Установка внешних библиотек. Работа с Git

Цель занятия

После освоения темы вы:

- узнаете термины, которые используются в работе с сетью, и основные сетевые протоколы;
- сможете использовать основные функции, методы, модули и библиотеки Python для написания собственных клиентских и серверных приложений.

План занятия

1. Инструкция `import`
2. Модули стандартной библиотеки
3. Создание своего модуля на Python
4. Создание виртуального окружения
5. Установка внешних библиотек Python
6. Инструменты статического анализа кода
7. Инструменты тестирования
8. Git, работа с распределенными системами управления версиями

Используемые термины

Модуль — файл, содержащий код, который можно повторно использовать в других программах.

Пакет — каталог, включающий в себя другие каталоги, файлы (модули) и специальный файл `__init__.py`.

Линтер — статический анализатор кода, который позволяет оценить стиль написания кода и соответствие определенным требованиям стиля написания.

Система контроля версий — программное обеспечение (утилита), позволяющее отслеживать изменения в проекте и при необходимости производить откат к предыдущим версиям.

Конспект занятия

1. 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.

2. 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.`Или 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).

3. 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.

4. Создание виртуального окружения

Прежде чем устанавливать виртуальные библиотеки, необходимо создать виртуальное окружение.

Виртуальное окружение — копия интерпретатора Python в отдельной папке. Все пакеты, которые будут устанавливаться, будут установлены в эту папку и не будут мешать работе системного интерпретатора.

Если вы работаете в Linux или macOS, у вас уже есть системный Python, который необходим самой операционной системе. Его лучше не изменять, устанавливая какие-либо пакеты. Есть риск «сломать» операционную систему.

Но даже если вы используете Python, который сами установили, лучше оставить его нетронутым. Во-первых, это убережет от поломки. Во-вторых, будет легче отследить, какие пакеты установлены, что именно работает и какую имеет версию.

В самом Python есть несколько модулей, которые можно запускать из терминала. Делается это следующим образом:

```
$ python -m venv venv
```

Здесь:

- `-m` — команда для запуска модуля;
- `venv` — имя модуля `virtual environment`;
- `venv` — название папки, в которую мы хотим установить виртуальное окружение.

Все действия нужно выполнять в каталоге работы с проектом:

```
$ ls
```

Таким образом была создана папка `venv`, которая содержит в себе несколько каталогов:

```
$ ls venv
```

Наиболее важный каталог `bin` содержит основные скрипты, которые мы будем запускать. Каталог `lib` содержит библиотеку Python.

В каталоге `bin` хранится несколько скриптов, которые позволяют активировать виртуальное окружение:

```
$ ls venv/bin
```

Есть несколько скриптов под разные операционные системы и оболочки. Для демонстрируемой оболочки `fish` (операционная система Linux) нужно запустить файл `activate.fish`:

```
$ source venv/bin/activate.fish
```

Если вы пользуетесь другой оболочкой — нужно будет запустить другой файл. Либо просто файл `activate`, если оболочка `bash`. Для командной строки Windows

запускают файл `activate.bat`, для оболочки PowerShell Windows — файл `Activate.ps1`.

После запуска в окне терминала появится дополнительное приглашение, показывающее, что мы находимся в виртуальном окружении. Далее Python будет запускаться из каталога `venv`. И все, что будет устанавливаться, будет установлено в этот же каталог, не мешая работе системы.

Для завершения работы в виртуальном окружении нужно дать команду:

```
$ deactivate
```

В операционной системе Linux есть команда, позволяющая узнать, какой файл будет запускаться той или иной командой:

```
$ which python
```

5. 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.

6. Инструменты статического анализа кода

Рассмотрим один из инструментов статического анализа кода — линтер `pylint`. **Линтер** — утилита, которая проверяет, что код «хорошо» написан. То есть написан с соблюдением определенных требований стиля и не содержит явных ошибок.

Установим утилиту `pylint` с помощью `pipenv`:

```
$ pipenv install pylint
```

Далее активируем виртуальное окружение:

```
$ pipenv shell
```

Запускаем утилиту `pylint` и передаем ей файл для анализа:

```
$ pylint some\ file.py
```

Утилита выдает общую оценку кода, и выводит список ошибок в стиле написания. Ошибки имеют код обозначения, буква в начале кода указывает на вид ошибки:

- W (warning) — предупреждение;
- C (convention) — нарушение соглашений;
- R (refactor) — рефакторинг;
- E (error) — ошибка.

Утилита `pylint` позволяет оставить в коде комментарии, чтобы настроить проверку.

Рассмотрим еще одну утилиту — `mypy`. Эта утилита проверяет соответствие типов в программе, если аннотации типов указаны.

Установим библиотеку mypy:

```
$ pipenv install mypy
```

Активировав новое виртуальное окружение, мы можем воспользоваться утилитой. Мы можем настроить `mypy` так, чтобы она проверяла весь проект, но пока сосредоточимся на проверке одного файла:

```
$ mypy student.py
```

Проверка всего проекта может занять много времени, поскольку утилита анализирует всю кодовую базу со всеми зависимостями. Для одного файла утилита сработает достаточно быстро.

В результате утилита покажет список всех ошибок с указанием строк, в которой данная ошибка содержится. Утилита проводит глубокий анализ кода, который позволяет избежать многих потенциальных ошибок. Но чтобы утилита была полезна, необходимо снабжать код аннотациями типов.

7. Инструменты тестирования

Написание тестов к коду — трудная рутинная работа. Но она может гарантировать качество кода.

В стандартной библиотеке Python есть модуль, который позволяет писать тесты к коду непосредственно в документации, — модуль `doctest`. Подробный разбор примера модуля можно посмотреть в документации. Чтобы посмотреть работу модуля, в конце файла есть строки:

```
if __name__ == "__main__":
    import doctest
```

```
    doctest.testmod()
```

После запуска файла ничего не появится. И это правильно, `doctest` прогоняет тесты, и если все сработало верно, никак это не отображает. Если будет ошибка — программа об этом сообщит. Если появится ошибка, незадокументированная в модуле, программа также сообщит об ошибке.

Работа модуля настраивается с помощью специальных комментариев:

```
# doctest: +NORMALIZE_WHITESPACE
```

заставляет проигнорировать лишние пробелы и прочитать данные вне зависимости от того, как расставлены пробелы.

```
# doctest: +ELLIPSIS
```

позволяет вставлять многоточие

В сложных проектах удобнее использовать библиотеку `pytest`. В тестирующий код импортируется нужная функция и модуль `pytest`. Далее мы пишем некоторые функции — все функции, которые начинаются со слова `test_`, библиотека `pytest` будет запускать. Сам файл также должен начинаться с `test_`.

Внутри каждой функции с помощью инструкции `assert` мы прописываем, что ожидаем получить.

Важно! В каждой тестирующей функции рекомендуется проверять что-то одно. Чаще всего имеет смысл проверить крайние случаи.

Чтобы выполнить тесты, необходимо зайти в терминал и запустить `pytest` без дополнительных параметров. Библиотека сама проанализирует каталог и увидит, что есть файлы с именем `test_`, и внутри них функции с `test_`. Утилита сама выполнит тесты. Утилита должна быть предварительно установлена.

8. Git, работа с распределенными системами управления версиями

Представим, что мы работаем над длительным проектом. И нам нужно иметь контроль истории проекта, чтобы посмотреть, какие изменения, когда и кем были внесены. Такие задачи решает система контроля версий.

Можно вместо системы контроля версий создать в каталоге проекта несколько папок с версиями проекта. Недостатки такого способа работы:

- требует наличия свободного места на диске;

- отсутствует автоматизированная система анализа истории проекта;
- можно забыть создать копию проекта.

Рассмотрим работу с системой контроля версий Git — самой популярной и известной системой, которая используется практически во всех open source проектах. Общие принципы работы подобных систем контроля версий достаточно похожи.

Разберем сначала работу через терминал. При работе в Windows необходимо установить не только утилиту Git, но и терминал, который будет с ней работать.

Чтобы создать проект в системе контроля версий (**репозиторий**), нужно дать команду:

```
$ git init
```

После чего появляется сообщение, что инициализирован пустой репозиторий Git по определенному пути. Утилита создает отдельный каталог — репозиторий. В этот каталог мы не будем заходить и работать с ним. С каталогом работает утилита Git, в нем хранится история версий и вся информация о репозитории. Мы работаем с проектом в текущей папке, как будто это обычная папка с файлами.

Важная команда, которой приходится пользоваться:

```
$ git status
```

Эта команда позволяет посмотреть, каков статус репозитория.

Система контроля версий позволяет создавать разные ветки, чтобы одновременно работать с разными версиями.

Коммит — важное понятие системы контроля версий, обозначает некоторое состояние файлов, зафиксированное в истории версий. Не любые изменения, которые вы вносите в файлы, будут автоматически фиксироваться. Это необходимо сделать специальной командой.

Создадим файл в проекте и посмотрим, как на это среагирует система контроля версий:

```
$ echo "Hello" > hello.txt
$ git status
```

Система говорит, что коммитов нет, но появился неотслеживаемый файл. Дело в том, что когда мы создаем какие-либо файлы в каталоге, система Git по умолчанию их не

отслеживает и не вносит в репозиторий. Причина — файлы могут появляться без ведома разработчика. Например, временные файлы или вспомогательные каталоги.

Чтобы включить файл в репозиторий воспользуемся:

```
$ git add hello.txt
```

Если еще раз проверить статус, то появится сообщение о новом файле. Теперь мы можем сделать коммит:

```
$ git commit -m 'add hello.txt'
```

Есть разные подходы к написанию сообщения коммита. В больших серьезных проектах, как правило, есть требования, что именно должно быть в сообщении. При этом само сообщение может быть достаточно большим. Когда сами работаем с сообщением, то достаточно, чтобы оно было понятным.

Сделаем еще несколько изменений:

```
$ echo "world" > hello.txt
$ git add hello.txt
$ git commit -m 'add world hello.txt'
```

Далее можно посмотреть историю коммитов:

```
$ git log
```

Каждый из коммитов имеет идентификатор — хеш-сумму, по которой можно найти коммит и переключиться на него.

Если требуется вернуться назад в историю, то тогда нужно найти требуемый коммит и переключиться на него:

```
$ git checkout идентификатор_коммита
```

При обращении к коммиту не обязательно полностью писать хеш-сумму, достаточно нескольких первых символов, по которым его можно однозначно определить. После этого появится сообщение с указанием текущего состояния.

Система указывает, что мы находимся в состоянии «отсоединенного указателя HEAD». Указатель HEAD указывает на коммит, в котором мы сейчас находимся.

Если мы посмотрим содержимое файла `hello.txt`, то увидим, что он содержит лишь одно слово `Hello`.

Таким образом, переключение на тот или иной коммит изменяет состояние файлов в текущем каталоге.

Состояние «отсоединенного указателя HEAD» лучше не использовать. В нем можно что-либо посмотреть или создать новую ветку. Если сейчас сделать какие-то коммиты, то они потеряются, если их не включить в какую-либо ветку.

Если мы захотим вернуться в предыдущее состояние, это можно сделать с помощью команды `checkout`, указав хеш-сумму для коммита. Но при этом мы все равно останемся в состоянии «отсоединенного указателя HEAD».

Поэтому правильнее будет записать (master — это название главной ветки):

```
$ git checkout master
```

Так мы будем находиться в текущем наиболее актуальном состоянии.

Утилита Git содержит справку:

```
$ git help
```

В справке приведено краткое описание команд утилиты с пояснением, что они делают. Получить справку можно и по конкретной команде:

```
$ git help имя_команды
```

Предположим, что в истории есть коммит, который особенно важен. Например, этот коммит содержит готовую рабочую версию. Не очень удобно искать его с помощью хеш-сумм, гораздо присвоить ему какое-либо имя и затем это имя использовать:

```
$ git tag имя_версии
```

Рассмотрим ситуацию, когда в файл были ошибочно внесены изменения, и нужно вернуться к предыдущему состоянию. Если изменения не были внесены в коммит, команда `git status` это покажет. Отменить изменения в файле можно с помощью команды:

```
$ git restore hello.txt
```

Сделанный коммит можно тоже отменить, если он не был записан в удаленный репозиторий. Команда отмены коммита существует в нескольких формах, главные из них:

- soft reset — оставляет состояние файлов на диске, которое соответствует состоянию сейчас. Как будто все коммиты сделаны, но при этом мы переходим на предыдущий коммит. Такое может быть полезно, если мы внесли правильные изменения в файлы, но неправильно их закоммитили.
- hard reset — уничтожает все изменения, переходя к состоянию в прошлом.

Рассмотрим пример отмены коммита с полной отменой сделанных изменений:

```
$ git reset --hard HEAD^
```

`HEAD^` означает, что нужно вернуться от текущего коммита на один коммит назад.

Пример, когда нужно только изменить сообщение коммита:

```
$ git commit --amend
```

После выполнения команды открывается текстовый редактор, в котором можно внести изменения. Вид текстового редактора зависит от настроек системы.

Утилита Git позволяет работать с удаленным репозиторием, который расположен на сервере. Есть полезные веб-сервисы, организующие и упрощающие работу с удаленными репозиториями, самый известный — GitHub. Для работы с удаленными репозиториями служат две команды:

- `git pull` — подгружает изменения с удаленного репозитория, если они там появились;
- `git push` — отправляет изменения на сервер.

После отправки изменений на сервер в удаленный репозиторий, в нем уже будет сохранена история, которую нельзя отменить.

Иногда удобнее работать с системой контроля версий Git не через терминал, а с помощью графического интерфейса, например, в PyCharm.

PyCharm

Как создать свой репозиторий

При открытии PyCharm по умолчанию создается виртуальное окружение. Чтобы создать новый проект, нужно в меню найти VCS (Version Control System), нажать Enable Version Control Integration и выбрать систему Git (Рис. 1).

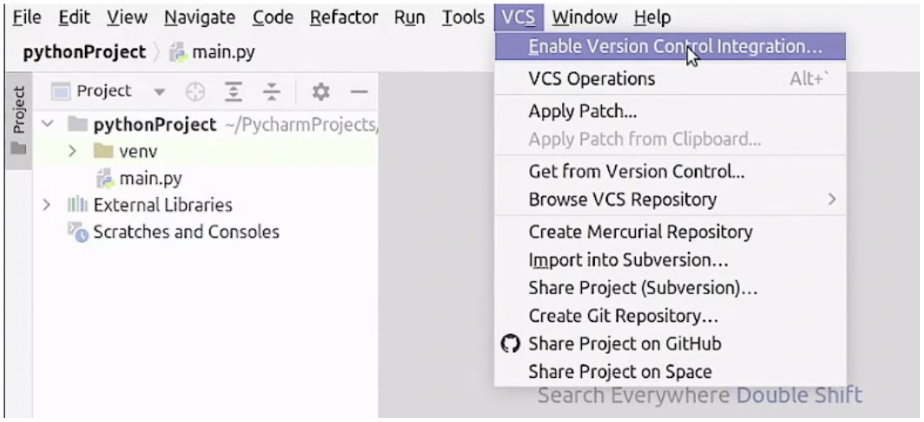

Рисунок 1. Как создать новый репозиторий в PyCharm

Перед началом работы над новым проектом проверим его содержание. В меню ниже нажимаем commit (галочка), и среди файлов выбираем main.py (все эти файлы не внесены в систему контроля версий), чтобы внести его в репозиторий.

Команда `initial commit` позволит нам перенести файл в систему контроля версий.

Важно! Файлы, которые были созданы по умолчанию самой программой PyCharm, переносить в репозиторий не нужно и даже вредно. Они отражают настройки программы специализированно под операционную систему создателя репозитория и если их загрузить с другого компьютера, то программа может не заработать. Лучше удалять эти файлы из репозитория.

Чтобы удалить ненужные файлы из репозитория, следует создать файл **.gitignore**. Заходим в папку проекта и создаем новый файл с таким названием. В этот файл можно добавить названия файлов, которые не будут отслеживаться. Например, можно добавить в нее .idea, __pycache__, *.jpg и другие файлы по мере создания кода. Не забудьте закоммитить файл .gitignore.

Как посмотреть изменения

PyCharm позволяет удобно просматривать изменения, которые были внесены в файл. Они отображаются в разделе Commit с названием файла, в который были внесены изменения, и версиями до изменений и после изменений. Если изменение вас устраивает, его можно закоммитить.

Важно! Пишите сообщение в commit кратко, но понятно, чтобы потом можно было отследить, что было сделано.

Как добавить свой репозиторий на сайт GitHub:

1. Подключите аккаунт к PyCharm (зарегистрируйтесь на сайте, если у вас еще нет аккаунта).
2. В меню нажмите Git -> GitHub -> Share Project on GitHub.
3. Можете поставить галочку и оставить проект приватным.
4. Нажмите Share.

После этого вам будут доступны все команды для работы с удаленным репозиторием.

Кроме PyCharm существуют и другие графические приложения для работы с репозиториями. В них так же будут доступны все опции работы терминала Git.

Как работать с удаленным репозиторием в группе из нескольких человек

Сначала подгрузите свой проект на GitHub. Затем подождите, пока программа загрузит на сайт текущее состояние вашего проекта. После этого вы можете открывать ваши файлы через браузер и вносить в них изменения.

При командной работе может возникнуть сложная ситуация, когда одновременно несколько человек внесли изменения в один и тот же файл репозитория. В таком случае коммиты, созданные с разных серверов, будут конфликтовать друг с другом, и операция завершится сообщением об отклонении действия.

Чтобы решить проблему нажмите pull и в открывшемся окне выберите действие: принять свою версию, принять чужую версию или объединить версии. При объединении нужно вручную зафиксировать изменения и показать программе, как по итогу будет выглядеть код.

Важно! После того как решите проблему, нажмите push, чтобы отправить новую версию на сервер.

Обычно при командной работе существуют определенные правила, которые позволяют минимизировать количество ошибок и подобных проблем.

Дополнительные материалы для самостоятельного изучения

1. The Python Standard Library
2. Installing Python Modules
3. pip documentation v22.3.1

4. Pipenv: Python Dev Workflow for Humans
5. Poetry - Python dependency management and packaging made easy
6. Pylint User Manual
7. Welcome to mypy documentation!
8. doctest — Test interactive Python examples
9. pytest Documentation
10. Learn Git Branching (тренажер по работе с Git)