



Лекция 6. Модули

Погружение в Python



Оглавление

На этой лекции мы	2
Дополнительные материалы к лекции	3
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	3
Подробный текст лекции. Введение	4
1. Ещё раз про import	4
Переменная sys.path	5
Антипримеры импорта	5
Использование from и as	6
Плохой import * (импорт звёздочка)	7
Переменная __all__	9
Задание	10
2. Виды модулей	10
Встроенные модули	11
Свои модули	12
Пишем свой модуль: __name__ == '__main__'	13
Разбор плохого импорта	14
Создание пакетов и их импорт	15
Разница между модулем и пакетом	16
Варианты импорта	17
Задание	18
3. Некоторые модули “из коробки”	19
Модуль sys	19
Запуск скрипта с параметрами	19
Модуль random	21
Задание	22

На этой лекции мы

1. Разберём работу с модулями в Python
2. Изучим особенности импорта объектов в проект
3. Узнаем о встроенных модулях и возможностях по созданию своих модулей и пакетов
4. Разберём модуль random отвечающий за генерацию случайных чисел

Дополнительные материалы к лекции

Стандартная библиотека Python. Документация.

<https://docs.python.org/3/library/index.html>

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрали решения задач в одну строку
2. Изучили итераторы и особенности их работы
3. Узнали о генераторных выражениях и генераторах списков, словарей, множеств
4. Разобрали создание собственных функций генераторов.

Термины лекции

- **Модуль** — это файл, содержащий определения и операторы Python. Во время исполнения же модуль представлен соответствующим объектом, атрибутами

которого являются объявления, присутствующие в файле и объекты, импортированные в этот модуль откуда-либо.

- **Пакет** — это набор взаимосвязанных модулей (при этом стоит уточнить, что сам пакет тоже является модулем), предназначенных для решения задач определенного класса некоторой предметной области. Пакеты — это способ структурирования пространства имен модулей Python с помощью «точечных имен модулей». Пакет представляет собой папку, в которой содержатся модули и другие пакеты и обязательный файл `__init__.py`, отвечающий за инициализацию пакета.

Подробный текст лекции

Введение

Прежде чем начать, немного о названиях. Когда вы пишете код на Python и сохраняете его в файл с расширением `py`, получаем программу на Python. Другие названия для программы в файле — скрипт, Python скрипт. Помимо скрипта вы можете встретить термины модуль и пакет.

Модули можно импортировать в другие файлы для использования написанного ранее кода в другом проекте. По сути любой Python скрипт является модулем. Для разработчика на Python можно утверждать, что программа, скрипт и модуль — синонимы.

Пакет представляет из себя набор модулей. В Python любой пакет одновременно является модулем. Но не каждый модуль считается пакетом. Подробнее о модулях и пакетах далее в лекции.

1. Ещё раз про `import`

Мы уже использовали зарезервированное слово `import` для добавление в собственный код расширенного функционала. Разберём `import` подробнее.



PEP-8! Строки импорта рекомендуется писать в самом начале файла, оставляя 1-2 пустые строки после него.

```
import sys

print(sys)
print(sys.builtin_module_names)
print(*sys.path, sep='\n')
```

Мы импортировали модуль `sys` из стандартной библиотеки Python. Механизм импорта выполняет поиск указанного имени (в нашем примере — “`sys`”) в нескольких местах. В первую очередь проверяется список встроенных модулей, который хранится в `sys.builtin_module_names`. Если имя модуля не найдено, проверка ведётся в каталогах файловой системы, которые перечислены в `sys.path`.



Важно! Обычно (но не всегда) имя модуля заканчивается расширением `.py`. При импорте расширение не указывается.

В результате импорта имя `sys` было добавлено в текущую, глобальную область видимости. Для того, чтобы получить доступ к переменным, функциям, классам и т.п. содержимому модуля используется точечная нотация: `имя_модуля<точка>имя_объекта`.

Обратиться к объекту внутри модуля напрямую при “обычном” импорте не получится. Подобный подход защищает от конфликта имён. Если и в вашем файле и в импортируемом модуле есть функция `func()`, конфликта имён не будет. Свою функцию вызываете как `func()`, а функцию из модуля как `module_name.func()`

- **Переменная `sys.path`**

Содержимое переменной `sys.path` формируется динамически. В качестве первого адреса указывается путь до основного файла. Например если вы используете Unix подобную ОС и ваш скрипт расположен по адресу `/home/user/project`, именно этот путь будет первым в списке поиска модулей.

Далее в `sys.path` перечислены пути из `PYTHONPATH` и пути, указанные при установке Python и создании виртуального окружения. Таким образом Python ищет импортируемый модуль практически во всех местах, где этот модуль мог быть установлен.



Важно! Если создать собственный файл с именем аналогичным имени модуля, Python импортирует ваш файл, а не модуль. Строго не рекомендуется использовать для своих файлов имена встроенных модулей. В редких исключительных ситуациях стоит добавлять символ подчёркивания в конце имени, чтобы избежать двойного именования.

- **Антипримеры импорта**

Взгляните на антипример. Мы создали файл `random.py` со следующим кодом.

```
def randint(*args):  
    return 'Не то, что вы искали!'
```

Импортируем модуль `random` в основном файле проекта и попробуем сгенерировать случайное число от 1 до 6.

```
import random  
  
print(random.randint(1, 6))
```

В результате работы получим “Не то, что вы искали!”. Подробнее о модуле для генерации случайных чисел поговорим далее в этом уроке.



PEP-8! При импорте нескольких модулей каждый указывается с новой строки.

Правильно:

```
import sys  
import random
```

Неправильно:

```
import sys, random
```

Использование `from` и `as`

Помимо обычного импорта можно использовать более подробную форму записи. Зарезервированное слово `from` указывает на имя модуля или пакета, далее `import` и имена импортируемых объектов.

```
from sys import builtin_module_names, path  
  
print(builtin_module_names)  
print(*path, sep='\n')
```

Теперь при обращении к импортированным объектам не нужно указывать имя модуля. Мы явно добавили их в наш код, включили имена в область видимости.



PEP-8! Конструкция `from import` допускает перечисление импортируемых имён объектов через запятую в одной строке. После `from` всегда указывается один модуль.

Кроме выборочного импорта можно создавать псевдонимы для объектов через зарезервированное слово `as`. При этом доступ к объекту будет возможен только через псевдоним. Один объект — одно имя.

```
import random as rnd
from sys import builtin_module_names as bmn, path as p

print(bmn)
print(*p, sep='\n')
print(rnd.randint(1, 6))
print(path)    # NameError: name 'path' is not defined
print(sys.path) # NameError: name 'sys' is not defined
```

В первой строке импортировали модуль `random` и присвоили ему имя `rnd` внутри текущей области видимости. Во второй строке импортировали переменную `builtin_module_names` под именем `bmn` и переменную `path` под именем `p`. Последние две строки вызывают ошибку имени. Мы не можем обратиться к переменной `path`, потому что дали ей другое имя — `p`. И обращение к модулю `sys` не работает, ведь мы его не импортировали. Только объекты из модуля.



Важно! Не стоит давать переменным короткие понятные лишь вам имена. Код должен легко читаться другими разработчиками. Исключения — общепризнанные сокращения, например `import numpy as np`.

Плохой `import *` (импорт звёздочка)

Ещё один вариант импорта: `from имя_модуля import *`

Подобная запись импортирует из модуля все глобальные объекты за исключением тех, чьи имена начинаются с символа подчёркивания. Рассмотрим на примере.

- Файл `super_module.py`

```

from random import randint

SIZE = 100
_secret = 'qwerty'
__top_secret = '1q2w3e4r5t6y'


def func(a: int, b: int) -> str:
    z = f'В диапазоне от {a} до {b} получили {randint(a, b)}'
    return z

result = func(1, 6)

```

В модуле есть следующие объекты:

- глобальная функция randint
- глобальная константа SIZE
- глобальная защищенная переменная _secret
- глобальная приватная переменная __top_secret
- глобальная функция func
- локальные параметры функции a и b
- локальная переменная функции z
- глобальная переменная result

 **Внимание!** Если название объекта (переменной, функции и т.п.) начинается с символа подчёркивания, объект становится защищённым. Если имя начинается с двух подчёркиваний, объект становится приватным. Объекты без подчёркивания в начале имени — публичные. Подробнее разберём на лекциях по ООП.

Импортируем модуль в основной файл программы через звёздочку и попробуем выполнить несколько операций.

- **Файл main.py**

```

from super_module import *

SIZE = 49.5

print(f'{SIZE = }\n{result = }')
print(f'{z = }')    # NameError: name 'z' is not defined
print(f'{_secret = }')    # NameError: name '_secret' is not defined
print(f'{func(100, 200) = }\n{randint(10, 20) = }')

```



```
def func(a: int, b: int) -> int:
    return a + b

print(f'{func(100, 200) = }')
```

Первая строка импортирует в файл все глобальные публичные объекты. Далее мы определяем константу SIZE. В этот момент значение константы из модуля затирается новым значением. Возник конфликт имён и Python разрешил его в пользу нового значения константы. При этом содержимое переменной result берётся из модуля super_module, т.к. других определений переменной нет в файле. При попытке обратиться к локальной и защищённой переменным получаем ошибки. Они не были импортированы “звёздочкой”.

Далее мы вызываем функции func и randint. Они верно отработывают код, т.к. обе были импортированы из внешнего модуля.

В финале создаём свою функцию func. Возникает очередной конфликт имён, который Python разрешает в пользу новой функции. В результате вызов func() после её определения возвращает совсем другой результат, чем вызовом ранее.

Промежуточный итог. Использование стиля from module import * зачастую приводит к неожиданным результатам, затрудняет отладку кода и мешает верному пониманию работы программы. Использовать подобный приём стоит в редких особенных случаях. Кроме того импорт всех доступных объектов может значительно замедлить работы программы, если таких объектов очень много.

Переменная `__all__`

При необходимости разработчик модуля может явно указать какие объекты нужно импортировать при использовании стиля from module import *. Для этого используется магическая переменная `__all__` (два нижних подчёркивания до, слово all и два нижних подчёркивания после). Изменим код модуля super_module.py, добавив строку с `__all__`

Файл super_module.py

```
from random import randint
```

```

__all__ = ['func', '_secret']

SIZE = 100
_secret = 'qwerty'
_top_secret = '1q2w3e4r5t6y'

def func(a: int, b: int) -> str:
    z = f'В диапазоне от {a} до {b} получили {randint(a, b)}'
    return z

result = func(1, 6)

```

Переменной `__all__` присваивается список имён объектов, заключённых в кавычки, т.е. `str` типа. В основной модуль попадут только указанные в списке имена, независимо от того являются они публичными, защищёнными или приватными. При этом объект должен быть глобальным. Если указать в списке имя локального объекта, например переменную `z` — локальную переменную функции `func`, получим ошибку.

Список `__all__` в приведённом примере используется для формирования списка импортируемых объектов модуля. Кроме этого `__all__` применяется для импорта модулей из пакета. Рассмотрим вариант импорта модулей из пакета далее на лекции.

Задание

Перед вами пример кода. Какие переменные будут доступны после импорта для работы в основном файле? У вас три минуты.

```

import sys
from random import *
from super_module import func as f

```

2. Виды модулей

Попробуем добавить некоторую системность в модули. В Python есть:

- встроенные модули,
- установленные внешние модули,
- модули, созданные разработчиком, свои.

Кроме того каждый вид модулей может быть внутри пакета, сгруппированной коллекции модулей. Разберём подробнее.

Встроенные модули

Один из плюсов языка Python — его батарейки. Так принято называть стандартную библиотеку языка. По сути библиотека представляет обширный набор пакетов и модулей для решения широкого спектра задач.

Некоторые модули стандартной библиотеки мы уже использовали. С другими будем знакомиться в рамках последующих уроков. Финальный урок курса целиком посвятим разговору о стандартной библиотеке. А пока определимся с несколькими вещами:

1. Стандартная библиотека устанавливается вместе с интерпретатором. Дополнительные манипуляции по установке не требуются. Всё работает “из коробки”.¹
2. Для использования модуля стандартной библиотеки достаточно его импортировать в ваш код.
3. Большинство частых задач легко решаются средствами стандартной библиотеки. Достаточно обратиться к [справке](#) и найти нужный модуль.
4. Некоторые модули стандартной библиотеки разрабатывались настолько давно, что не отвечают современным требованиям решения задач. В таком случае на помощь приходят внешние решения. И наоборот. Каждое обновление Python вносит улучшения в библиотеку, зачастую более эффективные, чем внешние решения.

¹ Установщики Python для платформы Windows обычно включают в себя всю стандартную библиотеку, а также множество дополнительных компонентов. Для Unix-подобных операционных систем Python обычно предоставляется в виде набора пакетов, поэтому может потребоваться использование инструментов упаковки, поставляемых с операционной системой, для получения некоторых или всех дополнительных компонентов.



PEP-8! Импорт модулей стандартной библиотеки пишется в начале файла, до импорта внешних и своих модулей. После импорта оставляют пустую строку, даже если далее идёт импорт модулей не из библиотеки.

Свои модули

А теперь пара слов о том как получить свой модуль. В вашей любимой IDE создаём файл с расширением `py`, пишем в нём код и сохраняем результат. Готово! Вы создали свой модуль. Ранее на лекции мы создали файл `super_module.py` и импортировали его в `main.py`, т.е. работали с файлом как с модулем.

Обычно модуль должен решать одну задачу или несколько однотипных задач.

Стандартная структура любого модуля следующая:

- документация по модулю в виде многострочного комментария (три пары двойных кавычек),
- импорт необходимых пакетов, модулей, классов, функций и т.п. объектов,
- определение констант уровня модуля,
- создание классов модуля при ООП подходе,
- создание функций модуля,
- определение переменных модуля,
- покрытие тестами, если оно не вынесено в отдельный пакет,
- `main` код.

В зависимости от решаемой задачи некоторые пункты могут отсутствовать.

Учебный пример модуля ниже:

Файл `base_math.py`

```
"""Four basic mathematical operations.

Addition, subtraction, multiplication and division as functions.
"""

_START_SUM = 0
_START_MULT = 1
_BEGINNING = 0
_CONTINUATION = 1


def add(*args):
    res = _START_SUM
    for item in args:
        res += item
```

```

    return res

def sub(*args):
    res = args[_BEGINNING]
    for item in args[_CONTINUATION:]:
        res -= item
    return res

def mul(*args):
    res = _START_MULT
    for item in args:
        res *= item
    return res

def div(*args):
    res = args[_BEGINNING]
    for item in args[_CONTINUATION:]:
        res /= item
    return res

```

Как вы заметили, в файле нет классов, только функции. Так же нам не понадобились переменные уровня модуля. Тесты мы не стали писать, т.к. не добрались до темы тестирования. Осталось добавить main код.

Пишем свой модуль: `__name__ == '__main__'`

Давайте всё же проверим работоспособность кода. Сделаем по 1-2 вызова каждой функции в том же файле и выведем результат на печать. Визуально убедимся, что код работает верно.

```

print(f'{add(2, 4) = }')
print(f'{add(2, 4, 6, 8) = }')
print(f'{sub(10, 2) = }')
print(f'{mul(2, 2, 2, 2, 2) = }')
print(f'{div(-100, 5, -2) = }')

```

Визуальное тестирование прошло успешно. Импортируем модуль в основной код нашего учебного проекта, в файл main.py

```
import base_math

x = base_math.mul    # Плохой приём
y = base_math._START_MULT    # Очень плохой приём
z = base_math.sub(73, 42)
print(x(2, 3))
print(y)
print(z)
```

Запускаем файл и наблюдаем вывод “принтов” из файла base_math раньше наших расчётов в “мейне”. Дело в том, что команда import запускает импортируемый модуль. В результате лишние вызовы функций, а следовательно более медленное выполнение кода. Для решения проблемы необходимо внести правки в файл base_math.py.

```
...
if __name__ == '__main__':
    print(f'{add(2, 4) = }')
...
```

После определения функций, но перед вызовом print добавили логическую проверку. Строки с print после проверки и до конца файла сдвинули на 4 пробела, поместили внутрь блока проверки.

Магическая переменная __name__ содержит полное имя модуля. Это имя используется для уникальной идентификации модуля в системе импорта.

Когда мы запускаем сам модуль base_math, в переменную __name__ попадает имя “__main__”. Оно указывает, что файл запущен, а не импортирован. Никакой разницы в работе модуля не произошло.

Когда модуль импортируется в другой файл, в переменную __name__ попадает его имя, название файла без расширения. import base_math также выполняет код в файле. Присваиваются значения константам, инициализируются функции. Но логическая проверка if __name__ == '__main__': возвращает ложь и дальнейший код пропускается.

Отличной. Добавление “мейн” проверки позволило завершить создание модуля. Теперь запуск main.py работает так, как мы ожидаем.

Разбор плохого импорта

Вернёмся к коду в `main.py` и разберём пару антипримеров.

1. `x = base_math.mul`

Мы передали (не путайте с вызвали) в переменную `x` функцию `mul`. Теперь для умножения можно использовать более короткое имя. Но `x` не является понятным для других разработчиков именем.

Логичнее было бы написать:

```
multiplication = base_math.mul
```

Если же нужно короткое имя, то

```
mul = base_math.mul
```

Но и тут мы сталкиваемся с проблемой. В большом файле может быть трудно заметить присвоение нового имени для функции. Самым логичным вариантом будет использование сочетания `from base_math import mul`. Или `from base_math import mul as mult`, если хотим дать новое имя.

2. `y = base_math._START_MULT`

В отличие от `from module import *` обычный импорт позволяет получить доступ к защищённым переменным. Мы смогли прочитать константу `_START_MULT`. В Python действует следующее правило. Все программисты взрослые люди. Если один из них нарушает соглашения по написанию кода, значит он готов взять на себя ответственность за возможные проблемы.

Важно! Не используйте защищённые и приватные объекты за пределами модуля, в котором они созданы. Особенно если это не ваш модуль. Исключение — прописанное в модуле указание на использование.

Создание пакетов и их импорт

Продолжим нашу идею с учебным математическим модулем. Наш проект развивается и решено добавить целочисленное деление и возведение в степень. А чтобы модуль не разрастался до бесконечности, продвинутую математику будем

хранить в отдельном файле `advanced_math.py`

```
"""Two advanced mathematical operations.

Integer division and exponentiation."""

__all__ = ['div', 'exp']
_BEGINNING = 0
_CONTINUATION = 1

def div(*args):
    res = args[_BEGINNING]
    for item in args[_CONTINUATION:]:
        res //= item
    return res

def exp(*args):
    res = args[_BEGINNING]
    for item in args[_CONTINUATION:]:
        res **= item
    return res

if __name__ == '__main__':
    print(f'{div(42, 4) = }')
    print(f'{exp(2, 4, 6, 8) = }')
```

Самое время объединить два схожих модуля в один пакет. Создаём директорию `mathematical` и переносим в неё оба файла: `base_math.py` и `advanced_math.py`. Далее создаём в каталоге пустой файл `__init__.py`. Пакет `mathematical` готов.

В Python любая директория с файлом `__init__.py` автоматически становится пакетом. При этом полезный функционал содержится в других питоновских файлах, а не в “инит”.

Разница между модулем и пакетом

Пакет — директория с `__init__.py` файлом и другими `py` файлами — модулями. В Python любой пакет является одновременно и модулем. Это означает, что пакет можно импортировать в проект как и модуль. Так же это означает, что пакет может хранить в себе другие пакеты — директории с “инит” файлом. Глубина вложенности

ограничена лишь здравым смыслом.

Пример пакета `sound`, содержащего пакеты `formats`, `effects` и `filters`, содержащие различные модули из официальной документации. Обратите внимание, что в проекте четыре директории и четыре файла `__init__.py`, по одному на пакет.

```
sound/                                Top-level package
  __init__.py                          Initialize the sound package
  formats/                             Subpackage for file format conversions
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
  effects/                             Subpackage for sound effects
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
  filters/                             Subpackage for filters
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...
```

Подробнее про `__init__.py`

Внутри файл `__init__.py` можно прописать код, который будет выполняться при импорте пакета. Кроме того, в файл можно добавить переменную `__all__` с именами всех модулей пакета. Например мы добавим в учебный проект следующую строку:

```
__all__ = ['base_math', 'advanced_math']
```



Важно! При добавлении или удалении модулей в пакет важно вносить изменения в список `__all__` для корректной работы импорта.

Варианты импорта

Рассмотрим особенности импорта пакетов и модулей.

```
import mathematical
```

```
x = mathematical.base_math.div(12, 5)
```

Если импортировать пакет верхнего уровня, для работы с функциями необходимо указать всю цепочку через точечную нотацию: имя пакета, имя модуля, имя объекта.

Для сокращения объема кода обычно импортируют нужные модули или объекты модуля через точечную нотацию после `import`.

```
from mathematical import base_math as bm  
from mathematical.advanced_math import exp
```

```
x = bm.div(12, 5)
```

```
z = exp(2, 3)
```

В первом случае импортировали модуль `base_math` под именем `bm`. Во-втором — функцию `exp` из модуля `advanced_math` пакета `mathematical`.

Отдельно стоит упомянуть особенности импорта внутри подпакетов. Например импорт модуля `other_module` в другой модуль того же пакета можно осуществить через относительный импорт:

```
from . import other_module
```

А если модулю надо выйти из своего пакета в пакет верхнего уровня, используют вторую точку:

```
from .. import other_module
```

Или так

```
from ..other_package import other_module
```



Важно! Модуль, который является основным в вашем проекте должен использовать только абсолютные имена пакетов и модулей. Связано это с тем, что у запускаемого модуля в переменной `__name__` хранится значение `"__main__"`, а не имя модуля.

Задание

Перед вами список файлов в директории проекта. Напишите в чат какие файлы надо удалить или переименовать, а какие верные. У вас три минуты.

```
__all__.py  
__init__.py  
__main__.py  
init.py  
math.py  
random.py
```

3. Некоторые модули “из коробки”

Рассмотрим несколько модулей из стандартной библиотеки Python. Точнее даже не модули целиком, а некоторые функции из модулей.

Модуль sys

С модулем `sys` мы уже познакомились. Он относится к группе служебных в Python. Модуль `sys` обеспечивает доступ к некоторым переменным, используемым или поддерживаемым интерпретатором, а также к функциям, тесно взаимодействующим с интерпретатором. Изучать все переменные и функции не имеет смысла. Даже опытные разработчики не пользуются всеми, а их в модуле около сотни. При необходимости вы всегда сможете найти описание той или иной функции или переменной в официальной документации. Рассмотрим лишь одну переменную модуля — `argv`.

Запуск скрипта с параметрами

Python позволяет запускать скрипты с параметрами. Для этого после имени исполняемого файла указываются ключи и/или значения через пробел.

Например создадим файл script.py со следующим кодом.

```
print('start')  
  
print('stop')
```

Открываем консоль операционной системы и вводим команду на запуск.

```
python3 script.py
```



Важно! Для UNIX ОС используем команду python3. В Windows — python.



Важно! Не перепутайте консоль ОС и терминал интерпретатора Python. Терминал выдаёт в начале строки приветствие на ввод — тройную стрелку >>>.

Скрипт вывел текст в консоль и завершил работу. Научим его принимать значения из командной строки.

```
from sys import argv  
  
print('start')  
print(argv)  
print('stop')
```

Переменная argv содержит список. В нулевой ячейке имя запускаемого скрипта. В последующих ячейках переданные значения.

Например при запуске следующей строки:

```
python script.py -d 42 -s "Hello world!" -k 100
```

получим следующий список:

```
['script.py', '-d', '42', '-s', 'Hello world!', '-k', '100']
```

Обратите внимание, что все значения переданы как строки даже числовые. Строка “Hello world!” передана как один объект, потому что при вызове скрипта была заключена в двойные кавычки.

Переменная argv позволяет решать простые задачи работы с командной строкой. Если ваш будущий проект будет требовать более сложной обработки переданных в скрипт параметров, обратите внимание на встроенный в Python модуль argparse.

Модуль random

Модуль используется для генерации псевдослучайных чисел. Почти все функции модуля зависят от работы функции `random()`, которая генерирует псевдослучайные числа в диапазоне от нуля включительно до единицы исключительно — $[0, 1)$.



Важно! Генераторы псевдослучайных чисел модуля не должны использоваться в целях безопасности. Для обеспечения безопасности или криптографии необходимо использовать модуль `secrets`.

Для управления состоянием используют следующие 3 функции:

- `seed(a=None, version=2)` — инициализирует генератор. Если значение `a` не указано, для инициализации используется текущее время ПК. Версия 2 используется со времён Python 3.2 как основная. Не стоит менять её.
- `getstate()` — возвращает объект с текущим состоянием генератора.
- `setstate(state)` — устанавливает новое состояние генератора, принимая на вход объект, возвращаемый функцией `getstate`.

Рассмотрим несколько часто используемых функции генерации чисел.

- `randint(a, b)` — генерация случайного целого числа в диапазоне от `a` включительно до `b` включительно — $[a, b]$.
- `uniform(a, b)` — генерация случайного вещественного числа в диапазоне от `a` до `b`. Правая граница может как входить, так и не входить в возвращаемый диапазон. Зависит от способа округления.
- `choice(seq)` — возвращает случайный элемент из непустой последовательности.
- `randrange(stop)` или `randrange(start, stop[, step])` работает как вложение функции `range` в функцию `choice`, т.е. `choice(range(start, stop, step))`. Возвращает случайное число от `start` до `stop` с шагом `step`.
- `shuffle(x)` — перемешивает случайным образом изменяемую последовательность `in place`, т.е. не создавая новую.
- `sample(population, k, *, counts=None)` — выбирает `k` уникальных элементов из последовательности `population` и возвращает их в новой последовательности. Параметр `counts` позволяет указать количество повторов элемента.

Все описанные функции представлены в листинге ниже.

```
import random as rnd
```

```

START = -100
STOP = 1_000
STEP = 10
data = [2, 4, 6, 8, 42, 73]

print(rnd.random())
rnd.seed(42)
state = rnd.getstate()
print(rnd.random())
rnd.setstate(state)
print(rnd.random())

print(rnd.randint(START, STOP))
print(rnd.uniform(START, STOP))
print(rnd.choice(data))
print(rnd.randrange(START, STOP, STEP))

print(data)
rnd.shuffle(data)
print(data)

print(rnd.sample(data, 2))
print(rnd.sample(data, 2, counts=[1, 1, 1, 1, 1, 100]))

```

Задание

Перед вами пример кода. Что выведет программа после запуска? У вас три минуты.

```

import random
from sys import argv

print(random.uniform(int(argv[1]), int(argv[2])))
print(random.randrange(int(argv[1]), int(argv[2]), int(argv[1])))
print(random.sample(range(int(argv[1]),
                           int(argv[2]),
                           int(argv[1])), 10))

```

Скрипт запущен командой: `python3 main.py 10 1010`

Вывод

На этой лекции мы:

1. Разобрали работу с модулями в Python
2. Изучили особенности импорта объектов в проект
3. Узнали о встроенных модулях и возможностях по созданию своих модулей и пакетов
4. Разобрали модуль `random` отвечающий за генерацию случайных чисел

Краткий анонс следующей лекции

1. Разберёмся в особенностях работы с файлами и каталогами в Python
2. Изучим функцию `open` для работы с содержимым файла
3. Узнаем о возможностях стандартной библиотеки для работы с файлами и каталогами

Домашнее задание

Соберите файлы прошлых уроков в отдельные директории и сделайте их пакетами. Измените сами файлы, чтобы они были импортируемыми модулями.



Подсказка. Используйте файлы `__init__.py` и проверку переменной `__name__`.