Factories

The idea of factories

Imagine this version of a money class:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
```

This constructor expects both dollar and cent amounts.

Constructor mismatch

What if our application is working with cents directly? We have to manually decompose it:

```
>>> # Emptying the penny jar...
... total_pennies = 3274
>>> # // is integer division
... dollars = total_pennies // 100
>>> cents = total_pennies % 100
>>> total_cash = Money(dollars, cents)
```

Suppose this is very common in our code. Can we encapsulate it a bit better?

Change the constructor?

One thing we can do is change the constructor:

```
class Money:
    def __init__(self, total_cents):
        self.dollars = total_cents // 100
        self.cents = total_cents % 100
```

That means we lose the first constructor, though.

(Some languages let you define several constructors. But even if Python let us do that, it would not solve all problems.)

Factory function

A better solution: keep the more general constructor, and create a "factory" function.

```
# Let's back up, to the original Money constructor.
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents

# From cents:
def money_from_pennies(total_cents):
    dollars = total_cents // 100
    cents = total_cents % 100
    return Money(dollars, cents)
```

As many as we want!

In fact, we can create as many of these factory functions as we want. For example, create `Money` from a string like "$140.75":

```
import re
def money_from_string(amount):
    # amount is a string like "$140.75"
    match = re.search(r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    if match is None:
        raise ValueError('Invalid amount: {}'.format(amount))
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

This works. But...

Subclassing

... it only works for the `Money` class. Subclasses need a whole different set of functions.

(And if we change the class name to, say, `Dollars`, we have a bit more refactoring to do too.)

Python provides a better solution.

@classmethod

`classmethod` is a built-in decorator that is applied to class methods. The method becomes associated with the class itself.

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    @classmethod
    def from_pennies(cls, total_cents):
        dollars = total_cents // 100
        cents = total_cents % 100
        return cls(dollars, cents)
```

Notice the first argument of `from_pennies`.

Class methods

You call it off the *class itself*, not an instance of the class.

```
>>> # It's like an extra constructor.
... piggie_bank_cash = Money.from_pennies(3217)
>>> type(piggie_bank_cash)
<class '__main__.Money'>
>>> piggie_bank_cash.dollars
32
>>> piggie_bank_cash.cents
17
>>> # And we can define as many as we want.
... piggie_bank_cash = Money.from_string("$14.72")
```

Subclassing

This automatically works with subclasses:

```
>>> class TipMoney(Money):
...     pass
...
>>> tip = TipMoney.from_pennies(475)
>>> type(tip)
<class '__main__.TipMoney'>
```

More maintainable. `@classmethod` is worth keeping in your toolbox.

Advantages

The OOP literature calls this the "simple factory" pattern. I prefer to call it "alternate constructor".

Its advantages:

- Can use descriptive method names
- Automatically extends to subclasses
- Encapsulated in the pertinent class

Static Methods

You can use static methods in Python too.

They tend to be less useful in Python than in other languages, because of `@classmethod` and other reasons.

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    @classmethod
    def from_pennies(cls, total_cents):
        dollars, cents = cls.pennies2dollarsandcents(total_cents)
        return cls(dollars, cents)
    # Utility function. Could also be a standalone function
    # in the same module as this class.
    @staticmethod
    def pennies2dollarsandcents(pennies):
        dollars = pennies // 100
        cents = pennies % 100
        return (dollars, cents)
```

Other factories

- "factory method" pattern (dynamic type pattern)
- "abstract factory" pattern (more complex, can be useful for DI)