Method dispatching techniques

in Python

The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.

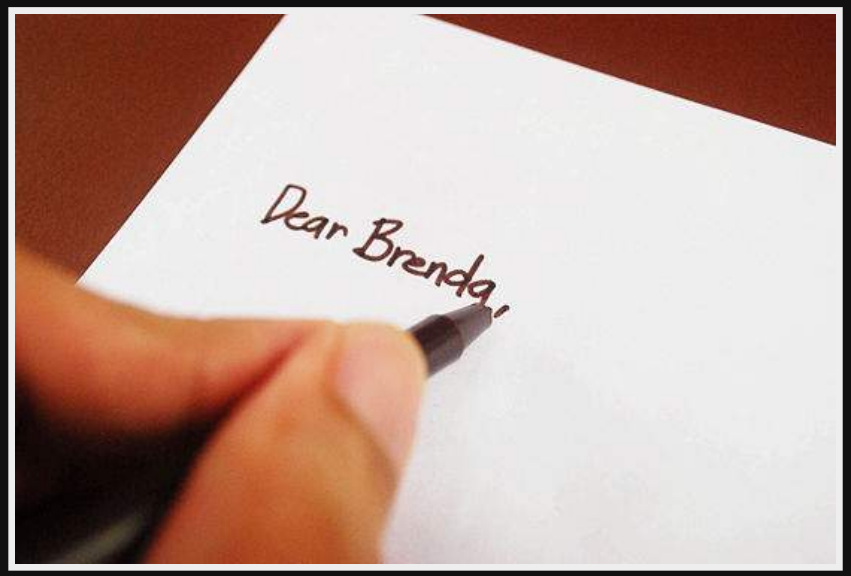
— Alan Kay on Messaging

Message passing

It's a way to **request behavior** from an entity in a distributed system.

To communicate via message is a **4-step** process

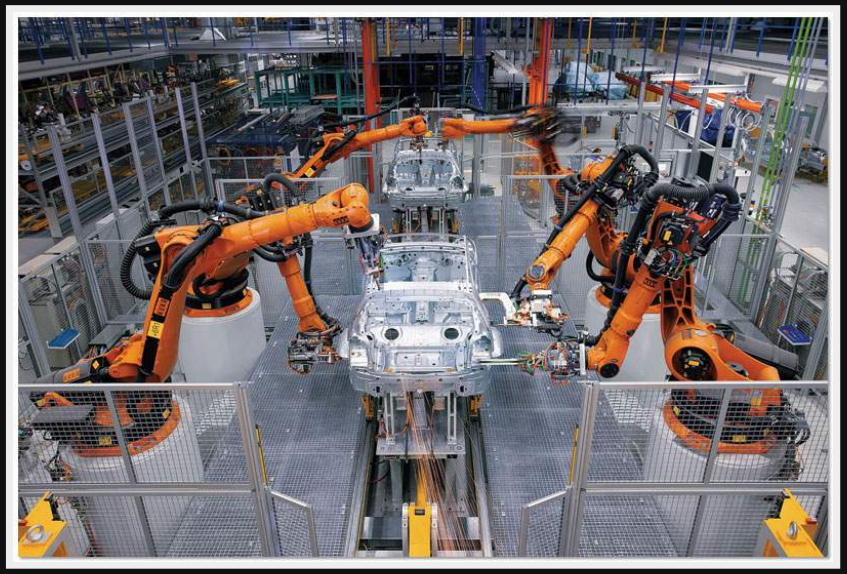
Creating the message:



Sending & Delivering the message:

Receiving the message:

Processing the message:



This last step is **method dispatching**!!

Method Dispatching
in Python

It's the process to select **which action** depending on the the message.

```
def __postoffice__(message):
  recipient = message.recipient
  name = message.contents.name
  args = message.contents.args
  kwargs = message.contents.kwargs

  if not hasattr(recipient, name):
    raise AttributeError

  member = getattr(recipient, name)
  return member(*args, **kwargs)
```

in (real) Python

Some differences:

1. The *post offices* are responsibilities of **each class**.
2. There are **two separated methods** to redefine method dispatching: one for *getting* and another for *setting*
3. The message is only the **member's name**

__getattribute__

It's the equivalent to `__postoffice__`

```
def __getattribute__(self, membername):
  """Get the value with the key <membername> inside
  self.__dict__; if not, look for the same key
  inside type(self).__dict__; if not, look for
  it through the class hierarchy. If not, call
  self.__getattr__(membername) if exists."""
```

Intercepts **all** the attemps to access a member before trying to retrieve it from the instance.

__getattr__

It's a **fallback** for when the member does not exist.

```
def __getattr__(self, membername):
  """If exists inside the instance, it's called when
  the membername was not found at all. It can return
  an object or raise AttributeError."""
```

Intercepts the attemps to access a member **only if it does not exist**.

__setattr__

It's the *post office* when trying to assign a value.

```
def __setattr__(self, membername, value):
  """Replace the entry in self.__dict__ for the
  membername with value.

  (Well, actually, it's a little bit more
  complicated)."""
```

Intercepts **all** the attemps to set a member with the specified value.

IMPORTANT:

if not called explicitly, the magic methods are **never called from the instance** but from its class.

```
from types import MethodType as method
class A(): pass

a = A()
a.__setattr__ = \
  method(lambda : print('Doing nothing...'), a)
a.test = 1
```

Fuzzy APIs

If I make a little typo, I want the API to be smart enough to figure out which member I'm referring and get it fixing the typo.

The idea is to use the most similar method without ambiguity when it's not found in the instance.

We use `difflib` and `SequenceMatcher` to calculate similarity:

```
from difflib import SequenceMatcher
def similarity(n, m):
  return SequenceMatcher(None, n, m).ratio()
```

And add `__getattr__()` to the class.

```
def __getattr__(self, name):
  issimilar = lambda n: similarity(n, name) >= 0.8
  matches = list(filter(issimilar, dir(self)))
  if not matches or len(matches) > 1:
    raise AttributeError()

  return getattr(self, matches[0])
```

Remeber it's only called if the member **is not found**.

Download fuzzyfy.py for an implementation.

Restricted Proxy Subclasses

From a base class, I want to derive another with partial access to its public members.

Provided a base class and a whitelist of members, only allow access to a member if it is in the whitelist.

So, as it's a metter of access, I need to intercept all access.

```
def __getattribute__(self, name):
  if name[0] != '_' and name not in whitelist:
    raise RuntimeError()

  return base.__getattribute__(self, name)
```

```
def __setattr__(self, name, value):
  if name[0] != '_' and name not in whitelist:
    raise RuntimeError()

  return base.__settattr__(self, name, value)
```

Download restricted.py for an implementation.

Private members

Deny the access to a member starting by _ if it's being used outside the class' (or subclasses') declarations.

When accessing a member, inspect the current stack to locate the line of code accessing the member and see if the line is part of the code for the class. If not, deny the access.

Use the module `inspect` to get the sourcefile and source lines from a class.

```
class A():
  """

  Source code

  """
  def f():
    print('Hello!')


import inspect
lines, offset = inspect.getsourcelines(A)
```

Copy to a file or it won't work.

Use in combination with `__mro__` to get the source files and source blocks for all classes in the class hierarchy.

```
klass = type(obj)
# ignore built-in object class
klasses = list(klass.__mro__[:-1])
sources = []
for k in klasses:
  sourcefile = inspect.getsourcefile(k)
  lines, offset = inspect.getsourcelines(k)
  end = offset + len(lines)
  sources.append(sourcefile, offset, end)
```

Use `getouterframes()` and `currentframe()` to get a list of the current execution stack.

```
def a():
  b()

def b():
  c()

def c():
  import inspect
  frames = inspect.getouterframes(
           inspect.currentframe())
  print(list(frames))
```

Each entry is a tuple including the sourcefile and line number.

Download protected.py for an implementation.

Jump to the infography about privates in Python.

Further reading

- Alan Kays on Messaging
- Smalltalk's dispatching
- Ptyhon datamodel
- Smart Command Line Opttions Parsing
- Git implementation of Levenshtein distance

About me

me
Salvador de la Puente González
twitter
@salvadelapuente
My web sites
http://unoyunodiez.com
http://github.com/lodr