Специальные методы классов и установка внешних библиотек

Цель занятия

После освоения темы вы:

- узнаете понятие и роль специальных методов классов в программировании на Python
- сможете использовать специальные методы в написании программ
- сможете использовать основные функции, методы, модули и библиотеки Python для написания собственных клиентских и серверных приложений.

План занятия

Конспект занятия

1. Специальные методы классов

Если с помощью функции `dir()` вывести все методы объекта, то в большом списке можно увидеть **специальные методы** — с двойным нижним подчеркиванием. Например:

```
s = 'Amat victoria curam'
dir(s)
```

['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', ...]

У этих методов в литературе есть и другие названия: dunder-методы (от Double UNDERscore) и магические методы.

Специальные методы обеспечивают механизм работы классов в Python. Они вызываются автоматически для определенных действий. Один из таких специальных методов — это `__init__` (конструктор), который используется при построении нового элемента класса.

По наличию или отсутствию специальных методов можно судить о том, какие операции поддерживаются данным классом объектов. Рассмотрим самые популярные.

Метод `__len__` сообщает размер коллекции и используется функцией `len`. Функция `len` работает со встроенными типами напрямую.

Определив метод `__len__` для собственного класса, мы можем «научить» функцию `len` работать с ним так же, как со встроенными типами.

Метод `__repr__` используется, чтобы показать на экране в более осмысленном виде представление объекта для вывода. Обычно он выдает текст, который можно скопировать и вставить в программу. Вывод выглядит так же, как мы написали ввод при создании объекта:

```
class Point:
      def __init__ (self, x, y):
            self.x = x
            self.y = y
      def __repr__(self):
            "”Возвращает строковое представление объекта для
вывода”"
            return f'Point({self.x},{self.y})'
>>> p = Point(1, -1)
```

```
>>> p
Point(1, -1)
```

Метод `__eq__` нужен для сравнения на равенство экземпляров классов с точки зрения их внутренней структуры:

```
class Point
      …
      def __eq__(self, other):
            return self.x == other.x and self.y == other.y
>>> p = Point(1, -1)
>>> p = Point(1, -1)
True
```

Нужно внимательно подходить к написанию специальных методов и понимать, что корректность их работы — это ответственность программиста. Можно написать метод `__eq__`, который будет всегда возвращать `True`, и все объекты будут считаться равными друг другу, что вызовет немало проблем в дальнейшей работе с кодом. Иногда для сравнения на равенство можно использовать не все атрибуты.

Важно! У любого класса обычно стоит реализовать как минимум три метода:

1. `__init__` (конструктор)
2. `__repr__` (отображение в текстовом виде)
3. `__eq__` (проверка на равенство)

2. Хеширование

Словари и множества работают как хеш-таблицы. Чтобы работать с объектом из хеш-таблицы, нужно его хешировать.

Давайте рассмотрим, с какими проблемами мы можем столкнуться. В данном примере мы не реализуем метод `__eq__` для класса и можем использовать такой объект в качестве ключа в словаре, но потом не сможем его найти:

```
class Point:
      …
      # Метод __eq__ не реализован
```

```
points = {Point(1, 1):10}
>>> points[Point(1, 1)]
KeyError: Point(1, 1)
>>> print('N{CONFUSED FACE}')
```

Без метода `__eq__` Python будет считать каждый объект этого класса уникальным. В случае если мы реализуем метод `__eq__`, то мы не сможем использовать объект в качестве ключа. Python сообщит нам, что объект не хеширован:

```
class Point:
    …
    # Метод __eq__ реализован
>>> points = {Point(1, 1):10}
TypeError: unhashable type; 'Point'
>>> print('N{SHOCKED FACE WITH EXPLODING HEAD}')
```

С решением этой ситуации нам поможет хеш-таблица. Принцип ее работы проиллюстрирован на рисунке 1.

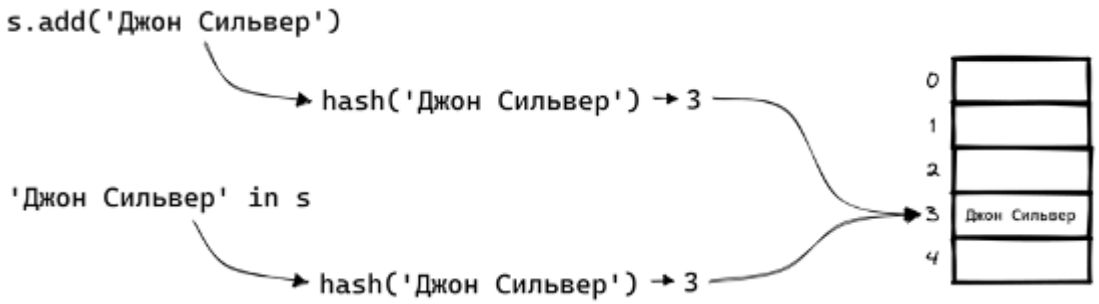

Рисунок 1. Хеш-таблица

Хеш-таблица представляет собой массив бóльшего размера, чем размер данных в нем. При добавлении элемента в хеш-таблицу Python вычисляет хеш-функцию этого элемента – хеш-функция может взять большой объект и превратить его в число. Это число будет давать нам индекс той ячейки, в которой находится элемент. Таким образом, преимущество хеш-таблицы — очень быстрый поиск за счет системы ячеек.

Метод `__hash__` нужно реализовать, чтобы использовать объект в хэш-таблицах (совместно с методом `__eq__`).

```
class Point
      …
      def __eq__(self, other):
            return self.x == other.x and self.y == other.y
      def __hash__(self):
      # ОПАСНО!
            return hash((self.x, other.y))
>>> p = Point(1, 1)
>>> points = {p: 10}
>>> p.x += 1
>>> points
{Point(2, 1): 10}

>>> points[p]
KeyError: Point(2, 1)
```

После того как мы использовали объект в качестве ключа, он может измениться. Словарь пытается найти данные там, где они больше не лежат. Чтобы правильно реализовать возможность хеширования, нужно гарантировать, что объект не будет изменяться. Сделать это сложно, потому что нет специального модификатора. Но мы можем использовать другой метод — метод `__setattr__`.

Метод `__setattr__` позволяет переопределить поведение объекта при записи в него какого-либо атрибута:

```
class Point:
      __frozen = False
      def __init__ (self, x, y):
            self.x = x
            self.y = y
            self.__frozen = True
      …
      def __setattr__(self, attr, value):
            if not self.__frozen:
```

```
            super().__setattr__(attr, value)
        else:
        return TypeError('Point object is immutable')
```

```
>>> p = Point(1, 1)
>>> p.x += 1
TypeError: Point object is immutable
```

В примере выше нам удалось сохранить хеширование объекта. Но сделать объект по-настоящему неизменяемым практически невозможно, но это и не нужно. Мы использовали «заморозку» только для того, чтобы уберечь код от ошибок и случайных действий.

3. Специальные атрибуты

Помимо специальных методов существуют специальные атрибуты, которые вызываются автоматически:

- **`__class__`** — ссылка на класс, экземпляром которого является данный объект;
- **`__dict__`** хранит все атрибуты объекта в виде словаря;
- **`__annotations__`** — словарь, содержащий аннотации типов, описанные в классе;
- **`__doc__`** — документ-строка, описанная в начале класса (именно ее выдает функция `help`). Этот атрибут выдает документацию.

4. Перегрузка операторов

Перегрузка оператора — это реализация оператором разной логики в зависимости от типов операндов, к которым он применяется.

Решим задачу: определить арифметические операторы для нашего класса.

```
class Vector:
    x: float
    y: float
    # ?
>>> a = Vector(1, 3)
>>> b = Vector(2, 1)
>>> c = a + 2*b # Мы хотим, чтобы это работало!
```

```
>>> c
Vector (5, 5)
```

Чтобы сложение сработало, нужно определить **метод `__add__`**:

```
class Vector:
    …
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
```

```
>>> a = Vector(1, 3)
>>> b = Vector(2, 1)
>>> a + b
Vector (3, 4)
```

Если `other` — не `Vector`, то на выводе мы получим ошибку о том, что число не содержит атрибута `x`:

```
>>> a = Vector(1, 3)
>>> a + 2
AttributeError: 'int' object has no attribute 'x'
```

Чтобы получить больше информации в сообщении ошибки, нужно для начала проверить является ли ваше число вектором с помощью `isinstance()`. После этого ошибка станет более информативной.

Для умножения используется **метод `__mul__`**. Он может быть реализован сложнее — например, вектор можно умножить на число.

Что нужно учитывать при написании метода умножение:

- от перестановки мест множителей может поменяться логика;
- Python пытается найти в классе `int` метод умножения на вектор, если в записи умножения число стоит первым.

Для решения этих проблем можно использовать модификатор метода (подставить вперед `r`) — **`__rmul__`**:

```
class Vector:
    …
```

```
    def __rmul__(self, other):
        return self * other
```

```
>>> a = Vector (1, 3)
>>> a * 2
Vector (2, 6)
```

Переставим местами аргументы:

```
>>> 2 * a
Vector (2, 6)
```

Арифметические операторы

Метод	Действие
__neg__	- a
__add__	a + b
__sub__	a - b
__mul__ (__rmul__	a * b
__truediv__	a /
__floordiv__	a // b
__mod__	a % b
__pow__	a ** b
__matmul__	a @ b (матричное произведение)

Битовые операции

Метод	Действие
__invert__	~ a
__and__	a & b
__or__	a | b

`__xor__`	`a ^ b`
`__rshift__`	`a >> b`
`__lshift__`	`a << b`

У всех методов есть модификация с буквой `i` (`inplace`):

`__iadd__` `a += b`

и с буквой `r` (`right`) — обратный порядок операндов:

`__radd__` `b + a`

5. Коллекции и итераторы

Все коллекции в Python тоже имеют определенный интерфейс, который описывается специальными методами. Чтобы создать собственный тип коллекций, нужно реализовать ряд специальных методов.

Давайте создадим свой класс, описывающий массив битов. Для хранения данных будем использовать обычный `int`, а получать нужные элементы будем с помощью битовых операций:

```
class Bitarray:
    def __init__(self, length=0, init_value=0):
        self._value = init_value
        self._length = length
    def __repr__(self):
        # Используем len(self), поскольку есть реализованный
        # метод __len__
        return f'Bitarray({self._value:0{len(self)}b})'
    def __eq__(self, other):
        # сначала выполняем более дешевую проверку len
        return len(self) == len(other) and self._value ==
        other._value
    def __len__(self):
        return self._length
```

```
>>> a = Bitarray(32)
>>> a
Bitarray (00000000000000000000000000000000)

>>> len(a)
32
```

Метод `__getitem__` позволяет обратиться к определенному элементу с помощью квадратных скобок:

```
class Bitarray:
      …
      def __getitem__(self, index):
         # Необходимо проверить выход за допустимые пределы
         if index >= len(self) or index < -len(self):
           raise IndexError('Index out of range')
         index %= len(self)
         return (self._value & (1 << index)) >> index

a = Bitarray(4, 0b1100)
for i in range(4):
      print(i, a[i])
```

Написанного метода уже достаточно, чтобы использовать созданный объект в цикле напрямую:

```
a = Bitarray(4, 0b1100)
for x in a:
      print(x, end=' ')
      # 0 0 1 1
```

С помощью `__getitem__` мы также можем превратить наш `Bitarray` в любую другую коллекцию:

```
>>> a = Bitarray(4, 0b1100)
>>> list(a)
```

```
[0, 0, 1, 1]

>>> tuple(a)
(0, 0, 1, 1)

>>> set(a)
{0, 1}
```

Или можем использовать распаковку:

```
>>> print(*a)
1 1 0 0

>>> first, &others = a
>>> first
1

>>> first
[1, 0, 0]
```

Метод `__setitem__` позволяет реализовать изменения в нашей коллекции. Метод принимает еще два аргумента помимо `self` — `index` и `value` (назвать можно и по-другому).

Работает следующим образом:

```
# Мы вызываем индексом элемент и записываем в него свое значение
>>> a = Bitarray(10)
>>> a
Bitarray(0000000000)

>>> a[5] = 1
>>> a
Bitarray(0000100000)
```

Как обратиться к элементу в цикле?

Метод `__iter__` — это специальный метод, который возвращает особый объект-итератор для использования в цикле `for`. Если метод `__iter__` не реализован, но есть `__getitem__`, то цикл будет перебирать индексы от 0 и далее, пока не получит `IndexError`.

Помимо этого метода мы можем создать и собственный **итератор** — объект, который реализует метод `__next__`. Цикл `for` вызывает этот метод, чтобы получить следующий элемент, пока не получит специальное исключение `StopIteration`. Реализация собственного итератора не всегда имеет смысл, но может пригодиться, если все элементы коллекции не хранятся в памяти, а каким-либо образом вычисляются.

Иногда итератор реализуют в виде отдельного класса. Получить итератор можно с помощью функции `iter`:

```
>>> range(10)
range(0, 10)

>>> iter(range(10))
<range_iterator object at 0x7fc4ac3c8ff0>
```

6. Контекстные менеджеры

Вы уже знакомы с контекстным менеджером на примере работы с файлом:

```
with open('filename') as file:
    …
    # Работаем с файлом
    …
# Файл закрыт автоматически
```

Контекстный менеджер — это объект, который можно подставить в блок `with`. Его суть в том, что мы можем выполнить некоторое действие ПЕРЕД и некоторое действие ПОСЛЕ определенного блока кода.

Контекстный менеджер – это объект, который реализует методы **`__enter__`** и **`__exit__`**.

В качестве примера рассмотрим контекстный менеджер, который замеряет время выполнения модуля для понимания производительности кода:

```
from time import perf_counter
```

```
class timeit:
    def __init__ (self, process_name):
        self.name = process_name

    def __enter__ (self):
        self.start = perf_counter()
        return self

    def __exit__ (self, exp_type, exc_val, exc_tb):
        if exc_type is None: # Блок кода завершился без ошибок
            print(f'{self.name}: {perf_counter() -
            self.start:0.3f} sec')

with timeit('Возведение в квадрат миллиона чисел'):
    # вызван метод __enter__
    for i in range(1000000):
        i = i ** 2
    # вызван метод __exit__
```

```
Возведение в квадрат миллиона чисел: 0.459 сек.
```

С помощью этой логики вы можете реализовывать собственные контекстные менеджеры.

7. Инструкция `import`

Для подключения модуля используется инструкция `import`. Модуль подключается путем указания после инструкции имени модуля:

```
import math
```

После этого модуль становится доступен по своему имени. Чтобы использовать функцию из модуля, ее записывают после имени модуля через точку:

```
math.sin(math.pi / 6)
```

Система модулей, как и все в Python, реализована как система особых специфических объектов. В этом можно убедиться, вызвав функцию `dir` для модуля:

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__',
 '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos',
 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp',
 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp',
 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod',
 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan',
 'tanh', 'tau', 'trunc', 'ulp']
```

Функция `dir` покажет все свойства объекта, которые в данном случае являются функциями модуля. Так можно сразу в интерпретаторе посмотреть имеющиеся в модуле функции.

Если из модуля нужны только несколько функций, можно использовать другую форму импорта:

```
from math import sin, cos, pi
cos(pi / 3)
```

С помощью оператора `as` можно импортировать модуль или отдельную функцию под другим именем:

```
from math import log as ln
```

Лучше не злоупотреблять оператором `as` за исключением некоторых случаев. Например, есть общепринятые сокращения — библиотеку `numpy` принято подключать следующим образом:

```
import numpy as np
```

Еще один случай — слишком длинное название подмодуля:

```
import aiogram.dispatcher.fsm.storage.redis as redis
```

Еще одна допустимая, но крайне опасная конструкция:

```
from math import *
```

Конструкция позволяет импортировать все имена, которые содержатся в данном модуле. Все импортируемые имена будут доступны по их обычному имени в глобальном пространстве имен:

```
from math import *
```

```
>>> asin(sin(pi/3) * cos(pi/6))
0.848062078981481
>>> log2(256)
8.0
>>> gcd(45, 120)
15
```

Это крайне опасно, поскольку Python позволяет переопределять любые имена. Например, мы хотим работать с файлами в формате bz2, для чего подключаем соответствующий модуль. Далее мы хотим записать что-либо в файл и вызываем функцию `open`, думая, что это встроенная функция. Но при попытке это сделать мы получаем ошибку, поскольку функция `open` была подменена функцией с таким же названием в модуле `bz2`:

```
>>> from bz2 import *
...
>>> f = open('hello.txt', 'w')  # Мы думаем, что это
                                # встроенная функция open
>>> f.write('Hello!')
...
TypeError: memoryview: a bytes-like object is required, not 'str'
>>> f
<bz2.BZ2File object at 0x7f5e7cf10610>
```

Если мы импортируем несколько модулей, и все они содержат одно и то же название, то когда мы вызываем функцию, мы даже не знаем, какая именно функция была вызвана:

```
from bz2 import *
from tarfile import *
from gzip import *
from lzma import *

f = open('filename')
# bz2.open?
# tarfile.open?
# gzip.open?
# lzma.open?
```

В рассматриваемом примере будет вызвана функция из последнего импортируемого модуля, поскольку каждая последующая инструкция импорта перезаписывает все предыдущие.

Важно! Приведенный пример слишком очевидно демонстрирует опасность использования инструкции `import` со звездочкой. На практике опасность импорта может быть не так очевидна, поскольку вы не всегда знаете, какие имена содержатся в модуле.

В реальных программах инструкцию `import` со звездочкой лучше не использовать. Пример допустимого применения подобной конструкции — работа в интерактивной консоли, когда нужно быстро опробовать использование функций модуля. Также звездочку можно использовать, если вы точно будете использовать один модуль.

8. Модули стандартной библиотеки

Полная информация о модулях стандартной библиотеки содержится в документации.

Математические модули

Модуль `math` содержит математические функции, а `cmath` — те же функции для комплексных чисел.

Кроме того Python содержит модуль `decimal`, дающий возможность работать с десятичными дробями с фиксированной точностью. Проблема формата `float` (числа с плавающей точкой) — имеет ограниченную точность, которая описывается в двоичной системе счисления, что может быть несколько странно:

```
>>> 0.1 * 3
0.30000000000000004
```

Причина — происходит округление, но в двоичной системе счисления. Причем это не проблема Python, а проблема компьютерной арифметики в целом.

Модуль `decimal` позволяет проводить точные вычисления с десятичными дробями, что может быть уместно и нужно в задачах, где нужна принципиальная точность:

```
>>> from decimal import Decimal
>>> x = Decimal('0.1')
>>> print(x * 3)
0.3
```

Точность настраивается. Но при использовании данного модуля происходит снижение производительности.

Модуль `fractions` умеет работать с обыкновенными дробями. Пример работы модуля:

```
>>> from fractions import Fraction
>>> x = Fraction('1/3')
>>> y = Fraction('1/4')
>>> print(x + y)
7/12
```

Модуль `random` предназначен для работы со случайными числами. Числа являются псевдослучайными, и использовать их для генерирования паролей или секретных

токенов — плохое решение. Модуль `random` содержит разные способы генерирования случайных чисел:

```
>>> import random
>>> random.randint(1, 9)
5

>>> random.uniform(1, 9)
6.458518913632032

>>> random.gauss(100, 1)
101.38539928863737

>>> random.choice('red green blue white black'.split())
'white'

>>> a = list(range(10))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> random.shuffle(a)
>>> a
[0, 5, 4, 7, 2, 8, 3, 6, 9, 1]
```

Может быть полезным модуль `statistics`, который содержит основные статистические функции. Они позволяют вычислять среднее, среднее отклонение, медиану и прочее:

```
>>> import statistics
>>> a = [random.gauss(100, 1) for i in range(1000)]
>>> statistics.mean(a)
100.04625295581766

>>> statistics.stdev(a)
0.9944075990623776

>>> statistics.median(a)
100.05381703876445

>>> statistics.quantiles(a)
[99.36124977075855, 100.05381703876445, 100.69974208859895]
```

Работа с текстом

Модуль `textwrap` содержит функции для выравнивания текста. Например, когда необходимо вывести длинное предложение на экран, но сделать это не в одну строку, а отформатировав по ширине. Для этого используется функция `wrap`:

```
>>> import textwrap
```

```
>>> print('\n'.join(textwrap.wrap(crime_and_punishment)))
В начале июля, в чрезвычайно жаркое время, под вечер один молодой
человек вышел из своей каморки, которую нанимал от жильцов в С-м
переулке, на улицу и медленно, как бы в нерешимости, отправился к
К-ну мосту.

>>> print('\n'.join(textwrap.wrap(crime_and_punishment, width=40,
...                       initial_indent='    ', subsequent_indent='
')))
    В начале июля, в чрезвычайно жаркое
 время, под вечер один молодой человек
 вышел из своей каморки, которую нанимал
 от жильцов в С-м переулке, на улицу и
 медленно, как бы в нерешимости,
 отправился к К-ну мосту.
```

Функция `shorten` позволяет вывести начало текста:

```
>>> import textwrap

>>> print(textwrap.shorten(crime_and_punishment, 60))
В начале июля, в чрезвычайно жаркое время, под вечер [...]
```

Еще одна удобная функция `dedent` удаляет лишний отступ у строки:

```
import textwrap
if 2*2 == 4:
    print(textwrap.dedent('''
        Иногда бывает нужно написать многострочный текст,
        но если его написать без отступов, это будет
        выглядеть некрасиво в коде, а если добавить отступы,
        будет выглядеть странно при выводе на экран.
        Функция dedent решает эту проблему.
    ''').strip())
```

Функция `repr` из модуля `reprlib` может быть полезна для более безопасного отображения объектов на экране. Например, мы хотим распечатать список из одного миллиона чисел. Но это может быть слишком много, и может привести к подвисанию консоли. Функция `repr` «обрежет» данные и покажет их с помощью многоточия:

```
>>> import reprlib
>>> a = [0] * 1_000_000
>>> print(reprlib.repr(a))
[0, 0, 0, 0, 0, 0, ...]
```

Еще одна интересная функция — `pprint` — красивая распечатка (pretty print), которая позволяет распечатывать объекты в более читабельном виде. Например, функция распечатает двумерный список:

```
>>> from pprint import pprint
>>> a = [[0] * 10 for i in range(10)]
```

```
>>> 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, 0, 0, 0, 0]]
```

Работа с датой и временем

Модуль `time` содержит низкоуровневые функции работы со временем, модуль `datetime` — удобные классы, представляющие даты и время.

В примере функция `time` из модуля `time` сообщает текущее время, что не очень удобно:

```
>>> import time
>>> time.time()
1667648379.2910438
```

Модуль `datetime` — сообщает время более удобно:

```
>>> from datetime import datetime, timedelta
>>> datetime.now()
datetime.datetime(2022, 11, 5, 14, 39, 52, 583388)
```

Объекты `datetime` позволяют выполнять над ними различные операции, например, вычитание:

```
>>> datetime(2023, 1, 1) - datetime.now()
datetime.timedelta(days=56, seconds=33594, microseconds=17083)
```

Дополнительные инструменты

Модуль `collections` содержит классы-контейнеры. Например, класс `Counter`, который позволяет посчитать, сколько раз тот или иной элемент встречается в объекте:

```
>>> from collections import Counter
>>> text = open('Анна Каренина.txt').read()
>>> letters = Counter(text)
>>> letters.most_common(4)
[(' ', 280124), ('о', 151294), ('е', 116880), ('а', 107428)]
```

Модуль `enum` позволяет описать класс, в котором описано несколько вариантов чего-либо. В примере модуль используется для создания класса, описывающего сложность задачи:

```
from enum import IntEnum

class Difficulty(IntEnum):
    UNDEFINED = 0
    PRIOR = 1
    EASY = 2
    NORMAL = 3
    HARD = 4
    INSANE = 5

...
task.difficulty = Difficulty.EASY
```

Модуль `itertools` предоставляет набор удобных итераторов. В примере показан итератор `combinations`, который выдает все возможные сочетания из некоторого множества:

```
>>> from itertools import combinations
>>> for x in combinations('abcde', 2):
...     print(x)
...
('a', 'b')
('a', 'c')
('a', 'd')
('a', 'e')
('b', 'c')
('b', 'd')
('b', 'e')
('c', 'd')
('c', 'e')
('d', 'e')
```

Модуль `contextlib` предоставляет удобные контекстные менеджеры. Контекстный менеджер `suppress` позволяет подавить какое-либо исключение:

```
from contextlib import suppress

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

Еще один пример из документации. Функция `redirect_stdout` позволяет перенаправить то, что распечатывается, на экран в некоторый файл:

```
import io
from contextlib import redirect_stdout
```

```
with redirect_stdout(io.StringIO()) as f:
    help(pow)
s = f.getvalue()
```

Модуль `sys` полезен для получения и настройки параметров интерпретатора Python:

```
>>> import sys
>>> print(sys.version)
3.10.5 (main, Jun 27 2022, 12:54:03) [GCC 9.4.0]

>>> print(sys.platform)
linux

>>> sys.getsizeof([0] * 1000)
8056
```

Рассмотрим подробнее пример. В модуле `sys` объявлена функция `displayhook`. Она вызывается, когда мы что-то пишем в интерпретаторе и результат операции должен быть выведен на экран. Функцию `displayhook` мы можем заменить — например, на функцию `pprint.И`ли использовать `reprlib.repr`, написав дополнительно лямбда-функцию. Если мы хотим восстановить `displayhook`, который был по умолчанию, он всегда хранится в атрибуте `sys.__displayhook__`:

```
>>> a = [[0] * 5 for i in range(5)]
>>> 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]]

>>> import sys
>>> import pprint
>>> sys.displayhook = 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]]

>>> import reprlib
>>> sys.displayhook = lambda x: print(reprlib.repr(x))
>>> a = [0] * 1000
>>> a
[0, 0, 0, 0, 0, 0, ...]
>>> sys.displayhook = sys.__displayhook__
```

Модуль `os` позволяет взаимодействовать с операционной и файловой системой. Можно узнать, в каком каталоге мы находимся, посмотреть список файлов, удалить, скопировать, переименовать файл и так далее. Также мы можем получить доступ к

переменным окружения, узнать размер терминала, узнать подробную информацию об операционной системе:

```
>>> import os
>>> os.getcwd()
'/home/anatoly/test'

>>> sorted(os.listdir())
['test_1.txt', 'test_2.txt', 'test_3.txt', 'test_4.txt',
'test_5.txt']

>>> os.remove('test_2.txt')
>>> sorted(os.listdir())
['test_1.txt', 'test_3.txt', 'test_4.txt', 'test_5.txt']

>>> os.getenv('HOME')
'/home/anatoly'

>>> os.get_terminal_size()
os.terminal_size(columns=120, lines=30)
```

Есть ряд модулей для работы с различными форматами файлов:

- `json`
- `csv`
- `configparser`
- `tomllib`
- `html`
- `xml`
- `tarfile`
- `bz2`
- `gzib`
- `lzma`
- `zlib`
- `zipfile`

Несколько недокументированных модулей:

- `import this` распечатывает «Дзен Python»;
- `import antigravity` — просто небольшая шутка (попробуйте сами).

9. Создание своего модуля на Python

Проект в Python редко состоит из одного файла. Те файлы, которые мы создаем в нашем проекте — это такие же модули, как и модули стандартной библиотеки.

Любой модуль в Python — это файл с расширением .py, доступный интерпретатору в текущей директории и имеющий «нормальное» название, под которым его можно подключить.

Рассмотрим простую структуру. В проекте содержатся два файла:

- `main.py` — главный файл проекта, пока пустой;
- `hello.py` — распечатывает фразу «Hello!».

С помощью инструкции `import` мы можем импортировать содержимое файла `hello.py` в файл `main.py`, записав:

```
import hello
```

Инструкция `import` просто исполняет файл, если он содержит какой-либо исполняемый код. Пусть файл `hello.py` содержит следующий код:

```
print('Hello!')
a = 1
def f(x):
    return x * 2
```

Если файл содержит какие-либо переменные и функции, то подключив его, мы можем посмотреть на объявленные переменные как на атрибуты модуля и вызвать функцию из модуля под ее названием:

```
print(hello.a)
print(hello.f(2))
```

Мы можем также переопределять переменные, объявленные в модуле:

```
hello.a = 10
```

То есть, модуль ведет себя как полноценный объект.

Важно! Переопределением переменных лучше не злоупотреблять. Чаще всего модуль воспринимается как нечто статичное.

После входа в интерактивный режим можно посмотреть, что представляет собой модуль:

```
>>> import hello
Hello
>>> hello
<module 'hello' from
'/home/anatoly/yadisk/students/mipt/course/09/modules/hello.py'>
>>> type(hello)
<class 'module'>
```

```
>>> dir(hello)
['__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'a', 'f']
```

Рассмотрим специальную переменную `__name__`. Запись ниже означает, что при запуске данного файла непосредственно нужно выполнить некоторый код. Но если файл импортируется, этот код выполнять не нужно:

```
if __name__ == '__main__':
    print('Hello!')
```

В примере фраза `'Hello!'` будет напечатана только если будет запущен непосредственно файл `hello.py`. Чаще всего в данный блок помещают тесты, чтобы проверить правильность работы модуля.

После импорта модуля в каталоге проекта дополнительно появляется папка `__pycache__`. При импорте Python компилирует код модуля. Предполагается, что модули меняют не так уж и часто, поэтому если ничего не изменилось, то можно не выполнять заново компиляцию.

Если модуль будет назван недопустимым образом, мы не сможем его импортировать. Например, модуль будет назван числом, содержать пробелы или ключевые слова языка.

Предположим, что импортируемый модуль содержит синтаксическую или другую ошибку. Тогда при исполнении модуля мы получим эту самую ошибку.

Ошибка возникнет и при импорте несуществующего модуля — ошибка `ModuleNotFoundError`, являющаяся разновидностью ошибки `ImportError`. Эта же ошибка возникнет при попытке импортировать из модуля несуществующую функцию.

Важно! Не следует называть файл с собственным модулем именем модуля стандартной библиотеки. Python ищет импортируемый файл в текущей папке, а затем в стандартной библиотеке. Поэтому подобное может привести к «поломке» кода.

Помимо модулей Python поддерживает так называемые пакеты. **Пакет** представляет собой папку с файлами. Чтобы импортировать модуль из пакета, нужно воспользоваться инструкцией `from`:

```
from имя_папки import имя_модуля
```

Или записать через точку:

```
import имя_папки.имя_модуля
```

Запись через точку используется при построении сложных проектов. Здесь разработчики Python взяли за основу файловую систему при проектировании иерархии модулей — по сути, мы прописываем путь к нужному модулю через точку. Но при импорте каталога файлы, подпапки и подфайлы, которые в нем лежат, автоматически не импортируются. Импорт необходимо прописать явно.

Иногда необходимо, чтобы при импорте подключался модуль:

```
>>> имя_папки.имя_модуля
```

В этом случае необходимо в папке создать специальный файл `__init__.py` и прописать в нем, что необходимо импортировать.

Чтобы заново выполнить импорт, необходимо перезапустить консоль, так как Python не выполняет импорт одного и того же модуля дважды.

С помощью такой системы можно навести порядок в своем проекте, сделав структуру проекта наглядной и понятной.

10. Установка внешних библиотек Python

Самый простой способ установки дополнительных пакетов — использовать утилиту `pip`. Рассмотрим пример установки библиотеки `tqdm` — библиотеки для создания прогрессбар:

```
$ pip install tqdm
```

Чтобы воспользоваться библиотекой, запустим в терминале Python:

```
$ python
```

Подключим одноименную функцию из установленной библиотеки:

```
$ from tqdm import tqdm
```

Чтобы продемонстрировать работу с библиотекой, сымитируем длительный процесс с помощью функции задержки:

```
>>> import time
>>> for in range(20):
...     time.sleep(0.1)
```

Чтобы создать прогрессбар, обернем итератор, по которому мы идем в цикле, в функцию `tqdm`:

```
>>> for in tqdm(range(20)):
...     time.sleep(0.1)
```

После того как были установлены какие-либо пакеты, иногда требуется зафиксировать это состояние. Утилита `pip` позволяет посмотреть, что было установлено. Команда

```
$ pip list
```

покажет список установленных пакетов в удобном виде.

Команда

```
$ pip freeze
```

выведет в более «техническом» виде, понятном для утилиты `pip`.

Используя перенаправление

```
$ pip freeze > requirements.txt
```

мы можем записать вывод списка утилит в файл.

Далее мы можем воспользоваться файлом `requirements.txt` при запуске на другом компьютере. Получив такую команду, система считывает и устанавливает разом все зависимости:

```
$ pip install -r requirements.txt
```

Важно! Файл может называться и по-другому, `requirements.txt` — общепринятое название для данного файла.

Рассмотрим утилиту `pipenv`, которая является оберткой для утилит `pip` и `venv`. Эта утилита одновременно создает виртуальное окружение и устанавливает зависимости:

```
$ pipenv install tqdm
```

Утилита понимает, что отсутствует виртуальное окружение. Затем сначала создает его в текущем каталоге, а затем устанавливает библиотеку в виртуальное окружение. При использовании `pipenv` виртуальное окружение сохраняется не в этой же папке, а в отдельном каталоге. Далее можно запустить Python из созданного виртуального окружения:

```
$ pipenv run python
```

Чтобы навсегда перейти в созданное виртуальное окружение, нужно использовать команду:

```
$ pipenv shell
```

После этого происходит активация виртуального окружения. Выйти из виртуального окружения можно с помощью закрытия сеанса. В Linux и macOS это выполняется с помощью клавиш `Ctrl+D`, в Windows — `Ctrl+Z` и `Enter`.

Рассмотрим утилиту `poetry`, которая позволяет более сложно управлять проектом. Команда создания проекта (`project` — имя проекта):

```
$ poetry new project
```

При первом запуске утилита спросит вас различные настройки.

В папке проекта созданы несколько папок:

- `project` — будут созданы исходные коды;
- `tests` — будут созданы тесты.

Также созданы файлы `README.md` и `pyproject.toml` — они содержат информацию о проекте в стандартном виде.

Чтобы добавить какую-либо зависимость, нужно дать команду:

```
$ poetry add tqdm
```

Если виртуальное окружение отсутствовало, оно создается и далее устанавливается дополнительный пакет. В файл `pyproject.toml` добавляется информация о созданной зависимости.