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 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.

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

Some objects do not have a simple Python-readable string.

```
>>> 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'
```

---

Polymorphic Functions

Polymorphic functions: 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
- Question: How would we implement this behavior?

The behavior of `str` is also complicated:
- An instance attribute called `__str__` is ignored
- `str` returns the `__repr__` of the object
- Question: How would we implement this behavior?

```
# Demo

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

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

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

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

```
# Demo

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

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

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

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

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

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

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

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

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

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__
__repr__
__add__
__bool__
__float__
```

Method invoked automatically when an object is constructed
Method invoked to display an object as a Python expression
Method invoked to add one object to another
Method invoked to convert an object to True or False
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)
```

Generic Functions

A polymorphic function might take 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

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

```
>>> 1 + Ratio(1, 3)
Ratio(4, 3)
```

```
>>> from math import pi
>>> Ratio(1, 3) + pi
3.4749259869231266
```

```
>>> Ratio(1, 3) + 1
Ratio(4, 3)
```

```
>>> from math import pi
>>> Ratio(1, 3) + pi
3.4749259869231266
```

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) + 1
Ratio(4, 3)
```

```
>>> from math import pi
>>> Ratio(1, 3) + pi
3.4749259869231266
```
Section: Modular Design

**Announcements**

1. Modular Design
2. Separation of Concerns

A design principle: Isolate different parts of a program that address different concerns. A modular component can be developed and tested independently.

<table>
<thead>
<tr>
<th>Hog Game Simulator</th>
<th>Game Commentary</th>
<th>Player Strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Game rules</td>
<td>• Event descriptions</td>
<td>• Decision rules</td>
</tr>
<tr>
<td>• Ordering of events</td>
<td>• State tracking to generate commentary</td>
<td>• Strategy parameters (e.g., margin &amp; number of dice)</td>
</tr>
</tbody>
</table>

**Hog Game Simulator**

**Game Commentary**

- Event descriptions
- State tracking to generate commentary

**Player Strategies**

- Decision rules
- Strategy parameters (e.g., margin & number of dice)

**Example: Hog Game Simulator**

**Game Commentary**

Example: Restaurant Search

Given the following data, look up a restaurant by name and show related restaurants.

```json
{
  "business_id": "gclB3ED6uk6viWlolSb_uA",
  "name": "Cafe 3",
  "stars": 2.0,
  "price": 1,
  ...
}
{
  "business_id": "WXKx2I2SEzBpeUGtDMCS8A",
  "name": "La Cascada Taqueria",
  "stars": 3.0,
  "price": 2
}
...
{
  "business_id": "gclB3ED6uk6viWlolSb_uA",
  "user_id": "xVocUszkZtAqCxgWak3xVQ",
  "stars": 1,
  "text": "Cafe 3 (or Cafe Tre, as I like to say) used to be the bomb diggity when I first lived in the dorms but sadly, quality has dramatically decreased over the years....",
  "date": "2012-01-19",
  ...
}
{
  "business_id": "WXKx2I2SEzBpeUGtDMCS8A",
  "user_id": "84dCHkhWG8IDtk30VvaY5A",
  "stars": 2,
  "text": "- Excuse me for being a snob but if I wanted a room temperature burrito I would take one home, stick it in the fridge for a day, throw it in the microwave for 45 seconds, then eat it. NOT go to a restaurant and pay like seven dollars for one....",
  "date": "2009-04-30",
  ...
}
...
```

**Restaurant Search Data**

Example: Similar Restaurants

Given the following data, look up a restaurant by name and show related restaurants.

```json
...
```
Discussion Question: Most Similar Restaurants

Implement `similar`, a `Restaurant` method that takes a positive integer `k` and a function `similarity` that takes two restaurants as arguments and returns a number; higher similarity values indicate more similar restaurants. The `similar` method returns a list containing the `k` most similar restaurants according to the `similarity` function, but not containing `self`.

```python
def similar(self, k, similarity):
    """Return the K most similar restaurants to SELF, using SIMILARITY for comparison.""
    others = list(Restaurant.all)
    others.remove(self)
    return sorted(others, key=lambda r: -similarity(self, r))[:k]
```

A custom key function can be supplied to customize the sort order, and the reverse flag can be set to request the result in descending order.

Example: Reading Files

Set Intersection

```python
def fast_overlap(s, t):
    """Return the overlap between sorted S and sorted T.
    >>> fast_overlap([3, 4, 6, 7, 9, 10], [1, 3, 5, 7, 8])
    2
    """  
i, j, count = 0, 0, 0
    while i < len(s) and j < len(t):
        if s[i] == t[j]:
            count, i, j = count + 1, i + 1, j + 1
        elif s[i] < t[j]:
            i = i + 1
        else:
            j = j + 1
    return count
```

Linear-Time Intersection of Sorted Lists

Give two sorted lists with no repeats, return the number of elements that appear in both.

Sets

One more built-in Python container type
- Set literals are enclosed in braces
- Duplicate elements are removed as constructions
- Sets have arbitrary order
- Sets have `len()`, `iter()`, `x in S`, `x not in S`, `x + y`, `len()`, `x.union(y)`, `x.intersection(y)`,