Lecture 13

Special Methods

String Representations

String Representations

- An object value should behave like the kind of data it is meant to represent
- For instance, by producing a string representation of itself

String Representations

- An object value should behave like the kind of data it is meant to represent
- For instance, by producing a string representation of itself

In Python, all objects produce **two** string representations:

- The **str** is easy for humans, to be readable
  - Informal representation
- The **repr** is easy for the Python interpreter
  - Official representation

String Representations

- An object value should behave like the kind of data it is meant to represent
- For instance, by producing a string representation of itself
- Strings are important: they represent language and programs

In Python, all objects produce **two** string representations:

- The **str** is easy for humans
- The **repr** is easy for the Python interpreter (expression in Python)

The **str** and **repr** strings are often the same, but not always

What is easier to understand? + demo

```
>>> a =datetime.datetime.now()
>>> a
datetime.datetime(2018, 2, 27, 15, 5, 59, 559961)
>>> str(a)
'2018-02-27 15:05:59.559961'
>>> repr(a)
'datetime.datetime(2018, 2, 27, 15, 5, 59, 559961)'
>>>
```

eval( ):

utility which lets a Python program run Python code within itself, by evaluating expressions.

- It is used on sites like `codepad.org` to allow you to execute scripts in a test environment.

```
eval(expression, globals=None, locals=None)
```

- **expression**: this string is parsed and evaluated as a Python expression
- **globals (optional):** a dictionary to specify the available global methods and variables.
- **locals (optional):** another dictionary to specify the available local methods and variables.

(quick demo)

The repr String for an Object

The **repr** function returns a Python expression (a *string*) that evaluates to an equal object: https://docs.python.org/3/library/functions.html#repr

The result of calling **repr** on a value is what Python prints in an interactive session

```
>>> 12e12
12000000000000.0
```

The repr String for an Object

The **repr** function returns a Python expression (a string) that evaluates to an equal object

The result of calling **repr** on a value is what Python **prints** in an interactive session

```
>>> 12e12
12000000000000.0
```

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

The repr String for an Object

The **repr** function returns a Python expression (a string) that evaluates to an equal object: https://docs.python.org/3/library/functions.html#repr

The result of calling repr on a value is what Python prints in an interactive session

Some objects do not have a simple Python-readable string

```
>>> 12e12
12000000000000.0
```

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

The repr String for an Object

The **repr** function returns a Python expression (a string) that evaluates to an equal object: https://docs.python.org/3/library/functions.html#repr

The result of calling repr on a value is what Python prints in an interactive session

Some objects do not have a simple Python-readable string

```
>>> 12e12
120000000000000.0
```

```
>>> print(repr(12e12))
120000000000000.0
```

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

The repr String for an Object

The **repr** function returns a Python expression (a string) that evaluates to an equal object: https://docs.python.org/3/library/functions.html#repr

Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to eval(), otherwise...

```
repr(object) -> string

For most object types, eval(repr(object)) == object.
```

The repr String for an Object

The **repr** function returns a Python expression (a string) that evaluates to an equal object: https://docs.python.org/3/library/functions.html#repr

Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to eval(), otherwise

```
repr(object) -> string

For most object types, eval(repr(object)) == object.
```

```
>>> name = "Marina"
>>> repr(name)
"'Marina'"
>>> eval(repr(name))
'Marina'
>>> eval(repr(name)) is name
True
```

The str String for an Object

Human interpretable strings are useful as well:

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

The str String for an Object

Human interpretable strings are useful as well:

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

The str String for an Object (demo)

Human interpretable strings are useful as well:

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

The result of calling **str** on the value of an expression is what Python prints using the print function:

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

Question: Output?

```
name = " ' Marina ' "
```

	eval(name)	eval(repr(name))	str(name)
A:	‘ Marina ’	‘ Marina ’	\” “ ‘ Marina ‘ “ \”
B:	Error	“ ‘ Marina ‘ “	‘ Marina ’
C:	Marina	Error	‘ Marina’
D:	“ ‘ Marina ‘ “	‘ Marina ’	“ ‘ Marina ‘ “
E:	‘ Marina ‘	“ ‘ Marina ‘ “	“ ‘ Marina ‘ “

Question

```
class Animal:
voice = lambda: "Grrrr"

    def __init__(self, type, sound):
        self.type = type

        self.voice = lambda: sound
```

Desired output:

woof-woof
Grrrr

```
a = Animal("dog", "woof-woof")
```

A: a.voice()
Animal.voice()

B: Animal.voice()
a.voice()

C: a.voice(“woof-woof”)
Animal.voice(“Grrrrr”)

D: Animal.voice(“woof-woof”)
a.voice(“Grrrrr”)

E: Not possible

Polymorphic Functions

Polymorphic Functions

Polymorphic function: A function that applies to many (*poly*) different forms (morph) of data

str and **repr** are both polymorphic; they apply to any object:

Polymorphic Functions

Polymorphic function: A function that applies to many (*poly*) different forms (morph) of data

str and **repr** are both polymorphic; they apply to any object:

repr asks an argument to display itself

invokes a zero-argument method **__repr__** on its argument

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

Polymorphic Functions

Polymorphic function: A function that applies to many (poly) different forms (morph) of data

str and **repr** are both polymorphic; they apply to any object:

repr invokes a zero-argument method **__repr__** on its argument

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

str invokes a zero-argument method __str__ on its argument

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

Implementing repr and str

The behavior of **repr** is slightly more complicated than invoking **__repr__** on its argument:

- An **instance** attribute called __repr__ is **ignored**! Only **class** attributes are found

How would we implement this behavior? Which of the following function definitions corresponds to a function **repr** that takes in some argument, looks up the class attribute called **__repr__** and invokes it?

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

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

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

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

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

Implementing repr and str

The behavior of **str** is also complicated:

- An instance attribute called **__str__** is ignored
- If no **__str__** attribute is found, uses repr string
- (By the way, **str** is a class, not a function)
    - When you call str, you call a constructor for built-in string type called str

Question: How would we implement this behavior?
(demo)

What will be printed?

```
class Bear:
    """A Bear."""
    def __init__(self):
        self.__repr__ = lambda: 'misha'
        self.__str__ = lambda: 'misha the bear'

    def __repr__(self):
        return 'Bear()'

    def __str__(self):
        return 'a bear'
```

```
misha = Bear()
print(misha)
print(repr(misha))
print(str(misha))
print(misha.__repr__())
print(misha.__str__())
```

Special Method Names

Special Method Names in Python

Certain names are special because they have **built-in** behavior. Always start and end with two *underscores*

`__init__`	Method invoked automatically when an object is constructed
`__repr__`	Method invoked to display an object as a Python expression
`__add__`	Method invoked to add one object to another
`__bool__`	Method invoked to convert an object to True or False
`__float__`	Method invoked to convert an object to a float (real number)

Special Method Names in Python

Certain names are special because they have built-in behavior. Always start and end with two underscores

`__init__`	Method invoked automatically when an object is constructed
`__repr__`	Method invoked to display an object as a Python expression
`__add__`	Method invoked to add one object to another
`__bool__`	Method invoked to convert an object to True or False
`__float__`	Method invoked to convert an object to a float (real number)

```
>>> zero, one, two = 0, 1, 2
>>> one + two
3
```

Special Method Names in Python

Certain names are special because they have built-in behavior. Always start and end with two underscores

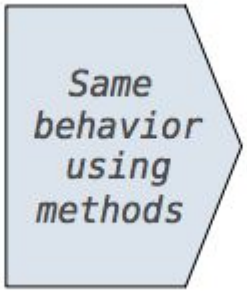
`__init__`	Method invoked automatically when an object is constructed
`__repr__`	Method invoked to display an object as a Python expression
`__add__`	Method invoked to add one object to another
`__bool__`	Method invoked to convert an object to True or False
`__float__`	Method invoked to convert an object to a float (real number)

```
>>> zero, one, two = 0, 1, 2
>>> one + two
3
```

```
>>> bool(zero), bool(one)
(False, True)
```

`__init__`	Method invoked automatically when an object is constructed
`__repr__`	Method invoked to display an object as a Python expression
`__add__`	Method invoked to add one object to another
`__bool__`	Method invoked to convert an object to True or False
`__float__`	Method invoked to convert an object to a float (real number)

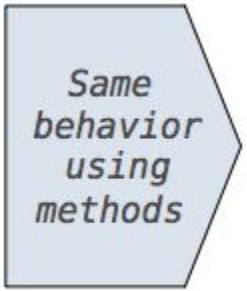
```
>>> zero, one, two = 0, 1, 2
>>> one + two
3
>>> bool(zero), bool(one)
(False, True)
```




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

`__init__`	Method invoked automatically when an object is constructed
`__repr__`	Method invoked to display an object as a Python expression
`__add__`	Method invoked to add one object to another
`__bool__`	Method invoked to convert an object to True or False
`__float__`	Method invoked to convert an object to a float (real number)

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




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

Example:

https://www.python-course.eu/python3_magic_methods.php

```
class UnusualStr(str):

    def __add__(self, other):

        reverse_other = other[::-1]
        return str.__add__(self, reverse_other) #calling + from str class



str1 = UnusualStr("Marina")
str2 = UnusualStr("Langlois")

print(str1+str2)
```

Example for __add__

```
class UnusualStr(str):

    def __add__(self, other):

        reverse_other = other[::-1]
        return str.__add__(self, reverse_other) #calling + from str class


str1 = UnusualStr("Marina")
str2 = UnusualStr("Langlois")

print(str1+str2)
```

```
MarinasiolgnaL
```

Special Methods:

Adding instances of user-defined classes invokes either the `__add__` or `__radd__` method

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

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

Special Methods:

Adding instances of user-defined classes invokes either the `__add__` or `__radd__` method

```
>>> 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)
```

http://getpython3.com/diveintopython3/special-method-names.html

https://docs.python.org/3/reference/datamodel.html#special-method-names