Лабораторная работа 3

Постановка задачи: Найдите информацию по новым типам данных или изменениям, касающихся стандартных типов данных в Python.

Особое внимание следует уделить такого рода изменениям в версиях Python 3.7, 3.8, 3.9 и следующих версиях языка (см. предложения, описанные в PIP).

Python 3.7.

Встроенный breakpoint()

Дебагинг – это важная часть программирования. Python 3.7 предоставляет новую встроенную функцию breakpoint(). Она не дает Python никакого нового функционала, но делает процесс дебагинга более интуитивным и гибким.

Брейкпоинт – это сигнал внутри вашего кода, чьё выполнение должно на время приостановиться, так что вы можете разобраться с текущим положением программы. Как его разместить? В Python 3.6 и ранее, мы пользовались этой странной строкой:

```
def divide(e, f):
    import pdb; pdb.set_trace()
    return f / e
```

Здесь pdb – это дебагер Python из стандартной библиотеки. В Python 3.7 вы можете использовать вызов новой функции breakpoint() в качестве короткого пути:

```
def divide(e, f):
    breakpoint()
    return f / e
```

За кулисами, breakpoint() сначала импортирует pdb, после чего вызывает для вас pdb.set_trace(). Очевидная польза в том, что breakpoint() проще запомнить, и нужно ввести только 12 символов, вместо 27. Однако, главный бонус в использовании breakpoint() – это простота настройки.

Запустите скрипт bugs.py вместе с breakpoint():

```
$ python3.7 bugs.py
```

```
> /home/gahjelle/bugs.py(3)divide()
-> return f / e
(Pdb)
```

Новая функция breakpoint() работает не только с дебагерами. Еще один удобный пример использования – это просто создать интерактивную оболочку внутри вашего кода. Например, чтобы начать сессию IPython, вы можете использовать следующее:

```
$ PYTHONBREAKPOINT=IPython.embed python3.7 bugs.py
IPython 6.3.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: print(e / f)
0.0
```

Вы также можете создать собственную функцию, и указать breakpoint(), чтобы он её вызывал. Следующий код выводит все переменные в локальном масштабе. Добавьте его в файл под названием bp_utils.py:

```
from pprint import pprint
import sys

def print_locals():
    caller = sys._getframe(1)  # Caller на 1 фрейм выше
    pprint(caller.f_locals)
```

Чтобы использовать эту функцию, настройте PYTHONBREAKPOINT как и ранее, при помощи обозначения .:

```
$ PYTHONBREAKPOINT=bp_utils.print_locals python3.7 bugs.py
{'e': 0, 'f': 1}
ZeroDivisionError: division by zero
```

Обычно, breakpoint() будет использован для вызова функций и методов, которым не нужны аргументы. Однако, вы можете передавать аргументы в том числе. Измените строк у breakpoint() в bugs.py на:

```
breakpoint(e, f, end="<-END\n")
```

Запустите этот код с breakpoint(), маскирующейся под функцию print(), чтобы увидеть простой пример передачи аргументов:

```
$ PYTHONBREAKPOINT=print python3.7 bugs.py
0 1<-END
ZeroDivisionError: division by zero
```

Классы данных

Новый модуль dataclasses упрощает написание собственных классов, так как специальные методы, такие как .**init**(), .**repr**() и .**eq**() добавлены автоматически. Используя декоратор @dataclass, вы можете написать что-нибудь в духе следующего:

```
from dataclasses import dataclass, field

@dataclass(order=True)
class Country:
    name: str
    population: int
    area: float = field(repr=False, compare=False)
    coastline: float = 0

    def beach_per_person(self):
        """Метры береговой линии на человека"""
        return (self.coastline * 1000) / self.population
```

Главная цель классов данных – это писать надежные классы быстро и легко, в частности, небольшие классы, которые (в целом) рассчитаны на хранение данных.

Классы данных делают то же самое, что и namedtuple. Однако, самое большое вдохновение черпается ими из проекта attrs.

Типизация данных

Контроль типа и аннотаций были в постоянной разработки в течение всей серии Python 3. Типизирование Python теперь стала заметно стабильнее. При этом Python 3.7 предоставляет заметные улучшения: лучшая производительность, поддержка ядра, и дальнейшие референсы.

К сожалению, то, что использования типизирования требует модуль typing — не совсем правда. Модуль typing – один из самых медленных в стандартной библиотеке. PEP 560 добавляет кое-какую поддержку ядра для типизирования в Python 3.7, что значительно ускоряет модуль typing.

Так как типизирование в Python объяснимо впечатляет, есть небольшая проблема – это предварительное объявление. Типизирование – оцениваются во время импорта модуля. Поэтому все имена уже должны быть определены перед использованием. Мы не можем выполнить следующее:

```
class Tree:
    def __init__(self, left: Tree, right: Tree) -> None:
        self.left = left
        self.right = right
```

Запуск кода приводит к ошибке NameError, так как класс Tree еще не до конца определен в определении метода .**init**():

```
Traceback (most recent call last):
  File "tree.py", line 1, in <module>
    class Tree:
  File "tree.py", line 2, in Tree
    def __init__(self, left: Tree, right: Tree) -> None:
NameError: name 'Tree' is not defined
```

Чтобы обойти это, вам нужно вписать «Tree» в виде строкового литерала вместо:

```
class Tree:
    def __init__(self, left: "Tree", right: "Tree") -> None:
        self.left = left
        self.right = right
```

Python 3.8.

Оператор морж (walrus)

Заглавная особенность Python 3.8, и в то же время — самая спорная. Путь принятия решения о PEP 572 («Assignment Expressions») был довольно ухабистым, что привело к новой модели управления языком.

Представим, что новое правительство готовится заменить давнего доброжелательного диктатора, которого мы с вами знаем всю жизнь, Гвидо ван Россума, после того как он уйдет в отставку из-за беспорядков, связанных с PEP 572 (конфликт между сообществом разработчиков из за добавления нового синтаксиса).

В связи с этим выходит новый оператор, который часто называют “оператор-морж” из-за ассоциации с его отображением. Использование := в if или while позволяет присвоить значение переменной во время тестирования. Предполагается, что это упростит такие задачи как сопоставление с несколькими шаблонами, так называемые “полтора цикла”, итак:

Было:

```
m = re.match(p1, line)
if m:
    return m.group(1)
else:
    m = re.match(p2, line)
    if m:
        return m.group(2)
    else:
        m = re.match(p3, line)
        ...
```

Стало:

```
if m := re.match(p1, line):
    return m.group(1)
elif m := re.match(p2, line):
    return m.group(2)
elif m := re.match(p3, line):
    ...
```

И цикл над неповторяемым объектом, например:

```
ent = obj.next_entry()
while ent:
    ...   # process ent
    ent = obj.next_entry()
```

Может стать:

```
while ent := obj.next_entry():
    ... # process ent
```

Позиционные параметры

Еще одно изменение в Python 3.8 предоставляет чистым функциям Python все те же опции для параметров, что уже реализованы в C. PEP 570 («Python Positional-Only Parameters») вносит новый синтасксис, который может быть использован в определениях функции для обозначения только позиционных аргументов — параметров, которые не могут быть переданы в качестве аргументов ключевых слов.

Например, встроенная функция pow() должна вызываться с соответствующими аргументами:

```
pow(2, 3) # 8

pow(x=2, y=3)
...
TypeError: pow() takes no keyword arguments
```

Но если бы pow() была чистой-функцией Python, чего и ожидала бы альтернативная реализация Python, было бы трудно изменить это поведение. Функция может принимать только *args и *kwargs, затем ставить условие, что kwargs является пустым, однако потом скрывает, что собирается сделать функция. Есть и другие причины, описанные в PEP, но с большей частью из них Python-программисты не то чтобы часто сталкиваются.

Однако те, кто сталкиваются, возможно обрадуются тому факту, что они могут написать функцию pow() на чистом Python, которая будет вести себя так же, как и встроенная. Вот так:

```
def pow(x, y, z=None, /):
    r = x**y
    if z is not None:
        r %= z
    return r
```

Наш / обозначает конец позиционных параметров в списке аргументов. Суть такая же, как и для *, который может быть использован в списке аргументов для делимитации ключевых аргументов (те, которые могут быть переданы как keyword=...), что указано в PEP 3102 («Keyword-Only Arguments»).

Итак, следующий пример:

```
def fun(a, b, /, c, d, *, e, f):
    ...
```

Говорит нам, что a и b должны быть переданы позиционально, c и d могут быть переданы как позиционально, так и как ключевое слово, наши e и f должны быть переданы по ключевому слову. Итак:

КОД # легально КОД # легально КОД # нелегально

```
fun(1, 2, 3, 4, e=5, f=6)          # правильно
fun(1, 2, 3, d=4, e=5, f=6)        # правильно
fun(a=1, b=2, c=3, d=4, e=5, f=6)  # неправильно
```

Похоже, что большая часть Python-программистов не сталкивалась как с *, так и с /.

Подвижный **pycache **

Директория **pycache** создана интерпретатором Python 3 (начиная с версии 3.2) для хранения файлов .pyc . Эти файлы содержат байт код, который кешируется после того, как интерпретатор компилирует файлы .py . Ранние версии Python просто выкидывали файлы .pyc, но PEP 3147 («PYC Repository Directories») изменило это.

Намерением была поддержка множественных установленных версий Python, наряду с вероятностью того, что многие из них могут не быть CPython вообще (например PyPy). Так что, например, стандартные файлы библиотеки могут быть скомилированы и кэшированы каждой версией Python так, как нужно.

Каждый может записать файл типа name.interp-version.pyc в **pycache**. Так что, например на системе Fedora, foo.py будет скомилирован при первом использовании, и будет создан **pycache**/foo.cpython-37.pyc

С точки зрения производительности — это отлично, но может быть не так оптимально по другим причинам. Карл Мейер выполнил запрос функции, запрашивая у переменной среды сказать Python, где найти (и внести) эти файлы кэша. Он столкнулся с проблемами доступов в его системе и в конечном счете с отключением файлов кэша.

Итак, он добавил переменную среды PYTHONPYCACHEPREFIX (также доступную через флаг командной строки -X pycache_prefix=PATH) для указания интерпретатору о том, что место для хранения этих файлов будет другим.

Python 3.9.

Оператор объединения словарей (PEP-584)

До этих пор, объединить словари можно было несколькими способами, однако каждый из них имел небольшие недостатки или нюансы.

Теперь можно писать просто:

```
united_dict = d1 | d2
# или, для того чтобы добавить элементы одного словаря другому, что аналогично методу update(
d1 |= d2
```

Упрощение аннотаций для контейнеров и других типов, которые могут быть параметризованы (PEP-0585)

Следующее нововведение очень пригодится тем, кто пользуется аннотацией типов. Теперь упрощается аннотация коллекций, таких как list и dict, и вообще параметризованных типов.

Для таких типов вводится термин Generic — это тип который может быть параметризован, обычно контейнер. Например, dict. И вопрос в том, как следует корректно переводить его, так чтобы не ломило зубы. Очень уж не хочется пользоваться словом «дженерик». Так что в комментариях очень жду другие предложения. Может где-то в переводах встречалось получше название? Параметризованный generic: dict[str, int].

Так вот для таких типов теперь не надо импортировать соответствующие аннотации из typing, а можно использовать просто названия типов:

```
# раньше
from typing import OrderedDict
OrderedDict[str, int]
# теперь
from collections import OrderedDict
OrderedDict[str, int]
```

Несколько изменений в модуле math

Функция math.gcd() нахождения наибольшего общего делителя теперь принимает список целых чисел, так что можно находить одной функцией общий делитель больше, чем для двух чисел.

Появилась функция для определения наименьшего общего кратного math.lcm(), которая так же принимает неограниченное количество целых чисел.

Следующие две функции взаимосвязаны. math.nextafter(x, y) — вычисляет ближайшее к x число с плавающей точкой, если двигаться в направлении y. math.ulp(x) — расшифровывается как «Unit in the Last Place» и зависит от точности расчетов вашего компьютера. Для положительных чисел вернется наименьшее значение числа, такое что при его прибавлении x + ulp(x) получится ближайшее число с плавающей точкой.

```
import math

math.gcd(24, 36)
>> 12
math.lcm(12, 18)
>> 36
math.nextafter(3, -1)
>> 2.9999999999999996
3 - math.ulp(3)
>> 2.9999999999999996
math.nextafter(3, -1) + math.ulp(3)
>> 3.0
```

Новый класс functools.TopologicalSorter для топологической сортировки направленных ациклических графов (bpo-17005)

Граф который передается в сортировщик, должен быть словарем, в котором ключи выступают вершинами графа, а значением является итерируемый объект с предшественниками (вершинами, дуги которых указывают на ключ). В качестве ключа, как обычно, подойдет любой хешируемый тип.

И еще несколько изменений:

Ускорены встроенные типы (range, tuple, set, frozenset, list) (PEP-590)

"".replace("", s, n) теперь возвращает s, а не пустую строку, для всех ненулевых n. Это выполняется и для bytes и bytearray.

ipaddress теперь поддерживает парсинг адресов IPv6 с назначениями.