Домашнее задание №6

Дедлайн 1 (16 баллов): 18 октября, 13:00
Дедлайн 2 (8 баллов): 25 октября, 13:00

Домашнее задание нужно написать на Python 3 и сдать в виде одного файла. Правило именования файла: `name_surname_06.py`. Например, если вас зовут Иван Петров, то имя файла должно быть: `ivan_petrov_06.py`.

Перед отправкой решения убедитесь, что оно соответствует рекомендациями из списка pydonts: `https://github.com/superbobry/pydonts`.

1 В объектной системе Python отсутствует понятие интерфейса, то есть нельзя гарантировать, что класс определяет некоторое фиксированное множество атрибутов и методов. Этот "пробел" можно восполнить с помощью декоратора класса.

а Напишите функцию `peel`, которая принимает класс и возвращает множество всех публичных атрибутов и методов класса. Если у класса есть родитель, то результат функции `peel` должен включать также публичные методы родителя. Публичными считаются атрибуты и методы, имя которых не начинается с символа подчёркивания.

```
>>> class AbstractBase:
...     def some_method(self):
...         pass
...
>>> class Base(AbstractBase):
...     def some_other_method(self):
...         pass
...
>>> class Closeable(Base):
...     def close(self):
...         pass
...
>>> peel(Closeable)
{"some_method", "some_other_method", "close"}
```

Вам может быть полезна функция `dir`, которая принимает класс или экземпляр класса и возвращает список его атрибутов и методов с учётом наследования.

```
>>> class A:
...     some_list = []
...
>>> class B(A):
...     some_other_attribute = 42
...
>>> dir(A)
[..., 'some_list']
>>> dir(B())
[..., 'some_list', 'some_other_attribute']
```

б Реализуйте декоратор класса `implements`, который проверяет, что класс реализует "интерфейс", переданный декоратору в качестве аргумента. Для простоты можно считать, что класс реализует "интерфейс", если все публичные атрибуты и методы "интерфейса" являются публичными методами и атрибутами проверяемого класса.

```
>>> class Closeable:
...     def close(self):
...         pass
...
>>> @implements(Closeable)
... class FileReader:
...     # ...
...     def close(self):
...         self.file.close()
```

В случае неуспешной проверки должен сработать оператор **`assert`**:

```
>>> @implements(Closeable)
... class Noop:
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in wrapper
AssertionError: method 'close' not implemented
```

2 Символьное дифференцирование позволяет вычислить значение производной для выражения, заданного в виде последовательности символов. Людям свойственно делать ошибки при длительной однообразной работе, а вычисление производной, хоть и несложный, но утомительный процесс. Системы символьных вычислений[1] призваны избавить человечество от необходимости тратить своё время на потенциально ошибочные расчёты.

Мы попробуем реализовать простую систему символьного дифференцирования на Python. Основным типом в нашей системе будет класс `Expr` — выражение. Все наследники класса должны реализовывать:

- метод `__call__`, вычисляющий значение выражения в заданном контексте: контекст связывает имена переменных в выражении с конкретными значениями.
- метод `d`, который принимает название переменной `wrt` (*англ.* with respect to) и возвращает выражение для производной по этой переменной.

```
>>> class Expr:
...     def __call__(self, **context):
...         pass
...
...     def d(self, wrt):
...         pass
...
```

а Реализуйте классы для двух типов выражений: `Const` — константу и `Var` — переменную. Для удобства далее будем пользоваться не конструкторами классов, а их однобуквенными синонимами:

```
>>> V = Var
>>> C = Const
```

Пример использования классов:

```
>>> C(42)()
42
>>> C(42).d(V("x"))()
0
>>> V("x")(x=42)
42
>>> V("x").d(V("x"))()
1
>>> V("x").d(V("y"))()
0
```

б Реализуйте классы для бинарных операций: `Sum`, `Product` и `Fraction`. Бинарные операции по определению работают ровно с двумя операндами, поэтому конструктор у всех бинарных операций будет одинаковый. Его удобно вынести в отдельный базовый класс:

```
>>> class BinOp(Expr):
...     def __init__(self, expr1, expr2):
...         self.expr1, self.expr2 = expr1, expr2
...
```

Пример использования некоторых бинарных операций:

```
>>> x = V("x")
>>> Sum(x, Product(x, x)).d(x)(x=42)
```

[1] http://bit.ly/wolfram-sd

```
85
>>> Product(x, Sum(x, C(2)))(x=42)
1848
>>> Fraction(Product(x, V("y")), Sum(C(42), x)).d(x)(x=42, y=24)
0.14285714285714285
>>> Fraction(Product(x, V("y")), Sum(C(42), x)).d(V("y"))(x=42, y=24)
0.5
```

в Реализуйте класс `Power` для операции возведения в степень. Для простоты можно считать, что степень — это выражение, состоящее только из констант.

```
>>> Power(Fraction(V("x"), C(4)), C(2))(x=42)
110.25
>>> Power(Fraction(V("x"), C(4)), C(2)).d(V("x"))(x=42)
5.25
```

г Реализуйте для всех типов выражений метод `__str__`, который должен возвращать соответствующую формулу в виде S-выражения.

```
>>> x = V("x")
>>> print(Sum(x, Product(x, x)).d(x))
(+ 1 (+ (* x 1) (* 1 x)))
>>> print(Product(x, Sum(x, C(2))))
(* x (+ x 2))
>>> print(Power(Fraction(x, C(4)), C(2)))
(** (/ x 4) 2)
```

3 Базовая функциональность системы символьного дифференцирования уже готова, но пользоваться ей не очень удобно.

а Реализуйте перегрузку арифметических операторов для базового класса `Expr`. Всего нужно реализовать 7 операторов: 2 унарных и 5 бинарных:

```
-e        e.__neg__()
+e        e.__pos__()
e1 + e2   e1.__add__(e2)
e1 - e2   e1.__sub__(e2)
e1 * e2   e1.__mul__(e2)
e1 / e2   e1.__truediv__(e2)
e1 ** e2  e1.__pow__(e2)
```

Пример использования перегрузки операторов при определении выражения:

```
>>> print((C(1) - V("x")) ** C(3) + V("x"))
(+ (** (+ 1 (* -1 x)) 3) x)
```

Обратите внимание, что вместо выражения `1 - x` выводится `(+ 1 (* -1 x))`. Это результат того, что среди наших базовых операций нет вычитания.

в Реализуйте функцию `newton_raphson`, которая принимает выражение f — функцию от переменной x, начальное значение x_0 и положительное вещественное число ϵ. Результатом работы функции является значение x, обращающее выражение f в ноль.

Метод Ньютона-Рафсона используется для итеративного нахождения нуля некоторой дифференцируемой функции $f(x)$. Метод принимает на вход начальное значение x_0,

а затем итерирует правило $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ до тех пор, пока не будет удовлетворён критерий остановки $|x_{n+1} - x_n| \leq \epsilon$.

```
>>> expr = (V("x") + C(-1)) ** C(3) + V("x")
>>> zero = newton_raphson(expr, 0.5, threshold=1e-4)
>>> print(expr)
(+ (** (+ x -1) 3) x)
>>> print(zero, expr(x=zero))
0.31767219617165293 -7.855938122247608e-13
```

4 Система символьного дифференцирования работает, но её производительность оставляет желать лучшего. Попробуем изменить это, реализовав оптимизацию — “сворачивание” констант.

а Добавьте к классу `Expr` и ко всем его наследникам свойство `is_constexpr`, возвращающее **`True`**, если выражение состоит только из констант, и **`False`** в обратном случае.

```
>>> (V("x") + C(1)).is_constexpr
False
>>> (C(1) + C(42) * V("x")).d(V("x")).is_constexpr
False
>>> (C(1) + C(42) * C(2)).is_constexpr
True
```

б Дополните свойство `is_constexpr` свойством `simplified`. Значением свойства должно быть выражение в котором все подвыражения, состоящие только из констант, “свёрнуты”, то есть заменены на вычисленные значения.

```
>>> print((C(1) + C(42) * C(2)).simplified)
85
>>> print((C(1) * V("y") + C(42) * C(2) / V("x")).simplified)
(+ (* 1 y) (/ 84 x))
```

Вам может быть полезен “магический” атрибут `__class__` у экземпляра класса `BinOp`:

```
>>> expr = V("x") + C(2) * C(4)
>>> print(expr.__class__(V("x"), C(8)))
(+ x 8)
```