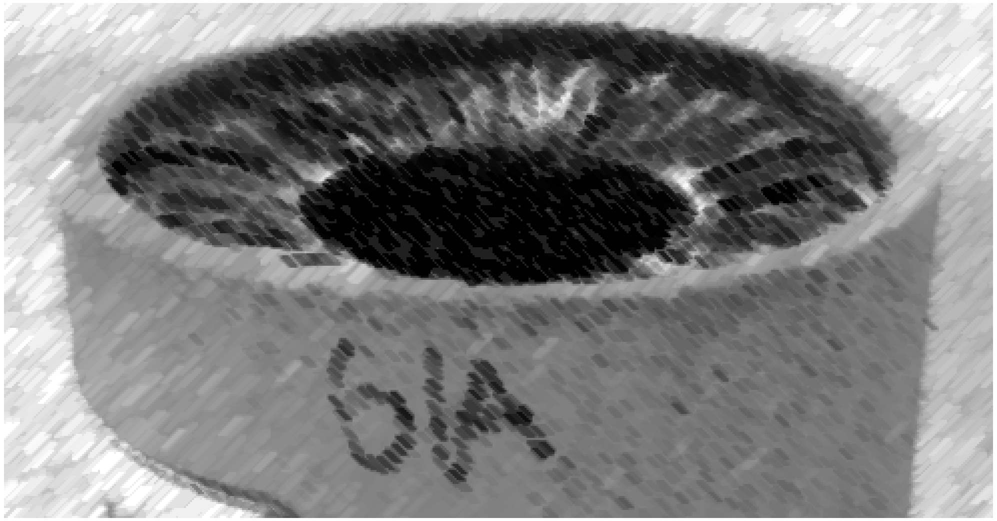


CS61A Lecture 21

Amir Kamil

UC Berkeley

March 11, 2013

Announcements

- HW7 due on Wednesday
- Ants project out

Looking Up Names

Looking Up Names

```
class CheckingAccount(Account):
    withdraw_fee = 1
    def withdraw(self, amount):
        return Account.withdraw(self,
                                amount + withdraw_fee)
```

Looking Up Names

Name expressions look up names in the environment

```
class CheckingAccount(Account):
    withdraw_fee = 1
    def withdraw(self, amount):
        return Account.withdraw(self,
                                amount + withdraw_fee)
```

Looking Up Names

Name expressions look up names in the environment

<name>

```
class CheckingAccount(Account):
    withdraw_fee = 1
    def withdraw(self, amount):
        return Account.withdraw(self,
                                amount + withdraw_fee)
```

Looking Up Names

Name expressions look up names in the environment

<name>

Dot expressions look up names in an object

```
class CheckingAccount(Account):
    withdraw_fee = 1
    def withdraw(self, amount):
        return Account.withdraw(self,
                                amount + withdraw_fee)
```

Looking Up Names

Name expressions look up names in the environment

```
<name>
```

Dot expressions look up names in an object

```
<expression> . <name>
```

```
class CheckingAccount(Account):
    withdraw_fee = 1
    def withdraw(self, amount):
        return Account.withdraw(self,
                                amount + withdraw_fee)
```

Looking Up Names

Name expressions look up names in the environment

```
<name>
```

Dot expressions look up names in an object

```
<expression> . <name>
```

```
class CheckingAccount(Account):
    withdraw_fee = 1
    def withdraw(self, amount):
        return Account.withdraw(self,
                                amount + withdraw_fee)
```

Error: **withdraw_fee** not bound in environment

Looking Up Names

Name expressions look up names in the environment

<name>

Dot expressions look up names in an object

<expression> . <name>

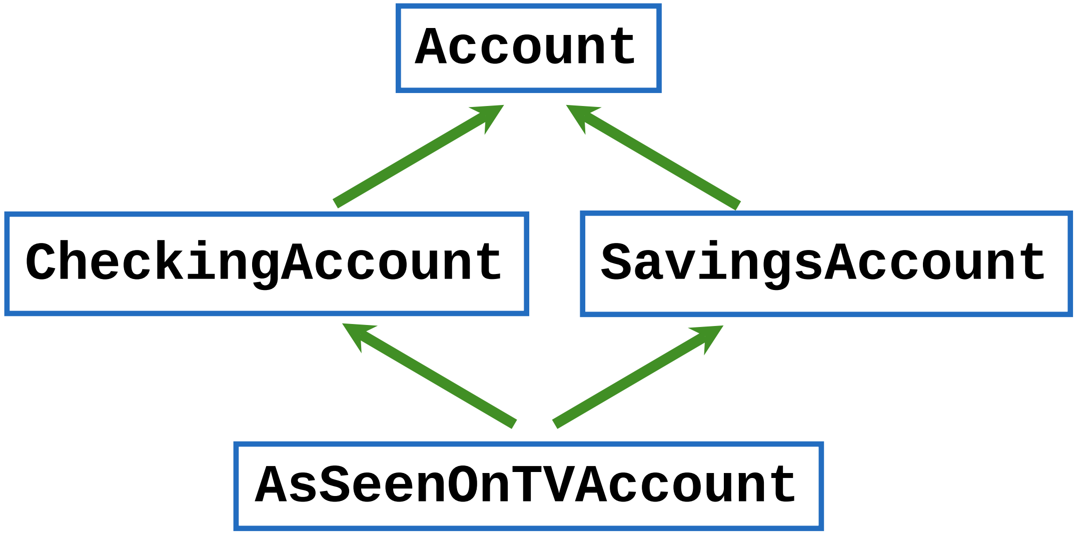
```
class CheckingAccount(Account):
    withdraw_fee = 1
    def withdraw(self, amount):
        return Account.withdraw(self,
                                amount + withdraw_fee)
```

Error: **withdraw_fee** not bound in environment

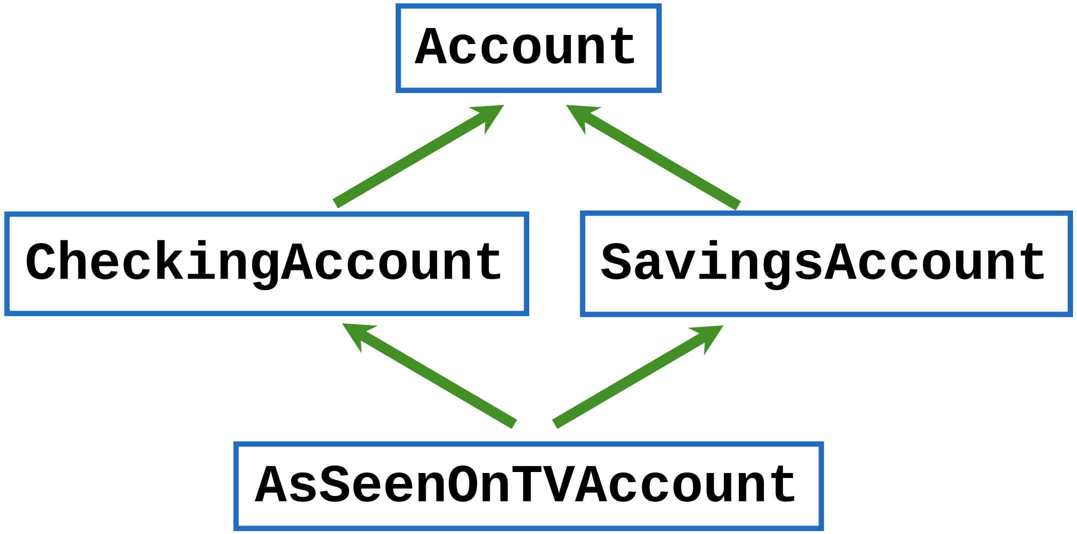
Not all languages work this way

Resolving Ambiguous Class Attribute Names

Resolving Ambiguous Class Attribute Names

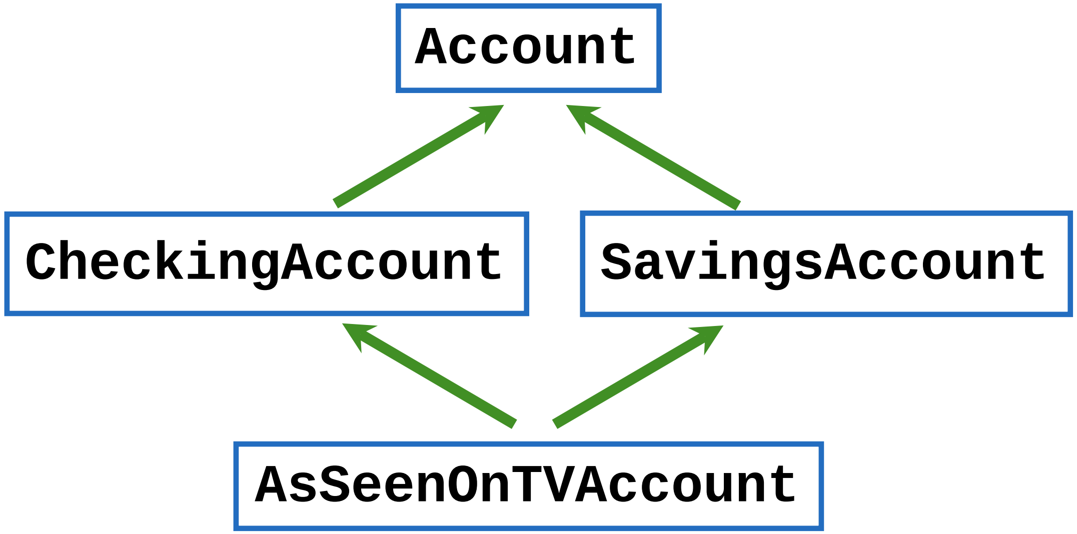


Resolving Ambiguous Class Attribute Names



Methods looked up from bottom to top, left to right

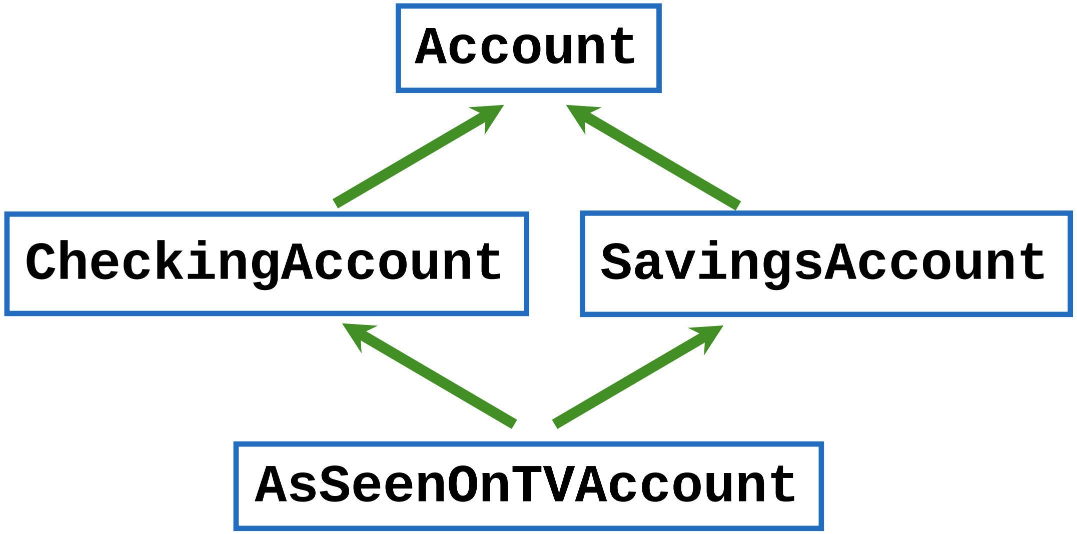
Resolving Ambiguous Class Attribute Names



Methods looked up from bottom to top, left to right

The **`mro`** method on a class lists the order in which classes are checked for attributes

Resolving Ambiguous Class Attribute Names

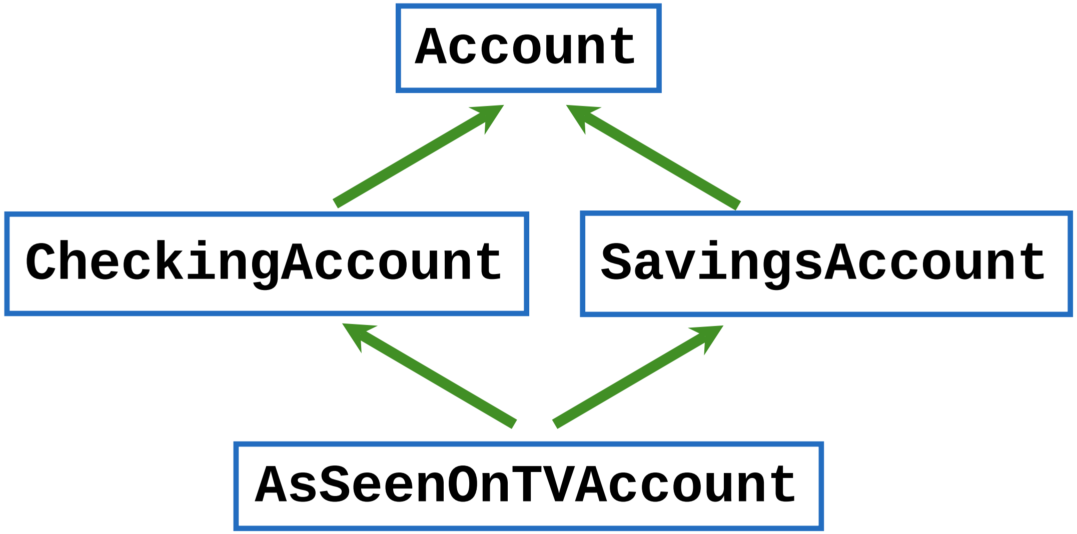


Methods looked up from bottom to top, left to right

The **`mro`** method on a class lists the order in which classes are checked for attributes

```
>>> [c.__name__ for c in AsSeenOnTVAccount.mro()]
['AsSeenOnTVAccount', 'CheckingAccount',
'SavingsAccount', 'Account', 'object']
```

Resolving Ambiguous Class Attribute Names



Methods looked up from bottom to top, left to right

The **`mro`** method on a class lists the order in which classes are checked for attributes

```
>>> [c.__name__ for c in AsSeenOnTVAccount.mro()]
['AsSeenOnTVAccount', 'CheckingAccount',
'SavingsAccount', 'Account', 'object']
```

OOP Odds and Ends

OOP Odds and Ends

The **`object`** class is at the root of the inheritance hierarchy

OOP Odds and Ends

The **`object`** class is at the root of the inheritance hierarchy

- **`object`** should be given as the base class when no other meaningful base class exists

OOP Odds and Ends

The **`object`** class is at the root of the inheritance hierarchy

- **`object`** should be given as the base class when no other meaningful base class exists

Class names should be in CamelCase

OOP Odds and Ends

The **`object`** class is at the root of the inheritance hierarchy

- **`object`** should be given as the base class when no other meaningful base class exists

Class names should be in CamelCase

Error messages can be confusing when calling methods with the wrong number of arguments:

OOP Odds and Ends

The **`object`** class is at the root of the inheritance hierarchy

- **`object`** should be given as the base class when no other meaningful base class exists

Class names should be in CamelCase

Error messages can be confusing when calling methods with the wrong number of arguments:

```
>>> tom_account = Account('Tom')
```

OOP Odds and Ends

The **`object`** class is at the root of the inheritance hierarchy

- **`object`** should be given as the base class when no other meaningful base class exists

Class names should be in CamelCase

Error messages can be confusing when calling methods with the wrong number of arguments:

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100, 200)
```

OOP Odds and Ends

The **`object`** class is at the root of the inheritance hierarchy

- **`object`** should be given as the base class when no other meaningful base class exists

Class names should be in CamelCase

Error messages can be confusing when calling methods with the wrong number of arguments:

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100, 200)
TypeError: deposit() takes exactly 2 positional arguments (3 given)
```

OOP Odds and Ends

The **`object`** class is at the root of the inheritance hierarchy

- **`object`** should be given as the base class when no other meaningful base class exists

Class names should be in CamelCase

Error messages can be confusing when calling methods with the wrong number of arguments:

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100, 200)
TypeError: deposit() takes exactly 2 positional arguments (3 given)
```

Compare to partially curried function:

OOP Odds and Ends

The **`object`** class is at the root of the inheritance hierarchy

- **`object`** should be given as the base class when no other meaningful base class exists

Class names should be in CamelCase

Error messages can be confusing when calling methods with the wrong number of arguments:

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100, 200)
TypeError: deposit() takes exactly 2 positional arguments (3 given)
```

Compare to partially curried function:

```
>>> add3 = curry(add)(3)
```

OOP Odds and Ends

The **`object`** class is at the root of the inheritance hierarchy

- **`object`** should be given as the base class when no other meaningful base class exists

Class names should be in CamelCase

Error messages can be confusing when calling methods with the wrong number of arguments:

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100, 200)
TypeError: deposit() takes exactly 2 positional arguments (3 given)
```

Compare to partially curried function:

```
>>> add3 = curry(add)(3)
>>> add3(4, 5)
```

OOP Odds and Ends

The **`object`** class is at the root of the inheritance hierarchy

- **`object`** should be given as the base class when no other meaningful base class exists

Class names should be in CamelCase

Error messages can be confusing when calling methods with the wrong number of arguments:

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100, 200)
TypeError: deposit() takes exactly 2 positional arguments (3 given)
```

Compare to partially curried function:

```
>>> add3 = curry(add)(3)
>>> add3(4, 5)
TypeError: op_add expected 2 arguments, got 3
```

Generic Functions

Generic Functions

An abstraction might have more than one representation.

Generic Functions

An abstraction might have more than one representation.

- Python has many sequence types: tuples, ranges, lists, etc.

Generic Functions

An abstraction might have more than one representation.

- Python has many sequence types: tuples, ranges, lists, etc.

An abstract data type might have multiple implementations.

Generic Functions

An abstraction might have more than one representation.

- Python has many sequence types: tuples, ranges, lists, etc.

An abstract data type might have multiple implementations.

- Some representations are better suited to some problems

Generic Functions

An abstraction might have more than one representation.

- Python has many sequence types: tuples, ranges, lists, etc.

An abstract data type might have multiple implementations.

- Some representations are better suited to some problems

A function might want to operate on multiple data types.

Generic Functions

An abstraction might have more than one representation.

- Python has many sequence types: tuples, ranges, lists, etc.

An abstract data type might have multiple implementations.

- Some representations are better suited to some problems

A function might want to operate on multiple data types.

Message passing enables us to accomplish all of the above, as we will see today and next time

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**.

When the “str” and “repr” **strings are the same**, that’s evidence that a programming language is legible by humans!

The “repr” String for an Object

The “repr” String for an Object

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

The “repr” String for an Object

The **`repr`** function returns a Python expression (as 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 (as 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 the value of an expression is what Python prints in an interactive session.

The “repr” String for an Object

The **`repr`** function returns a Python expression (as 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 the value of an expression is what Python prints in an interactive session.

```
>>> 12e12
```

The “repr” String for an Object

The **`repr`** function returns a Python expression (as 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 the value of an expression is what Python prints in an interactive session.

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

The “repr” String for an Object

The **`repr`** function returns a Python expression (as 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 the value of an expression is what Python prints in an interactive session.

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

The “repr” String for an Object

The **`repr`** function returns a Python expression (as 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 the value of an expression 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 (as 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 the value of an expression is what Python prints in an interactive session.

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

Some objects don't have a simple Python-readable string.

The “repr” String for an Object

The **`repr`** function returns a Python expression (as 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 the value of an expression is what Python prints in an interactive session.

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

Some objects don't have a simple Python-readable string.

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

The “str” String for an Object

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:

```
>>> import datetime
```

The “str” String for an Object

Human interpretable strings are useful as well:

```
>>> import datetime
>>> today = datetime.date(2013, 3, 11)
```

The “str” String for an Object

Human interpretable strings are useful as well:

```
>>> import datetime
>>> today = datetime.date(2013, 3, 11)
>>> repr(today)
```

The “str” String for an Object

Human interpretable strings are useful as well:

```
>>> import datetime
>>> today = datetime.date(2013, 3, 11)
>>> repr(today)
'datetime.date(2013, 3, 11)'
```

The “str” String for an Object

Human interpretable strings are useful as well:

```
>>> import datetime
>>> today = datetime.date(2013, 3, 11)
>>> repr(today)
'datetime.date(2013, 3, 11)'
>>> str(today)
```

The “str” String for an Object

Human interpretable strings are useful as well:

```
>>> import datetime
>>> today = datetime.date(2013, 3, 11)
>>> repr(today)
'datetime.date(2013, 3, 11)'
>>> str(today)
'2013-03-11'
```

The “str” String for an Object

Human interpretable strings are useful as well:

```
>>> import datetime
>>> today = datetime.date(2013, 3, 11)
>>> repr(today)
'datetime.date(2013, 3, 11)'
>>> str(today)
'2013-03-11'
```

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

Message Passing Enables Polymorphism

Message Passing Enables Polymorphism

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

Message Passing Enables Polymorphism

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

`str` and **`repr`** are both polymorphic; they apply to anything.

Message Passing Enables Polymorphism

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

`str` and **`repr`** are both polymorphic; they apply to anything.

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

Message Passing Enables Polymorphism

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

`str` and **`repr`** are both polymorphic; they apply to anything.

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

```
>>> today.__repr__()
'datetime.date(2012, 10, 8)'
```

Message Passing Enables Polymorphism

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

`str` and **`repr`** are both polymorphic; they apply to anything.

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

```
>>> today.__repr__()
'datetime.date(2012, 10, 8)'
```

`str` invokes a zero-argument method **`__str__`** on its argument. (But **`str`** is a class, not a function!)

Message Passing Enables Polymorphism

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

`str` and **`repr`** are both polymorphic; they apply to anything.

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

```
>>> today.__repr__()
'datetime.date(2012, 10, 8)'
```

`str` invokes a zero-argument method **`__str__`** on its argument. (But **`str`** is a class, not a function!)

```
>>> today.__str__()
'2012-10-08'
```

Inheritance and Polymorphism

Inheritance and Polymorphism

Inheritance also enables polymorphism, since subclasses provide at least as much behavior as their base classes

Inheritance and Polymorphism

Inheritance also enables polymorphism, since subclasses provide at least as much behavior as their base classes

Example of function that works on all accounts:

Inheritance and Polymorphism

Inheritance also enables polymorphism, since subclasses provide at least as much behavior as their base classes

Example of function that works on all accounts:

```
def welfare(account):
```

Inheritance and Polymorphism

Inheritance also enables polymorphism, since subclasses provide at least as much behavior as their base classes

Example of function that works on all accounts:

```
def welfare(account):
    """Deposit $100 into an account if it has less
      than $100."""
```

Inheritance and Polymorphism

Inheritance also enables polymorphism, since subclasses provide at least as much behavior as their base classes

Example of function that works on all accounts:

```
def welfare(account):
    """Deposit $100 into an account if it has less
      than $100."""
    if account.balance < 100:
```

Inheritance and Polymorphism

Inheritance also enables polymorphism, since subclasses provide at least as much behavior as their base classes

Example of function that works on all accounts:

```
def welfare(account):
    """Deposit $100 into an account if it has less
      than $100."""
    if account.balance < 100:
        return account.deposit(100)
```

Inheritance and Polymorphism

Inheritance also enables polymorphism, since subclasses provide at least as much behavior as their base classes

Example of function that works on all accounts:

```
def welfare(account):
    """Deposit $100 into an account if it has less
      than $100."""
    if account.balance < 100:
        return account.deposit(100)
```

```
>>> alice_account = CheckingAccount('Alice')
```

Inheritance and Polymorphism

Inheritance also enables polymorphism, since subclasses provide at least as much behavior as their base classes

Example of function that works on all accounts:

```
def welfare(account):
    """Deposit $100 into an account if it has less
      than $100."""
    if account.balance < 100:
        return account.deposit(100)
```

```
>>> alice_account = CheckingAccount('Alice')
>>> welfare(alice_account)
```

Inheritance and Polymorphism

Inheritance also enables polymorphism, since subclasses provide at least as much behavior as their base classes

Example of function that works on all accounts:

```
def welfare(account):
    """Deposit $100 into an account if it has less
      than $100."""
    if account.balance < 100:
        return account.deposit(100)
```

```
>>> alice_account = CheckingAccount('Alice')
>>> welfare(alice_account)
100
```

Inheritance and Polymorphism

Inheritance also enables polymorphism, since subclasses provide at least as much behavior as their base classes

Example of function that works on all accounts:

```
def welfare(account):
    """Deposit $100 into an account if it has less
      than $100."""
    if account.balance < 100:
        return account.deposit(100)
```

```
>>> alice_account = CheckingAccount('Alice')
>>> welfare(alice_account)
100
>>> bob_account = SavingsAccount('Bob')
```

Inheritance and Polymorphism

Inheritance also enables polymorphism, since subclasses provide at least as much behavior as their base classes

Example of function that works on all accounts:

```
def welfare(account):
    """Deposit $100 into an account if it has less
      than $100."""
    if account.balance < 100:
        return account.deposit(100)
```

```
>>> alice_account = CheckingAccount('Alice')
>>> welfare(alice_account)
100
>>> bob_account = SavingsAccount('Bob')
>>> welfare(bob_account)
```

Inheritance and Polymorphism

Inheritance also enables polymorphism, since subclasses provide at least as much behavior as their base classes

Example of function that works on all accounts:

```
def welfare(account):
    """Deposit $100 into an account if it has less
      than $100."""
    if account.balance < 100:
        return account.deposit(100)
```

```
>>> alice_account = CheckingAccount('Alice')
>>> welfare(alice_account)
100
>>> bob_account = SavingsAccount('Bob')
>>> welfare(bob_account)
98
```

Interfaces

Interfaces

Message passing allows **different data types** to respond to the **same message**.

Interfaces

Message passing allows **different data types** to respond to the **same message**.

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

Interfaces

Message passing allows **different data types** to respond to the **same message**.

A shared message 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 allows **different data types** to respond to the **same message**.

A shared message 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**.

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

Interfaces

Message passing allows **different data types** to respond to the **same message**.

A shared message 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**.

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

Classes that implement **`__len__`** and **`__getitem__`** are sequences.

Special Methods

Special Methods

Python operators and generic functions make use of methods with names like “**__name__**”

Special Methods

Python operators and generic functions make use of methods with names like “**__name__**”

These are *special* or *magic methods*

Special Methods

Python operators and generic functions make use of methods with names like "**`__name__`**"

These are *special* or *magic methods*

Examples:

Special Methods

Python operators and generic functions make use of methods with names like “**__name__**”

These are *special* or *magic methods*

Examples:

len **__len__**

Special Methods

Python operators and generic functions make use of methods with names like “**`__name__`**”

These are *special* or *magic methods*

Examples:

`len`	**`__len__`**
`+, +=`	**`__add__, __iadd__`**

Special Methods

Python operators and generic functions make use of methods with names like “**`__name__`**”

These are *special* or *magic methods*

Examples:

`len`	**`__len__`**
`+, +=`	**`__add__, __iadd__`**
`[], []=`	**`__getitem__, __setitem__`**

Special Methods

Python operators and generic functions make use of methods with names like “**__name__**”

These are *special* or *magic methods*

Examples:

len	**__len__**
+, +=	**__add__, __iadd__**
[], []=	**__getitem__, __setitem__**
.	**__getattr__, __getattribute__, __setattr__**

Special Methods

Python operators and generic functions make use of methods with names like “**`__name__`**”

These are *special* or *magic methods*

Examples:

`len`	`__len__`
`+, +=`	`__add__, __iadd__`
`[], []=`	`__getitem__, __setitem__`

`.`	`__getattr__, __getattribute__,`
	`__setattr__`

`a[i]` is equivalent to **`type(a).__getitem__(a, i)`**

Example: Rational Numbers

Example: Rational Numbers

```
class Rational(object):
```

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
```

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
```

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
```

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
```

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
    def __repr__(self):
```

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numerator,
```

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numerator,
                                           self.denominator)
```

Example: Rational Numbers

```python
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numerator,
                                           self.denominator)
    def __str__(self):
```

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numerator,
                                           self.denominator)
    def __str__(self):
        return '{0}/{1}'.format(self.numerator,
                                  self.denominator)
```

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numerator,
                                           self.denominator)
    def __str__(self):
        return '{0}/{1}'.format(self.numerator,
                                  self.denominator)
    def __add__(self, num):
```

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numerator,
                                           self.denominator)
    def __str__(self):
        return '{0}/{1}'.format(self.numerator,
                                self.denominator)
    def __add__(self, num):
        denom = self.denominator * num.denominator
```

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numerator,
                                           self.denominator)
    def __str__(self):
        return '{0}/{1}'.format(self.numerator,
                                  self.denominator)
    def __add__(self, num):
        denom = self.denominator * num.denominator
        numer1 = self.numerator * num.denominator
```

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numerator,
                                           self.denominator)
    def __str__(self):
        return '{0}/{1}'.format(self.numerator,
                                  self.denominator)
    def __add__(self, num):
        denom = self.denominator * num.denominator
        numer1 = self.numerator * num.denominator
        numer2 = self.denominator * num.numerator
```

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numerator,
                                           self.denominator)
    def __str__(self):
        return '{0}/{1}'.format(self.numerator,
                                  self.denominator)
    def __add__(self, num):
        denom = self.denominator * num.denominator
        numer1 = self.numerator * num.denominator
        numer2 = self.denominator * num.numerator
        return Rational(numer1 + numer2, denom)
```

Example: Rational Numbers

```python
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numerator,
                                           self.denominator)
    def __str__(self):
        return '{0}/{1}'.format(self.numerator,
                                self.denominator)
    def __add__(self, num):
        denom = self.denominator * num.denominator
        numer1 = self.numerator * num.denominator
        numer2 = self.denominator * num.numerator
        return Rational(numer1 + numer2, denom)
    def __eq__(self, num):
```

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numerator,
                                           self.denominator)
    def __str__(self):
        return '{0}/{1}'.format(self.numerator,
                                  self.denominator)
    def __add__(self, num):
        denom = self.denominator * num.denominator
        numer1 = self.numerator * num.denominator
        numer2 = self.denominator * num.numerator
        return Rational(numer1 + numer2, denom)
    def __eq__(self, num):
        return (self.numerator == num.numerator and
                  self.denominator == num.denominator)
```

Property Methods

Property Methods

Often, we want the value of instance attributes to be linked.

Property Methods

Often, we want the value of instance attributes to be linked.

```
>>> f = Rational(3, 5)
```

Property Methods

Often, we want the value of instance attributes to be linked.

```
>>> f = Rational(3, 5)
>>> f.float_value
```

Property Methods

Often, we want the value of instance attributes to be linked.

```
>>> f = Rational(3, 5)
>>> f.float_value
0.6
```

Property Methods

Often, we want the value of instance attributes to be linked.

```
>>> f = Rational(3, 5)
>>> f.float_value
0.6
>>> f.numerator = 4
```

Property Methods

Often, we want the value of instance attributes to be linked.

```
>>> f = Rational(3, 5)
>>> f.float_value
0.6
>>> f.numerator = 4
>>> f.float_value
```

Property Methods

Often, we want the value of instance attributes to be linked.

```
>>> f = Rational(3, 5)
>>> f.float_value
0.6
>>> f.numerator = 4
>>> f.float_value
0.8
```

Property Methods

Often, we want the value of instance attributes to be linked.

```
>>> f = Rational(3, 5)
>>> f.float_value
0.6
>>> f.numerator = 4
>>> f.float_value
0.8
>>> f.denominator -= 3
```

Property Methods

Often, we want the value of instance attributes to be linked.

```
>>> f = Rational(3, 5)
>>> f.float_value
0.6
>>> f.numerator = 4
>>> f.float_value
0.8
>>> f.denominator -= 3
>>> f.float_value
```

Property Methods

Often, we want the value of instance attributes to be linked.

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

Property Methods

Often, we want the value of instance attributes to be linked.

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

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 be linked.

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

```
@property
def float_value(self):
    return (self.numerator //
            self.denominator)
```

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 be linked.

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

```
@property
def float_value(self):
    return (self.numerator //
            self.denominator)
```

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

It allows zero-argument methods to be called without an explicit call expression.