Looking Up Names

Name expressions look up names in the environment

\(<\text{name}>\) . \(<\text{name}>\)

Dot expressions look up names in an object

\(<\text{expression}>\).\(<\text{name}>\)

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

Methods looked up from bottom to top, left to right

The \texttt{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

The \texttt{object} class is at the root of the inheritance hierarchy

* \texttt{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

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**

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

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

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

```python
>>> repr(min)
"<built-in function min>"
```

---

**The “str” String for an Object**

Human interpretable strings are useful as well:

```python
>>> 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**

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

(But `str` is a class, not a function!)

```python
>>> today.__repr__()
'datetime.date(2012, 10, 8)'
>>> today.__str__()
'2012-10-08'
```

---

**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:

```python
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
```

```python
>>> bob_account = SavingsAccount('Bob')
>>> welfare(bob_account)
98
```

---

**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.
Python operators and generic functions make use of methods with names like "\_\_name\_\_"
These are special or magic methods
Examples:

```python
len
+-+=\n\[\],[\]=
.\.
\_\_getitem\_,\_\_setitem\_
\_\_getattribute\_,\_\_getattribute_,\_\_setattr_
```

`a[i]` is equivalent to `type(a).\_\_getitem\_(a, i)`

Each method is a zero-argument function that can be called without an explicit call expression.

### 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):
        return (self.numerator == num.numerator and self.denominator == num.denominator)
```

```python
# Example usage:
>>> 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.

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

### Property Methods

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

```python
>>> 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.

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