при поддержке
{а}вижн

Лекция 17

Перевод и адаптация материалов Джона ДеНиро (John DeNero). Используется с разрешения автора.

Строковые представления

Строковые представления

Значение объекта должно «вести себя» в соответствии с видом данных, которые он описывает.

Например, создавать собственные строковые представления.

Значения строк высоко: они представляют язык и программы

В Python все объекты имеют два строковых представления:

- **str**: представление для людей
- **repr**: представление для интерпретатора Python

Строковые представления **str** и **repr** часто одинаковы, но не всегда

Строка repr для объекта

Функция **repr** возвращает выражение Python (строку), которое при вычислении возвращает объект равный исходному

```
repr(object) -> строка

Возвращает каноническое строковое представление объекта.
Для большинства объектных типов, eval(repr(object)) == object.
```

Для значений результат применения **repr** соответствует тому, что Python выводит в интерактивной сессии

```
>>> 12e12
12000000000000.0
>>> print(repr(12e12))
12000000000000.0
```

Некоторые объекты не имеют простого строкового представления «понятного» интерпретатору Python

```
>>> repr(min)
'<built-in function min>'
```

Строка str для объекта

Строки для понимания людьми также полезны:

```
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> repr(half)
'Fraction(1, 2)'
>>> str(half)
'1/2'
```

Результат применения **str** к значению выражения соответствует тому, что Python выводит при печати с помощью функции **print**:

```
>>> print(half)
1/2
```

(Пример)

Полиморфные функции

Полиморфные функции

Полиморфная функция: функция, которая способна обрабатывать множество (поли-) различных форм (морф) данных.

Функции **str** и **repr** полиморфны; они применимы к любому объекту.

Функция **repr** вызывает безаргументный метод __repr__ своего аргумента

```
>>> half.__repr__()
'Fraction(1, 2)'
```

Функция **str** вызывает безаргументный метод __str__ переданного ей объекта.

```
>>> half.__str__()
'1/2'
```

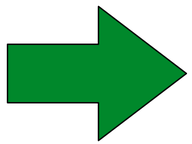
Реализация repr и str

Поведение **repr** немногим более сложно, чем просто вызов метода `__repr__` аргумента:

- Атрибут экземпляра `__repr__` игнорируется! Только атрибуты класса!
- *Вопрос*: Как реализовать такое поведение?

Поведение **str** также нетривиально:

- Атрибут экземпляра `__str__` игнорируется
- Если не найден атрибут `__str__`,
  то используется результат **repr**
- *Вопрос*: Как реализовать такое поведение?
- **str** – это класс, а не функция



```
def repr(x):
    return x.__repr__(x)
```

```
def repr(x):
    return x.__repr__()
```

```
def repr(x):
    return type(x).__repr__(x)
```

```
def repr(x):
    return type(x).__repr__()
```

```
def repr(x):
    return super(x).__repr__()
```

(Пример)

Интерфейсы

Передача сообщений: объекты взаимодействуют через обращения к атрибутам друг друга.

Правила поиска атрибутов позволяют различным типам данных по-разному реагировать на одно и то же сообщение.

Общие сообщения (имена атрибутов), которые вызывают одинаковое поведение у различных классов объектов — мощный метод абстракции

Интерфейс — это набор общих сообщений, вместе с описанием их поведения.

Например:

Классы реализующие методы `__repr__` и `__str__`, которые возвращают соответствующие строки, реализуют интерфейс строковых представлений.

Особые имена методов

Особые имена методов

Некоторые имена являются особыми, поскольку для них определено встроенное поведение.

Эти имена всегда начинаются и заканчиваются двойным подчёркиванием.

`__init__`	Вызывается автоматически при создании объекта
`__repr__`	Вызывается для вывода объекта в виде выражения Python
`__add__`	Вызывается при сложении объектов
`__bool__`	Вызывается для приведения объекта к True или False
`__float__`	Вызывается для приведения объекта к float

```
>>> zero, one, two = 0, 1, 2
>>> one + two
3
>>> bool(zero), bool(one)
(False, True)
```

То же поведение с использованием методов

```
>>> zero, one, two = 0, 1, 2
>>> one.__add__(two)
3
>>> zero.__bool__(), one.__bool__()
(False, True)
```

Особые методы

Сложение экземпляров пользовательских объектов вызывает методы __add__ или __radd__.

```
>>> Ratio(1, 3) + Ratio(1, 6)
Ratio(1, 2)

>>> Ratio(1, 3).__add__(Ratio(1, 6))
Ratio(1, 2)

>>> Ratio(1, 6).__radd__(Ratio(1, 3))
Ratio(1, 2)
```

https://habr.com/ru/post/186608/

http://docs.python.org/py3k/reference/datamodel.html#special-method-names

(Пример)

Обобщенные функции

Полиморфная функция может принимать два и более аргументов разного типа.

Диспетчеризация типов: определяется тип аргумента и выбирается поведение.

Приведение типов: преобразование значения к другому типу

```
>>> Ratio(1, 3) + 1
Ratio(4, 3)

>>> 1 + Ratio(1, 3)
Ratio(4, 3)

>>> from math import pi
>>> Ratio(1, 3) + pi
3.474925986923126 6
```

(Пример)

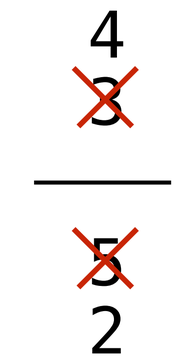
Методы свойств

Методы свойств

Зачастую необходимо чтобы значения атрибутов экземпляра были непротиворечивы

```
>>> f = Rational(3, 5)
>>> f.float_value
0.6
>>> f.numer = 4
>>> f.float_value
0.8
>>> f.denom -= 3
>>> f.float_value
2.0
```

Нет вызова метода!



Декоратор @property перед описанием безаргументного метода указывает на то, что этот метод будет вызван при любом обращении к этому атрибуту.

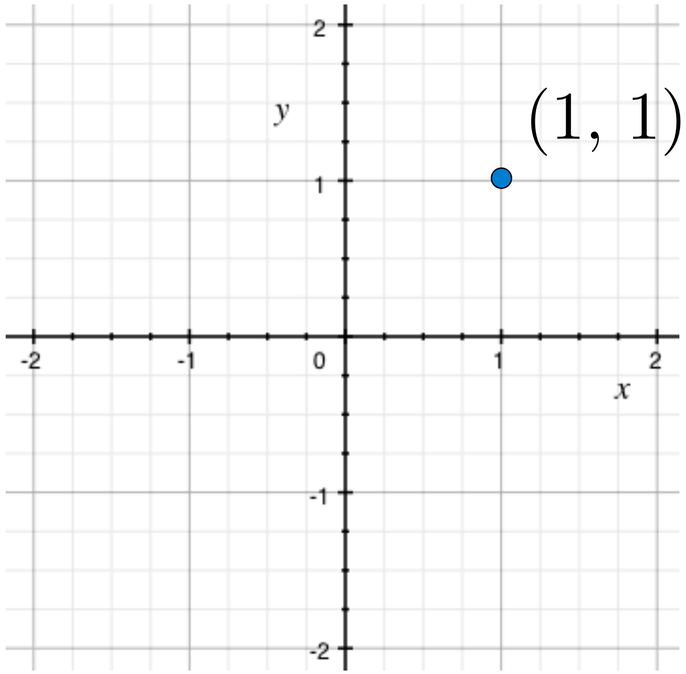
Этот декоратор позволяет вызывать методы неявно (без вызывающего выражения)

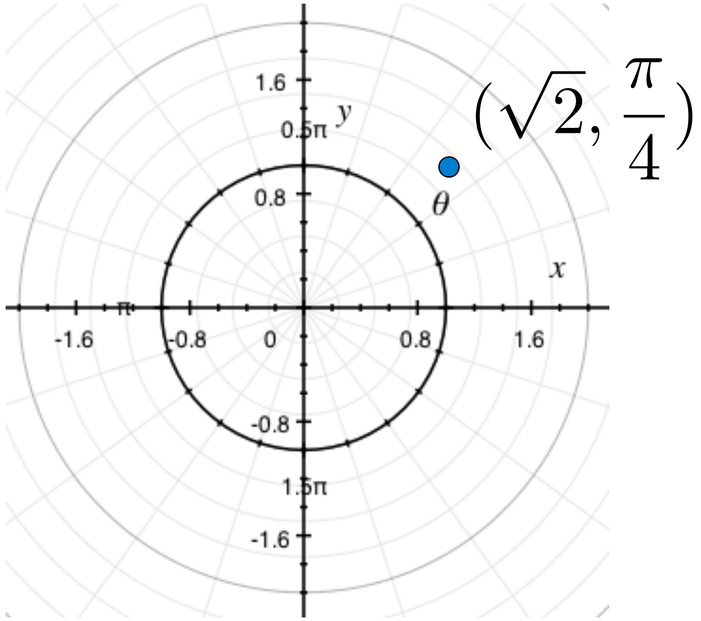
(Пример)

Пример: комплексные числа

Множественные представления абстрактных данных

Декартово и полярное представление комплексных чисел





Большинство программ не заботятся о представлении

Некоторые арифметические операции проще в одном представлении, чем в другом

Реализация комплексной арифметики

Представим, что есть два класса для представления комплексных чисел

Число	Декартово представление	Полярное представление
$1+\sqrt{-1}$	**ComplexRI**(1, 1)	**ComplexMA**(sqrt(2), pi/4)

Пусть арифметические действия используют наиболее удобные представления

```
class Complex:
    def add(self, other):
        return ComplexRI(self.real + other.real,
                         self.imag + other.imag)
    def mul(self, other):
        return ComplexMA(self.magnitude * other.magnitude,
                         self.angle + other.angle)
```

Барьеры абстракции комплексной арифметики

Часть программы,которая…	Рассматривает комплексные числа как…	Используя…
Использует комплексные числа для вычислений	цельные значения	`x.add(y)`, `x.mul(y)`
Складывает комплексные числа	действительную и мнимую части	`real`, `imag`, `ComplexRI`
Перемножает комплексные числа	модуль и аргумент	`magnitude`, `angle`, `ComplexMA`

Реализация в объектной системе Python

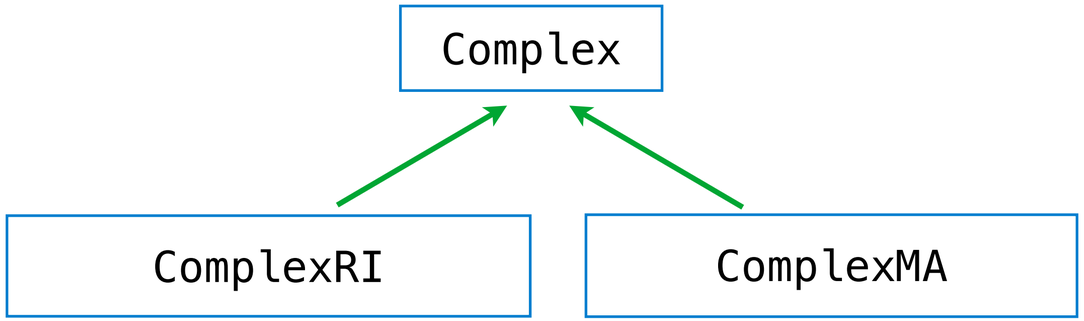
Реализация комплексных чисел

Интерфейс для комплексных чисел

Все комплексные числа должны иметь атрибуты `real` и `imag`

Все комплексные числа должны иметь атрибуты `magnitude` и `angle`

Все комплексные числа должны иметь общие реализации для `add` и `mul`



(Пример)

Декартово представление

```
class ComplexRI:

    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    @property
    def magnitude(self):
        return (self.real ** 2 + self.imag ** 2) ** 0.5

    @property
    def angle(self):
        return atan2(self.imag, self.real)

    def __repr__(self):
        return 'ComplexRI({0}, {1})'.format(self.real, self.imag)
```

Декоратор свойства: «Вызвать эту функцию при обращении к атрибуту»

math.atan2(y,x): Угол между осью x и точкой (x,y)

Декоратор @property позволяет вызывать безаргументный метод без использования синтаксиса вызывающего выражения. Эти методы выглядят как простые атрибуты.

Полярное представление

```
class ComplexMA:

    def __init__(self, magnitude, angle):
        self.magnitude = magnitude
        self.angle = angle

    @property
    def real(self):
        return self.magnitude * cos(self.angle)

    @property
    def imag(self):
        return self.magnitude * sin(self.angle)

    def __repr__(self):
        return 'ComplexMA({0}, {1})'.format(self.magnitude, self.angle)
```

Использование комплексных чисел

Любой тип комплексных чисел может быть аргументом `add` и `mul`:

```
class Complex:
    def add(self, other):
        return ComplexRI(self.real + other.real,
                         self.imag + other.imag)
    def mul(self, other):
        return ComplexMA(self.magnitude * other.magnitude,
                         self.angle + other.angle)
```

```
>>> from math import pi
>>> ComplexRI(1, 2).add(ComplexMA(2, pi/2))
ComplexRI(1.0000000000000002, 4.0)
>>> ComplexRI(0, 1).mul(ComplexRI(0, 1))
ComplexMA(1.0, 3.141592653589793)
```

$1 + 4 \cdot \sqrt{-1}$

-1