхождений
  - {n,m} — не менее n и не более m вхождений символа в строку.
  - * - ноль или более вхождений.
  - + - одно или большее число вхождений.
  - ? - ни одного или одно вхождение символа.
  - Все квантификаторы — жадные (ищется самая длинная подстрока). Чтобы ограничить жадность, необходимо после квантификатора указать символ ?.
- | - операция ИЛИ. например, под регулярное выражение abc|de попадут строки abc и de
- Стандартные классы:
  - \d — любая цифра. [0-9]
  - \w — любая цифра, буква или символ подчеркивания. [a-zA-Z0-9_]
  - \s — любой пробельный символ. [\t\r\n\f\v]
  - \D — не цифра. [^0-9]
  - \W — не буква, не цифра и не символ подчеркивания. [^\w]
  - \S — не пробельный символ. [^\s]

- (...) - Круглые скобки часто используются для группировки фрагментов внутри шаблона. Чтобы избежать захвата фрагмента, следует после открывающейся круглой скобки разместить символы ?:

К найденному фрагменту в круглых скобках внутри шаблона можно обратиться с помощью механизма обратных ссылок. Для этого порядковый номер круглых скобок указывается после слэша, например, \1. Нумерация скобок начинается с 1.

Существует также возможность сравнения шаблона с предыдущими и последующими символами:

- (?=...) - заглядывание вперед. Например, при поиске aba(?=caba) в строке abacaba в результат войдёт только aba
- (?!...) - отрицательный просмотр вперед.
- (?<=...) - положительный просмотр назад.
- (?<!...) - отрицательный просмотр назад.
- Требуется, чтобы шаблоны, используемые при просмотре назад, имели фиксированную длину. Т.е. , к примеру, можно использовать (?<=[,]), но нельзя использовать (?<=[,]+).

6. Использование регулярных выражений.

Для использования нужно импортировать модуль re.

Методы

compile() - создание скомпилированного шаблона регулярного выражения.
Формат: re.compile(регвыр, модификатор)
fullmatch() - проверяет полное сопоставление строки шаблону.
Поиск первого совпадения с шаблоном:

- match() - проверяет соответствие с началом строки. Возвращает объект Match, если найдено соответствие. Иначе — None.
- search() - проверяет соответствие с любой частью строки. Возвращает объект Match, если найдено соответствие. Иначе — None.

Методы объекта Match:

- group(group1, group2, …,groupN) - возвращает фрагменты, соответствующие шаблону. Если параметр не задан или указано значение 0, то возвращается фрагмент, полностью совпадающий с шаблоном.
- groups(default_value) - возвращает кортеж, содержащий значения всех групп. С помощью параметра можно указать значение, которое будет выводиться вместо None для групп, не имеющих совпадений.
- start(group_number_or_group_name) - индекс начала фрагмента.
- end(group_number_or_group_name) - индекс конца фрагмента.
- span(group_number_or_group_name) - кортеж, содержащий начальный и конечный индексы фрагмента.

Поиск всех совпадений с шаблоном:

- findall()- если соответствия найдены, то возвращается список с фрагментами(иначе пустой список). Если внутри шаблона более одной группы, то каждый элемент списка будет кортежем, а не строкой.
- finditer()- аналогичен findall(), но возвращает итератор, а не список. В каждой итерации цикла возвращается объект Match.

Форматы использования match() search() findall() finditer() :

1. regexp_object.method_name(string, start_position, end_position)
2. re.method_name(regexp, string, modificator)

Замена в строке:

- sub() - поиск всех совпадений с шаблоном и замена их указанным значением.
- subn() - аналогичен sub(), но возвращает не строку, а кортеж из двух элементов - измененной строки и количества произведенных замен.

Форматы методов sub() и subn():

- regexp_object.method_name(new_fragment, replacement string, max replacement count)
- re.method_name(template, new_fragment, replacement string, max replacement count)

Символ 'r' перед регулярным выражением указывает, что ничего в строке не должно быть экранировано. Для поиска символа '\' необходимо использовать четыре символа '\'в регвыре без 'r', тогда как с 'r' понадобится два символа '\'.

7. Классы. Методы __new__, __init__. Переопределение арифметических операций над объектами.

Класс – это объект, включающий в себя набор переменных и функции для управления этими переменными. Переменные – это атрибуты класса, функции – методы.
При создании объекта сначала вызывается метод __new__, затем метод __init__
__new__ вызывается в момент создания объекта нашего класса. Выделяется некоторое место в памяти + возвращается объект данного класса, шаблон. Для immutable объектов значения полей складываются в память здесь.

__init__ - инициализирует как конструктор.

Класс описывается по следующей схеме:

```
class class_name (base_class1, base_class2...base_classN):
        some_attrib = 0

        def some_method(*parameters):
                pass
```

Вышеописанная инструкция создает новый объект(класс) и присваивает ссылку на него идентификатору, указанному после ключевого слова *class*, в данном случае идентификатор - *class_name*. Выражения внутри инструкции *class* выполнятся при создании класса, а не при создании экземпляра класса. Класс создаётся при считывании программного модуля.

Создание переменной(атрибута) внутри класса аналогично созданию обычной переменной. Метод внутри класса создается также как и обычная функция, с помощью инструкции *def*. Методам класса в первом параметре автоматически передается ссылка на экземпляр класса(общепринято называть *self*). Доступ к атрибутам и методам класса производится через переменную self с помощью точечной нотации.

Для создания экземпляра класса используется следующий синтаксис:

```
class_exemplar = class_name(arguments)
```

При обращении к методам класса не нужно передавать ссылку на экземпляр класса в качестве параметра. Для этого используется следующий формат:

```
class_exemplar.method_name(arguments)
```

Обращение к атрибутам класса осуществляется аналогично. Все атрибуты класса открытые (public). Атрибуты являются частными, доступными только из методов объекта класса. Атрибуты можно создавать динамически после создания класса.

Важно понимать разницу между **атрибутами объекта класса** и **атрибутами экземпляра класса**. Атрибут объекта класса доступен всем экземплярам класса, но после изменения атрибута значение изменится во всех экземплярах класса. Атрибут экземпляра класса может хранить уникальное значение для каждого экземпляра и изменение такого атрибута не затронет значение одноименного атрибута в других экземплярах.

Инкапсуляция в Python работает лишь на уровне соглашения между программистами о том, какие атрибуты являются общедоступными, а какие — внутренними.
Одиночное подчеркивание в начале имени атрибута говорит о том, что переменная или метод не предназначен для использования вне методов класса, однако атрибут доступен по этому имени

А что означает двойное подчеркивание?
оно означает какое-то зарезервированное имя для функции(на счет переменных хз). то есть если я пишу у класса __init__ я должен понимать, что это инициализатор, а не просто непонятная херня типа moyafunction
b493801cdb-831abddfaf2Если коротко, то:
_internal
__private

`__new__` - это первый метод, который будет вызван при создании объекта. Он принимает в качестве параметров класс и потом любые другие аргументы. Он создает экземпляр объекта. Затем аргументы передаются в инициализатор __init__

```
class ClassName:
    def __new__(cls, *args, **kwargs):
        print(cls # cls - о б ъ е к т  к л а с с а
```

```
            print(args)
            print(kwargs)
            return object.__new__(cls)

        def __init__(self, x):
            print(self)
            print(x)

    c = ClassName(10)
```

Можно определить сравнения объектов класса методом __lt__ (<), __le__(<=), __gt__ (>), __ge__(>=), __eq__(==), __ne__(!=).
Метод для проверки истинности данного объекта __bool__
Методы __add__, __sub__, __mul__, __truediv__, __floordiv__, __mod__, __pow__ для реализации математических операций над объектами (+, -, *, /, //, %, ** соответственно)

8. Итераторы. Устройство цикла for.

Итератор - вспомогательный объект, который помогает перечислять элементы контейнера. Контейнер поддерживает итерирование, если определен метод __iter__.
Итератор - это объект, в котором определены два метода - __iter__() и __next__().
iter() и next() вызывают эти методы. iter() возвращает сам итератор. next() возвращает очередной элемент или выбрасывает исключение StopIteration (если элементы закончились).
Итератор невозможно отмотать назад. Всё что может делать итератор - возвращать себя и переходить к очередному элементу.
При итерировании по словарю будут возвращаться ключи.
range(a, b) является итератором.
Устройство for:
for пытается взять итератор от переданной ему последовательности.Затем пытается вызвать next от итератора. Если элементы остались, next возвращает очередной элемент, иначе если перечисление закончилось, то цикл обрабатывает исключение StopIteration и обрабатывает завершение цикла.

Break, else:
У циклов в языке python есть дополнительная возможность - необязательный блок else после тела цикла.
Выполняется в случае, если выполнение цикла было успешно завершено и не было прервано инструкцией break.

```
for/while …:
        # тело цикла
else:
```

зайдет только если break НЕ был вызван. Даже если в тело цикла мы не заходили

9. Генераторы. Генераторные выражения.

Генератором является все, что определяет методы __iter__(), __next__(), send(arg), throw(), close(). Генератор является итератором.
send() нужен для того, чтобы передать генератору некоторую информацию и скорректировать работу генератора. send передаёт информацию генератору через yield. Вы в функции пишете yield result. И вот этот yield - это тоже функция, которая имеет возвращаемое значение.
Если вызывается next, то yield возвращает None, если же вызывается send(something), то yield возвращает something. *Пример:*

```
>>> def range_or_print():
      current = 0
      while True:
            was_sent = yield current
            if was_sent is not None:
                  print(was_sent)
            current += 1
>>> a = range_or_print()
>>> next(a)
0
>>> next(a)
1
>>> a.send("i fuck your mother right now")
i fuck your mother right now
2
```

Синтаксис генераторных выражений: gen = (x for x in iterable). В генераторных выражениях могут присутствовать вложенные циклы for, а также if.

10. Генераторные функции.

Наличие слова yield внутри функции говорит о том, что функция генераторная. При вызове такой функции она возвращает генератор. Так как генераторная функция определяет методы итератора, то с генератором можно работать как с итератором. yield from помимо функциональности, аналогичной for и yield позволяет организовать связь между внешним генератором и внутренним генератором. send, применённый к внешнему генератору будет отправлен во внутренний генератор.
Метод throw() позволяет пробрасывать исключение внутри генератора. yield from пробрасывает в том числе и эти исключения.

11. list, set, dict comprehensions.

Comprehensions очень внешне похожи на генераторные выражения, но они существенно различны.
В случае с Comprehensions цикл for начинает выполняться сразу и все элементы,

возвращаемые на каждой итерации сохраняются в оперативную память. Если итераций было много - памяти может не хватить. В случае с генераторными выражениями никаких вычислений при их создании не происходит, а первые действия будут выполнены при вызове метода next, то есть генераторные выражения гораздо эффективнее по памяти, чем compr.
Comprehensions возвращают либо список, либо словарь, либо множества.
Например, a = [x*x for x in range(10)]. type(a) = list. Также как и генераторные выражения, в comprehensions можно использовать вложенные for, if.
b = {x*x for x in range(10)}. type(b) = set
c = {x: x*x for x in range(10)}. type(c) = dict

Модуль itertools:
chain(*iterables) возвращает iterable, который перечисляет по очереди каждый из переданных iterable-ов.
cycle(iterable) возвращает зацикленный принятый iterable
combinations(itetable, number) возвращает iterable, перечисляющий подмножества из множества элементов переданного iterable, содержащие number элементов.
permutations(iterable, number) аналогично с перестановками.
product(*iterables, repeat=1) вычисляет декартово произведение переданных iterables.
repeat - повторяет объект заданное число раз

12. Функции как объекты. Декораторы.

Так функции являются самостоятельными объектами, их можно сохранять в переменные. Функции можно передавать в качестве аргументов в другие функции. Например, встроенные функции map и filter принимают в качестве первого параметра функцию.
Для реализации простых функций в python существуют анонимные функции. Например, lambda x: x ** 2 (без слова return). в лямбда-функции должно быть одно простое однострочное выражение. Если этого недостаточно, значит надо выносить в отдельную функцию.
Можно также возвращать функции из других функций. Чтобы вернуть функцию, ее нужно определить внутри той функции, из которой возвращаем.
У функции есть следующие поля:
__name__ - хранит имя функции (при сохранении функции в переменную, имя останется неизменным). у лямбда-функций __name__='<lambda>'
__module__ - возвращает название модуля, в котором определена функция
__doc__ - возвращает docstring функции (краткая документации по функции в тройных кавычках ''' text '''). По умолчанию поле __doc__ - пустое.
Функции объявляют, чтобы:

- избежать дублирования кода
- выделить логические блоки кода для повторного использования
- улучшить читаемость кода
- повысить управляемость кода (удобно для нахождения ошибок)

Декоратор - это функция, которая принимает на вход функцию и возвращает новую функцию.
Есть полезный модуль functools. Он содержит некоторые функции и декораторы.

cmp_to_key принимает функцию сравнения двух элементов и возвращает функцию, получающую ключ по элементу, такой, что при сортировке по этому ключу получится то же, что и при сортировке по компоратору (в python сортировки есть только по ключу. Если проще написать функцию сравнения, то cmp_to_key нужен).

lru_cache - кеширующий декоратор

partial - принимает функцию и возвращает функцию с зафиксированными некоторыми параметрами.
Например, есть функция add(a, b).
Тогда inc(b) которая будет эквивалентна add(1, b) - это просто partial(add, 1).

reduce - лучше пояснить на примере.
reduce(func, [1, 2, 3]) выдаст результат выражения func(func(1, 2), 3).
Если передать необязательный параметр initial - он будет приписываться в начале к последовательности.
reduce(func, [1, 2, 3], -1) выдаст результат func(func(func(-1, 1), 2), 3)

total_ordering - декоратор для класса, который доопределяет все функции сравнения, если определены необходимые(всегда мечтал).
Например, если определены функции lt и eq(обязательна), то можно из этого определить le, ge, gt. Вот он это как раз и делает.

```
import functools

def log(arg):
    def decorator(func):
        @functools.wrapper(func)
        def wrapper(*args, **kwargs):
            print(str(arg))
            msg = "Called {} with {} and {}".format(func.__name__,
args, kwargs)
            print(msg)
            rv = func(*args, **kwargs)
            print("{}({}) -> {}".format(func.__name__, str(args) +
", " + str(kwargs), str(rv)))
            return rv
        return wrapper
    return decorator
```

13. Наследование в классах (без множественного наследования).

super.

По умолчанию наследование происходит от object

```
class A:
      def f(self): pass
class B(A):
      def f(self): pass
a = A()
b = B()
```

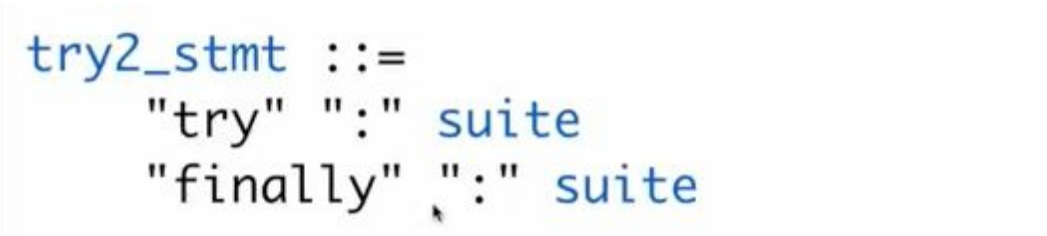
type(a) - возвращает последний тип в иерархии наследования. type(b) is A == False
isinstance(b, A) == True - функция для проверки наследования.
isinstance(a, object) == True, так как все классы наследуется от object.
issubclass(A, B) - функция для проверки наследования между классами. Является ли класс B наследником A.
a.__class__ - хранит информацию о классе объекта
B.__bases__ - хранит классы, от которых наследуется B
A.__subclasses__() - хранит классы, которые наследовались от A

При вызове метода какого-то класса, python сначала ищет метод в этом классе и если не находит, то ищет метод в родительских классах. Чтобы посмотреть в каком порядке будет вестись поиск методов, можно вызвать B.__mro__. Если метод не будет найден ни в одном из классов, генерируется исключение AttributeError.
Если надо вызвать метод, определённый в базовом классе, можно например вызвать A.f(b) или воспользоваться функцией super(B, b).f()
Чаще всего super используется в методе __init__. В python автоматически __init__ базового класса не вызывается. Для этого можно написать super().__init__()

14. Обработка исключений: `try`, `except`, `else`, `finally`, `raise`.

```
try1_stmt ::=
    "try" ":" suite
    ("except" [expression ["as" target]] ":" suite)+
    ["else" ":" suite]
    ["finally" ":" suite]
```

Выполнение: питон пытается выполнить то, что находится в try. Если исключения нет, то переходит в else. Если возникло исключение, то питон идет в except и проверяет соответствие исключений в том порядке, в котором они указаны, сохраняет в переменную, описанную через as объект, описывающий возникшее исключение, и выполняет блок кода except. В этом случае else не выполнится. Но finally выполняется всегда.
Вторая конструкция:



Генерация своих исключений:

если raise написан сам по себе, то будет повторно сгенерировано случившееся исключение, т.е пробросится выше.
sys.exc_info() - информация о последнем случившемся исключении. Возвращает тапл из типа исключения, самого исключения и специального объекта traceback.
Этот тапл выводится на экран, когда исключение не обрабатывается.
traceback.print_exc() выводит на экран traceback текущего исключения.
При возникновении исключения питон анализирует стек вызовов. Исключения “всплывают” по стеку вызовов.

15. Создание собственных классов исключений. Иерархии исключений.

Все исключения наследуются от BaseException. От него наследуется 4 типа исключений:
SystemExit - генерируется при вызове sys.exit()
KeyboardInterrupt - генерируется когда пользователь посылает сигнал завершения программы, обычно Ctrl C
GeneratorExit - генерируется когда у генератора зовут метод close
Exception - самый обширный. От него наследуются все основные исключения:

- **StopIteration** - когда перечисление элементов итератора завершилось
- **ArithmeticError** - арифметическая ошибка
    - ZeroDivisionError - деление на 0
    - FloatingPointError - порождается при неудачном выполнении операции с плавающей запятой. На практике встречается нечасто.
    - OverflowError - возникает, когда результат арифметической операции слишком велик для представления.
- AssertionError - когда метод assert возвращает False
- **AttributeError** - при обращении к несуществующему полю или методу класса
- BufferError - возможные ошибки с буфером обмена
- EOFError - вызвав input() и ничего ему не передав
- ImportError - при импорте несуществующего модуля
- LookupError - некорректный индекс или ключ
    - IndexError - индекс выходит за границы массива
    - KeyError - обращение по несуществующему ключу
- MemoryError - при попытке выделения слишком большого объёма памяти
- NameError - обращение к несуществующей переменной

- OSError - ошибки при работе с операционной системой, например FileNotFoundError (файл не найден)
- ReferenceError - при работе с weekref
- SyntaxError - синтаксические ошибки в коде
- TabError - табуляция и пробелы вперемешку
- SystemError - проблемы с интерпретатором
- TypeError - операция с типом, которая не поддерживается, например, сложить строку и число.
- ValueError - значение для функции не подходит под допустимый диапазон, например, math.acos(2)
- Warning - предупреждение

Для создания своего исключения нужно создать класс, который наследуется от Exception.

16. Устройство `with`.

Синтаксис:

```
with_stmt ::= "with" with_item
              ("," with_item)* ":" suite
with_item ::= expression ["as" target]
```

Контекст менеджер - любой объект, у которого есть методы:

- object.__enter__()
- object.__exit__(exc_type, exc_value, traceback)

Как работает with:

1. Сначала вызывается метод __enter__() и то что он вернул - сохраняется в переменную.
   Определяет, что должен сделать менеджер контекста в начале блока, созданного оператором `with`. Заметьте, что возвращаемое `__enter__` значение и есть то значение, с которым производится работа внутри `with`.
2. Вне зависимости, произошло исключение или нет в конце работы **ВСЕГДА** вызывается __exit__().
   Определяет действия менеджера контекста после того, как блок будет выполнен (или прерван во время работы). Может использоваться для контролирования исключений, чистки, любых действий которые должны быть выполнены незамедлительно после блока внутри with. Если блок выполнен успешно, `exception_type`,`exception_value`, и `traceback` будут установлены в `None`.
3. Если исключение было то передается информация о нем, если нет то __exit__(None, None, None).
4. В случае работы с файлами __enter__() возвращает сам объект описывающий файл и этот объект сохраняется в переменную VAR (синтаксис ‘with EXPR as

VAR: Block') а метод __exit__() закрывает файл.

Запись:

with A() as a, B() as b: suite

ЭКВИВАЛЕНТНО

```
with A() as a:
    with  B() as b:
        suite
```

Объекты являющиеся менеджерами контекста (поддерживающие with):

- file (open(...))
- socket (socket.socket(...))
- http.client.HTTPResponse(urllib.request.urlopen(...))
- И множество других объектов

Как сделать поддержку with для собственного объекта:

1) Определить методы __enter__() && __exit__() по правилам
2) Воспользоваться модулем .contextlib

Contextlib содержит:

- @contextlib.contextmanager - Декоратор
- contextlib.closing(thing) - из любого объекта поддерживающего close() можно сделать менеджер контекста. При __exit__ будет вызывать close()
- contextlib.suppress(*exceptions) - позволяет внутри блока with проигнорировать некоторые исключения.
  Пример: with contextlib.suppress(FileNotFoundError) - данный эксепшн будет проигнорирован в ходе выполнения блока with.
- class contextlib.ExitStack - для работы с неизвестным количеством файлов через with. После окончания работы они закрываются.

- Реализация closing с помощью декоратора.

```
import contextlib

@contextlib.contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

Декоратору передается генераторная функция. В этой функции один yield . В результирующем менеджере контекста то что будет возвращать __enter__() есть то что возвращает yield. В эту генераторную функцию можно с помощью throw передать исключение и обработать. в **ДАННОМ КОНКРЕТНОМ** случае с closing исключения не обрабатываются, они выбрасываются наружу, и всегда будет вызван метод close()

- Пример использования suppress - Удаление файла, не зависимо от его

```
import contextlib

with contextlib.suppress(FileNotFoundError):
    os.remove('somefile.tmp')
```

существования(файла нет - игнор)

17. Модули, пакеты, `import` и его варианты.

В python отдельные py-файлы называются модулями. Синтаксис:
import re
import numpy as np
from collections import Counter
from re import *
При импортировании модуля становятся доступны все объекты модуля через name.
Из модуля можно импортировать только конкретный объект.
import * - импортируется все из модуля в пространство имен, кроме тех имен, которые начинаются с одного подчеркивания. Так делать плохо, потому что в глобальных переменных появится много нового. Возможно, уже была какая то переменная с определенным именем и после импорта она будет замещена переменной из модуля.
Процесс импортирования модуля:

1. найти, в каком файле расположен данный модуль.

Во-первых, питон обращает внимание на кэш. Если там не был найден импортированный модуль, то питон пытается:
-поискать встроенный модуль
-некоторые модули бывают вмороженные в конкретную сборку файлов и он ищет там, обращается к некоторому списку путей, по которому он и пытается найти модуль.
sys.path отвечает за то, где будет производиться поиск модуля. Он начинается в текущей директории, а затем в некоторых системных папках. Значение этой переменной можно изменить как изнутри питона, так и снаружи, модифицируя переменную окружения PYTHONPATH.

2. создать соответствующие имена переменных в текущем пространстве имен.

Для создания своего модуля необходимо создать файл с расширением .py. Имя файла будет совпадать с именем модуля.
У модуля есть атрибуты:
__doc__ - docstring этого модуля
__file__ - файл, в котором он хранится
Конструкция if __name__ == ‘__main__’ позволяет определить, модуль запустили как самостоятельную программу или импортировали.

Пакеты содержат в себе наборы модулей. Отличительная черта пакета - это некоторая директория, которая содержит файл __init__.py. В этом файле можно описать код, который будет инициализировать пакет. Инициализационный код выполняется только один раз при импортировании пакета(даже если из него импортируются разные модули).
__all__ - список модулей, которые нужно импортировать при использовании *(звездочки).
Чтобы импортировать из одного модуля пакета другой модуль этого же пакета, надо указать не только имя импортируемого модуля, но и сначала имя самого пакета, так как поиск ведётся в текущей директории, а не в директории пакета.

Одни пакеты можно вкладывать в другие.

Чтобы импортировать из модуля по произвольному адресу: __path__.append('Your address')
Так же полезен модуль importlib

18. Пространства имён, области видимости.

Пространство имён - соответствие имени переменной и ссылки на объект, который хранит эта переменная.

Область видимости - участки кода из которого при использовании переменной будет доступен данный объект.

Пространства имён:

- пространство локальных имен
- пространство имен окружающих функций, начиная от самой ближайшей по вложенности
- пространство глобальных имён
- пространство встроенных имен

Доступ к пространствам имён:

- globals()
- locals()

Являются словарями, где ключ - имя переменной, значение - ссылка на объект.

Зарезервированные слова:

- global (если нужно изменить значение глобальной переменной внутри функции, если глобальной переменной до этого не было, то она будет создана)
- nonlocal (если нужно изменить не локальную переменную данной функции, будет подниматься вверх по вложенности функций, но смотрит только в локальных переменных)

Для присваивания по умолчанию создается новая переменная.
Локальные переменные не создаются в циклах и условиях.

19. Метод __del__. Удаление объектов при помощи подсчёта ссылок.

метод __del__ будет вызываться перед удалением объекта. В питоне реализован подсчет ссылок. Как только в какой-то переменной сохраняется ссылка на объект, то количество ссылок будет увеличено на 1.
В питоне объект удаляется только тогда, когда на него нет ссылок.
sys.getrefcount() - позволяет узнать количество ссылок на объект (на 1 больше, учитывая этот вызов функции). Слабые ссылки не увеличивают счетчик ссылок на объект.
Модуль `weakref.proxy` (от англ. weak reference - слабая ссылка) даёт средство для учёта объектов без создания ссылок на них. Когда объект больше не нужен, он автоматически удаляется из таблицы слабых ссылок и производится обратный вызов `weakref`-объектов. Если объект хранит ссылку сам на себя, то счетчик ссылок не станет 0.

Чтобы избежать подобных ситуаций, надо при вызове метода del() разрывать все циклические ссылки.
Помимо счетчика ссылок в питоне есть сборщик мусора.
Он доступен через модуль gc.
Его можно заставить включиться или выключиться, принудительно собрать мусор, выдать некую стату.

```
>>> a = "abc"
>>> b = a
>>> del a
>>> b
'abc'
>>> a
NameError: name 'a' is not defined
```

del удаляет лишь переменную из области видимости.
Сам же объект нельзя удалить, пока на него что то ссылается. И удаляет его уже непосредственно сборщик мусора, когда на него ничего не указывает.

20. Работа с файлами.

```
f = open("data.txt")   #открыть файл для чтения
f.read()          # можно указать кол-во символ для считывания
f.seek(i)          # перейти к i-ому символу
f.close()         # закрыть файл(вернуть ОС ее ресурсы)
f.readlines()     # возвращает список строк (считывает весь файл в ОП)

for line in f:                #считывает файл построчно
        print(line)

with open ('data.txt') as f:
        ...
```

После выхода из блока with файл закроется автоматически

```
f = open('data.txt', 'rb')          # считает байты
f = open('data.txt', 'w')           # открыть на запись
```

По умолчанию файл открывается на чтение

```
f.write('...')      # записать в файл, вернет количество успешно записанных символов
```

<u>Исключения при работе с файлами:</u>

- FileNotFoundError - пытаться открыть несуществующий файл
- IsADirectoryError - открыть директорию вместе файла
- PermissionError - пытаться записать в файл, который не поддерживает запись

21. Создание потоков, модуль `threading`.

потоки могут использовать общую область памяти. Создаются по средствам класса threading.Thread, которому в качестве аргумента передается функция, исполнять которую будет поток. В CPython потоки только лишь замедляют скорость исполнения программы, так как не будут реально распараллеливаться по причине GIL. В связи с этим они все же выполняются последовательно и все инструкции python де-факто атомарны. Возможна проблема лишь с не атомарностью больших команд, но работа со всеми коллекциями, чаще всего, атомарна. Убить поток нельзя, ибо никанон
Создание:

1) t = Thread(name="Vasya Pupkin", target=foo, {аргументы foo})
   t.start()
2)

```
class MyThread(threading.Thread):
        def __init__(self, a, b, c):
        threading.Thread.__init__(self)
        self.a = a
        self.b = b
        self.c = c

        def run(self):
        # do something, using self.a, self.b, self.c
        pass

th = MyThread(1, 2, 3)
th.start()
```

Ожидание завершения:

```
th.join() # блокирующий вызов
```

Можно сделать поток демоном (не будет мешать завершению работы программы):

```
th.setDaemon(True) # НЕ ПО PEP 8
```

В мультипоточной программе доступ к объектам иногда нужно синхронизировать.
С этой точки зрения все объекты (переменные) разделяются на:

- Неизменяемые. Если объект никто не меняет, то синхронизация доступа ему не нужна.
- Локальные. Если объект не виден остальным потокам, то доступ к нему синхронизировать тоже не требуется.
- Разделяемые и изменяемые. Необходима синхронизация.

В модуле threading есть полезный класс Lock, который позволяет синхронизировать потоки.

```
>>> lock = threading.Lock()
>>> lock.acquire() # захватить lock
>>> lock.release() # отпустить lock
```

Чтобы не забыть отпустить lock, можно воспользоваться конструкцией with:

```
>>> with lock:
        suite
```

22. Способы создания процессов

Процесс - самостоятельная единица ОС и может происходить ядерное распараллеливание. Процесс имеет как минимум 1 поток (главный).
Процессы имеют разные области памяти. В том числе, можно запустить их на разных интерпретаторах python (например p2 и p3)

1) os.fork для Unix-подобных. Быстр.При вызове fork у процесса возникает его идентичная копия. Возвращает идентификатор процесса (для родительского - его ребенка, а для дочернего- 0). По ним можно узнать в каком именно процессе мы сейчас находимся. Копирует все данные из одного процесса в другой. Могут возникнуть проблемы с процессобезопасностью. Для нормальной работы нужна синхронизация и взаимодействие между процессами. Взаимодействие может осуществляться с помощью file, signals, pipe, socket, mmap.
2) subprocess - позволяет создать новый процесс и обмениваться с ним информацией по трубам / стандартным потокам ввода/вывода
   cmd = "ping google.com"
   p = subprocess.Popen(cmd, shell = True)
   Мы можем проверять статус процесса, а также отправлять ему сигналы
   pid - идентификатор процесса.

3) multiprocessing.Process(target=foo) запуск нового процесса с исполняемой функцией
4) concurent.futures - позволяет удобно работать с процессами