Special Methods

- Polymorphism
- Polymorphic Functions (__str__, __repr__)
- Operator Overloading (+ and __add__)
- More Special Methods

Polymorphism

Polymorphism

- Ad Hoc Polymorphism

- Parametric Polymorphism

- Inclusion Polymorphism

Polymorphism

- Ad Hoc Polymorphism

  e.g., Overloading:

  ```
  foo(int) { xxx }
  foo(string) {xx xxx xx}
  ```

- Parametric Polymorphism

- Inclusion Polymorphism

Polymorphism

- Ad Hoc Polymorphism

e.g., Overloading:

```
foo(int) { xxx }

foo(string) {xx xxx xx}
```

- Parametric Polymorphism

e.g., Generic functions:

```
Template <typename T>
T foo(T x, T y) {
  return (x > y) ? x : y;
}
```

```
foo<int>(3,7)
foo<char>('h','k')
```

- Inclusion Polymorphism

Polymorphism

- Ad Hoc Polymorphism

e.g., Overloading:

```
foo(int) { xxx }

foo(string) {xx xxx xx}
```

- Parametric Polymorphism

e.g., Generic functions:

```
Template <typename T>
T foo(T x, T y) {
  return (x > y) ? x : y;
}
```

```
foo<int>(3,7)
foo<char>('h','k')
```

- Inclusion Polymorphism

Subtypes and inheritance:

```
T v; // T has many subtypes
… …
v.foo();
```

Polymorphism

- Ad Hoc Polymorphism

Next, we introduce two instances of ad hoc polymorphism to help illustrate some important *special methods* in Python:
polymorphic function (__str__, __repr__)
operator overloading (__add__)

- Parametric Polymorphism

e.g., Generic functions:

```
Template <typename T>
T foo(T x, T y) {
  return (x > y) ? x : y;
}
```

```
foo<int>(3,7)
foo<char>('h','k')
```

- Inclusion Polymorphism

Subtypes and inheritance:

```
T v; // T has many subtypes
… …
v.foo();
```

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

Strings are important: they represent language and programs

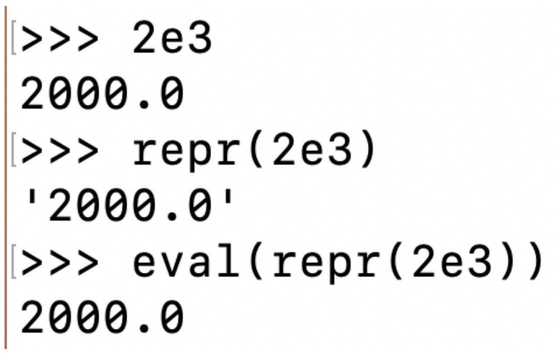
In Python, all objects produce two string representations:

- The **str** is legible to humans
- The **repr** is legible to the Python interpreter

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

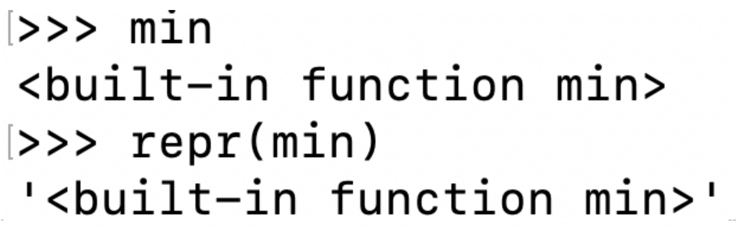
The repr String for an Object

repr: string representation of Python object. For most object types, eval will convert it back to that object, eval(repr(obj)) == obj



```
>>> 2e3
2000.0
>>> repr(2e3)
'2000.0'
>>> eval(repr(2e3))
2000.0
```

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



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

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

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

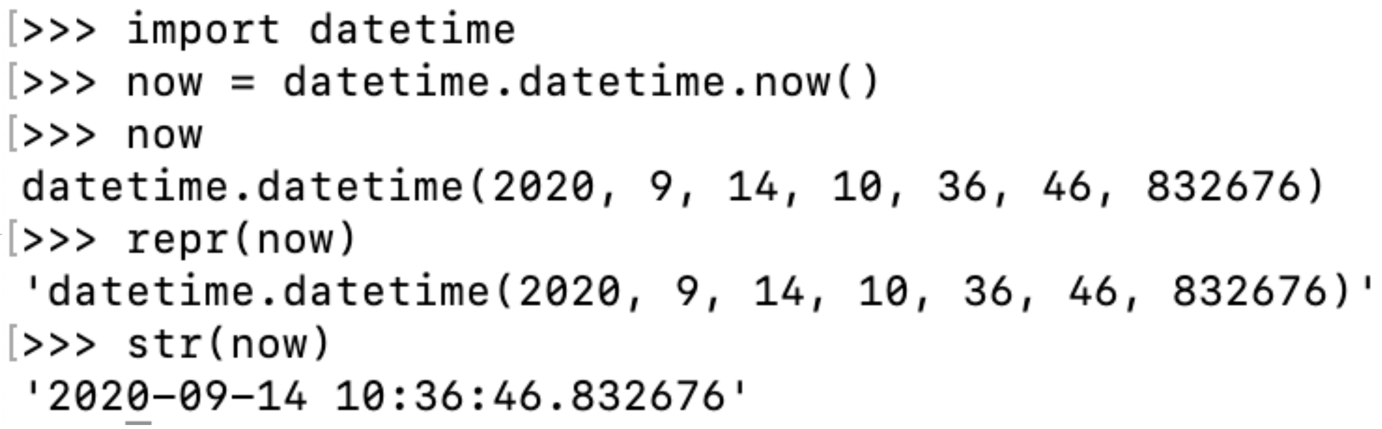
The str String for an Object

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

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



```
[>>> import datetime
[>>> now = datetime.datetime.now()
[>>> now
 datetime.datetime(2020, 9, 14, 10, 36, 46, 832676)
[>>> repr(now)
 'datetime.datetime(2020, 9, 14, 10, 36, 46, 832676)'
[>>> str(now)
 '2020-09-14 10:36:46.832676'
```

The str String for an Object

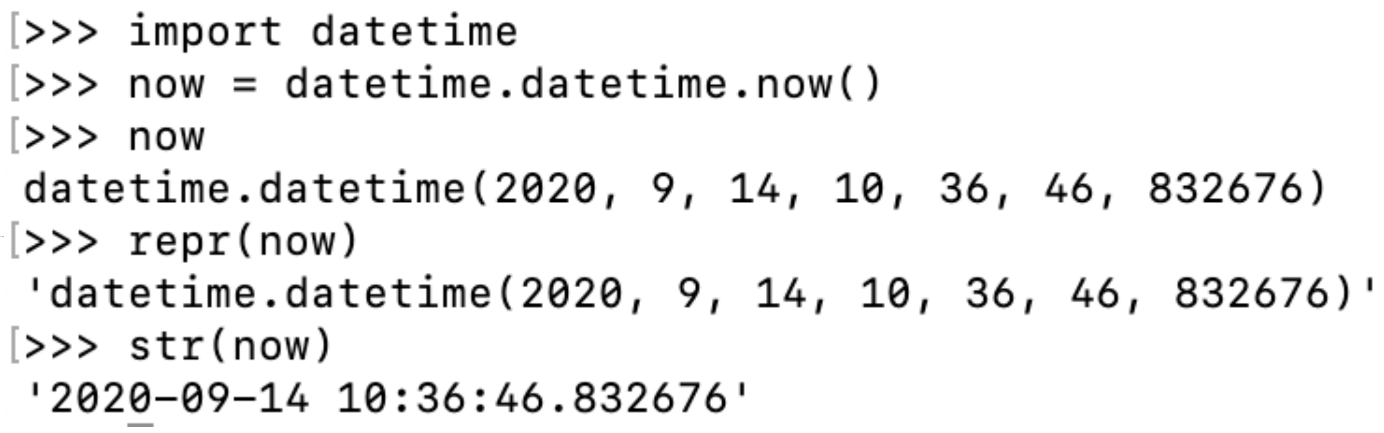
Human interpretable strings are useful as well:

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

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

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

repr is to be unambiguous
str is to be readable



```
[>>> import datetime
[>>> now = datetime.datetime.now()
[>>> now
datetime.datetime(2020, 9, 14, 10, 36, 46, 832676)
[>>> repr(now)
'datetime.datetime(2020, 9, 14, 10, 36, 46, 832676)'
[>>> str(now)
'2020-09-14 10:36:46.832676'
```

Polymorphic Functions

Polymorphic Functions

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

Polymorphic Functions

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

Polymorphic functions behave differently depending on the types of the arguments come in, while **parametric functions** execute the same code for arguments of any admissible types

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 and do not have much logic, and they defer to the object (comes in) to decide what to do

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 and do not have much logic, and they defer to the object (comes in) to decide what to do

repr invokes a zero-argument method __repr__ on its argument

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 and do not have much logic, and they defer to the object (comes in) to decide what to do

repr invokes a zero-argument method __repr__ on its argument

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

```
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> repr(half)
'Fraction(1, 2)'
>>> str(half)
'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 and do not have much logic, and they defer to the object (comes in) to decide what to do

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'
```

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

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
- *Question*: How would we implement this behavior?

```
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__()
```

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
- *Question*: How would we implement this behavior?

```
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__()
```

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
- *Question*: How would we implement this behavior?

```
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__()
```

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

- An instance attribute called __str__ is ignored
- If no __str__ attribute is found, uses **repr** string
- *Question*: How would we implement this behavior?

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
- *Question*: How would we implement this behavior?

```
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__()
```

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

- An instance attribute called __str__ is ignored
- If no __str__ attribute is found, uses **repr** string
- *Question*: How would we implement this behavior?

demo_1

Operator Overloading

Operator Overloading

Operator overloading is to give the operator extended meaning beyond its predefined operational meaning.

Operator Overloading

Operator overloading is to give the operator extended meaning beyond its predefined operational meaning.

e.g., adding instances of user-defined classes invokes **__add__** method

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

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

Operator Overloading

Operator overloading is to give the operator extended meaning beyond its predefined operational meaning.

e.g., adding instances of user-defined classes invokes **__add__** method

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

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

Operator '+' is **overloaded** by __add__ method when '+' is used to add user-defined objects

Operator Overloading

Operator overloading is to give the operator extended meaning beyond its predefined operational meaning.

e.g., adding instances of user-defined classes invokes **__add__** method

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

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

Operator '+' is **overloaded** by __add__ method when '+' is used to add user-defined objects

A + B, different behaviors of this adding expression may exhibit, depending on the types of the operands (A or B). Thus we say **operator overloading** is a kind of **polymorphism**.

Operator Overloading

Operator overloading is to give the operator extended meaning beyond its predefined operational meaning.

e.g., adding instances of user-defined classes invokes **__add__** method

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

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

Operator '+' is **overloaded** by __add__ method when '+' is used to add user-defined objects

A + B, different behaviors of this adding expression may exhibit, depending on the types of the operands (A or B). Thus we say **operator overloading** is a kind of **polymorphism**.

demo_2

Special Method Names in Python (Summary)

Certain names are special because they have built-in behaviors
These names always start and end with two underscores

__init__	Method invoked automatically when an object is constructed
__repr/str__	Method invoked to display an object as a Python expression
__add/radd__	Method invoked to add one object to another
__float__	Method invoked to convert an object to a float (real number)

More Special Methods:

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