61A Lecture 16
Announcements
String Representations
String Representations
String Representations

An object value should behave like the kind of data it is meant to represent
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.

Strings are important: they represent language and programs.
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:
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.
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.
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
The repr String for an Object

The `repr` function returns a Python expression (a string) that evaluates to an equal object.
The `repr` String for an Object

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

```
repr(object) -> string
```

Return the canonical string representation of the object.
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.

```python
repr(object) -> string
```

Return the canonical string representation of the object. For most object types, `eval(repr(object)) == object`.

The result of calling `repr` on a value is what Python prints in an interactive session.
The `repr` String for an Object

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

```
repr(object) -> string
```

Return the canonical string representation of the object. For most object types, `eval(repr(object)) == object`.

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

```python
>>> 12e12
```
The repr String for an Object

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

```python
repr(object) -> string
```

Return the canonical string representation of the object. For most object types, `eval(repr(object)) == object`.

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

```python
>>> 12e12
12000000000000.0
```
The repr String for an Object

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

```python
repr(object) -> string
```

Return the canonical string representation of the object.
For most object types, `eval(repr(object)) == object`.

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

```python
>>> 12e12
12000000000000.0
>>> print(repr(12e12))
```
The repr String for an Object

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

`repr(object) -> string`

Return the canonical string representation of the object. For most object types, `eval(repr(object)) == object`.

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

```python
>>> 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:

\[
\text{repr(object)} \rightarrow \text{string}
\]

Return the canonical string representation of the object. For most object types, \(\text{eval(repr(object))} == \text{object}\).

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

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

Some objects do not have a simple Python-readable string.
The repr String for an Object

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

\[
\text{repr}(\text{object}) \rightarrow \text{string}
\]

Return the canonical string representation of the object. For most object types, \(\text{eval}(	ext{repr}(	ext{object})) == \text{object}\).

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

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

Some objects do not have a simple Python-readable string

```python
>>> repr(min)
'\text{<built-in function min>}'
```
The str String for an Object

Human interpretable strings are useful as well:
The str String for an Object

Human interpretable strings are useful as well:

```python
>>> from fractions import Fraction
```
The `str` String for an Object

Human interpretable strings are useful as well:

```python
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
```
The str String for an Object

Human interpretable strings are useful as well:

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

The str String for an Object

Human interpretable strings are useful as well:

```python
>>> 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:

```python
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> repr(half)
'Fraction(1, 2)'
>>> str(half)
6
```
The str String for an Object

Human interpretable strings are useful as well:

```python
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> repr(half)
'Fraction(1, 2)'
>>> str(half)
'1/2'
```
The str String for an Object

Human interpretable strings are useful as well:

```python
>>> 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:
The str String for an Object

Human interpretable strings are useful as well:

```python
>>> 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:

```python
>>> print(half)
```
The str String for an Object

Human interpretable strings are useful as well:

```python
>>> 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:

```python
>>> print(half)
1/2
```
The str String for an Object

Human interpretable strings are useful as well:

```python
>>> 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:

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

(Demo)
Discussion

Human interpretable strings are useful as well:
Discussion

Human interpretable strings are useful as well:

```python
>>> from fractions import Fraction
```
Discussion

Human interpretable strings are useful as well:

```python
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
```
Human interpretable strings are useful as well:

```python
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> repr(today)
```
Human interpretable strings are useful as well:

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

Human interpretable strings are useful as well:

```python
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> repr(today)
'Fraction(1, 2)'
>>> str(today)
7
```
Discussion

Human interpretable strings are useful as well:

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

Human interpretable strings are useful as well:

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

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

Human interpretable strings are useful as well:

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

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

```python
>>> print(half)
```
Human interpretable strings are useful as well:

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

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

```python
def demo():
    >>> print(half)
    1/2
```
Discussion

Human interpretable strings are useful as well:

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

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

```python
>>> print(half)
1/2
```
Polymorphic Functions
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. `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` 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.

`repr` invokes a zero-argument method `__repr__` on its argument.

```python
>>> 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

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

`str` invokes a zero-argument method `__str__` 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

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

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

`str` invokes a zero-argument method `__str__` on its argument

```python
>>> half.__str__()
'1/2'
```
Implementing repr and str
Implementing repr and str

The behavior of `repr` is slightly more complicated than invoking `__repr__` on its argument:
Implementing \texttt{repr} and \texttt{str}

The behavior of \texttt{repr} is slightly more complicated than invoking \texttt{\_\_repr\_\_} on its argument:

- An instance attribute called \texttt{\_\_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?
Implementing repr and str

The behavior of \texttt{repr} is slightly more complicated than invoking \texttt{__repr__} on its argument:

- An instance attribute called \texttt{__repr__} is ignored! Only class attributes are found
- \textit{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 super(x).__repr__(x)
```
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)
```

The behavior of `str` is also complicated:

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

\[
\begin{aligned}
\text{def repr(x):} & \quad \text{return } x._{\text{__repr__}}(x) \\
\text{def repr(x):} & \quad \text{return } x._{\text{__repr__}}() \\
\text{def repr(x):} & \quad \text{return type(x)._{\text{__repr__}}(x)} \\
\text{def repr(x):} & \quad \text{return } \text{super}(x)._{\text{__repr__}}() \\
\end{aligned}
\]

The behavior of `str` is also complicated:

- An instance attribute called `__str__` is ignored
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 type(x).__repr__(x)
```

The behavior of `str` is also complicated:

- An instance attribute called `__str__` is ignored
- If no `__str__` attribute is found, uses `repr` string

```
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 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?

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

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?

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

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

```python
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)
```

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?
- `str` is a class, not a function

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

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

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

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

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?
- `str` is a class, not a function

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

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

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

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

(Demo)
Interfaces
Interfaces

Message passing: Objects interact by looking up attributes on each other (passing messages)
Interfaces

**Message passing:** Objects interact by looking up attributes on each other (passing messages).

The attribute look-up rules allow different data types to respond to the same message.
Interfaces

**Message passing:** Objects interact by looking up attributes on each other (passing messages)

The attribute look-up rules allow different data types to respond to the same message

A shared message (attribute name) that elicits similar behavior from different object classes is a powerful method of abstraction
Interfaces

Message passing: Objects interact by looking up attributes on each other (passing messages)

The attribute look-up rules allow different data types to respond to the same message.

A shared message (attribute name) that elicits similar behavior from different object classes is a powerful method of abstraction.

An interface is a set of shared messages, along with a specification of what they mean.
Interfaces

**Message passing**: Objects interact by looking up attributes on each other (passing messages).

The attribute look-up rules allow different data types to respond to the same message.

A *shared message* (attribute name) that elicits similar behavior from different object classes is a powerful method of abstraction.

An interface is a set of shared messages, along with a specification of what they mean.

**Example:**
Interfaces

**Message passing:** Objects interact by looking up attributes on each other (passing messages)

The attribute look-up rules allow different data types to respond to the same message.

A *shared message* (attribute name) that elicits similar behavior from different object classes is a powerful method of abstraction.

An interface is a set of shared messages, along with a specification of what they mean.

**Example:**

Classes that implement `__repr__` and `__str__` methods that return Python-interpretable and human-readable strings implement an interface for producing string representations.
Interfaces

**Message passing:** Objects interact by looking up attributes on each other (passing messages)

The attribute look-up rules allow different data types to respond to the same message

A **shared message** (attribute name) that elicits similar behavior from different object classes is a powerful method of abstraction

An interface is a set of shared messages, along with a specification of what they mean

**Example:**

Classes that implement `__repr__` and `__str__` methods that return Python-interpretable and human-readable strings implement an interface for producing string representations

(Demo)
Special Method Names
Special Method Names in Python

Certain names are special because they have built-in behavior
Special Method Names in Python

Certain names are special because they have built-in behavior

These names always start and end with two underscores
Special Method Names in Python

Certain names are special because they have built-in behavior

These names always start and end with two underscores

```
__init__
```
Special Method Names in Python

Certain names are special because they have built-in behavior

These names always start and end with two underscores

__init__ Method invoked automatically when an object is constructed
Special Method Names in Python

Certain names are special because they have built-in behavior.

These names always start and end with two underscores:

- `__init__` Method invoked automatically when an object is constructed.
- `__repr__`
Special Method Names in Python

Certain names are special because they have built-in behavior.

These names 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.
Special Method Names in Python

Certain names are special because they have built-in behavior

These names 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__`
Special Method Names in Python

Certain names are special because they have built-in behavior.

These names 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
Special Method Names in Python

Certain names are special because they have built-in behavior.

These names 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__**:
Special Method Names in Python

Certain names are special because they have built-in behavior.

These names 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 a boolean value.
- `__float__`: Method invoked to convert an object to a float.
Special Method Names in Python

Certain names are special because they have built-in behavior

These names 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
Special Method Names in Python

Certain names are special because they have built-in behavior.

These names 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

These names 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
Special Method Names in Python

Certain names are special because they have built-in behavior

These names always start and end with two underscores

```python
__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)
```

```python
>>> zero, one, two = 0, 1, 2
>>> one + two
3
```
Special Method Names in Python

Certain names are special because they have built-in behavior.

These names 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).

```python
>>> zero, one, two = 0, 1, 2
>>> one + two
3
>>> bool(zero), bool(one)
(False, True)
```
Special Method Names in Python

Certain names are special because they have built-in behavior.

These names 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).

```python
>>> zero, one, two = 0, 1, 2
>>> one + two
3
>>> bool(zero), bool(one)
(False, True)
```
Special Method Names in Python

Certain names are special because they have built-in behavior.

These names 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).

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

```python
>>> zero, one, two = 0, 1, 2
```

Same behavior using methods.
Special Method Names in Python

Certain names are special because they have built-in behavior

These names 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)

```python
>>> zero, one, two = 0, 1, 2
>>> one + two
3
>>> bool(zero), bool(one)
(False, True)
```
Special Method Names in Python

Certain names are special because they have built-in behavior

These names 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)

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

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

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

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

```python
>>> Ratio(1, 3) + Ratio(1, 6)
Ratio(1, 2)
```
Special Methods

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

```python
>>> 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.

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

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

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

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

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

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

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


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

Adding instances of user-defined classes invokes either the \_\_add\_\_ or \_\_radd\_\_ method

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

(Demo)
Generic Functions
Generic Functions

**Goal:** Write a function that operates on two or more arguments of different types

**Type Dispatching:** Inspect the type of an argument in order to select behavior

**Type Coercion:** Convert one value to match the type of another
Generic Functions

**Goal:** Write a function that operates on two or more arguments of different types

**Type Dispatching:** Inspect the type of an argument in order to select behavior

**Type Coercion:** Convert one value to match the type of another

(Demo)
Property Methods
Property Methods

Often, we want the value of instance attributes to stay in sync.

For example, what if we wanted a Ratio to keep its proportion when its numerator changes.
Property Methods

Often, we want the value of instance attributes to stay in sync.

For example, what if we wanted a Ratio to keep its proportion when its numerator changes:

```python
>>> f = Ratio(3, 5)
```

![Ratio Visual](image-url)
Property Methods

Often, we want the value of instance attributes to stay in sync.

For example, what if we wanted a Ratio to keep its proportion when its numerator changes?

```python
>>> f = Ratio(3, 5)
>>> f.gcd
1

\[
\frac{3}{5}
\]
Property Methods

Often, we want the value of instance attributes to stay in sync.

For example, what if we wanted a Ratio to keep its proportion when its numerator changes.

```python
>>> f = Ratio(3, 5)
>>> f.gcd
1
>>> f.numer = 6
```
Often, we want the value of instance attributes to stay in sync

For example, what if we wanted a Ratio to keep its proportion when its numerator changes

```python
>>> f = Ratio(3, 5)
>>> f.gcd
1
>>> f.numer = 6
```

```
6
3
10
```
Property Methods

Often, we want the value of instance attributes to stay in sync.

For example, what if we wanted a Ratio to keep its proportion when its numerator changes:

```python
>>> f = Ratio(3, 5)
>>> f.gcd
1
>>> f.numer = 6
>>> f.denom
10
```
Property Methods

Often, we want the value of instance attributes to stay in sync.

For example, what if we wanted a Ratio to keep its proportion when its numerator changes:

```python
>>> f = Ratio(3, 5)
>>> f.gcd
1
>>> f.numer = 6
>>> f.denom
10
>>> f.gcd
2
```
Property Methods

Often, we want the value of instance attributes to stay in sync.

For example, what if we wanted a Ratio to keep its proportion when its numerator changes:

```python
goldenRatio = Ratio(3, 5)
goldenRatio.gcd  # 1
>>> goldenRatio.numer = 6
>>> goldenRatio.denom
10
>>> goldenRatio.gcd
```

No method calls!
Property Methods

Often, we want the value of instance attributes to stay in sync

For example, what if we wanted a Ratio to keep its proportion when its numerator changes

```python
>>> f = Ratio(3, 5)
>>> f.gcd
1
>>> f.numer = 6
>>> f.denom
10
>>> f.gcd
2
```

The @property decorator on a method designates that it will be called whenever it is looked up on an instance
Property Methods

Often, we want the value of instance attributes to stay in sync.

For example, what if we wanted a Ratio to keep its proportion when its numerator changes.

```python
>>> f = Ratio(3, 5)
>>> f.gcd
1
>>> f.numer = 6
>>> f.denom
10
>>> f.gcd
2
```

The `@property` decorator on a method designates that it will be called whenever it is looked up on an instance.

A `@<attribute>.setter` decorator on a method designates that it will be called whenever that attribute is assigned. `<attribute>` must be an existing property method.
**Property Methods**

Often, we want the value of instance attributes to stay in sync.

For example, what if we wanted a Ratio to keep its proportion when its numerator changes.

```python
gf = Ratio(3, 5)
gf.gcd
1
>>> f.numer = 6
f.denom
10
>>> f.gcd
2
```

The `@property` decorator on a method designates that it will be called whenever it is looked up on an instance.

A `@<attribute>`.setter decorator on a method designates that it will be called whenever that attribute is assigned. `<attribute>` must be an existing property method.

(Demo)