1.1 Types

signature :: [type]
   a list of types

Dispatcher :: {signature: function}
   A mapping of type signatures to function implementations

namespace :: {str: Dispatcher}
   A mapping from function names, like ’add’, to Dispatchers

1.2 Dispatchers

A Dispatcher object stores and selects between different implementations of the same abstract operation. It selects
the appropriate implementation based on a signature, or list of types. We build one dispatcher per abstract operation.

\[ f = \text{Dispatcher}(’f’) \]

At the lowest level we build normal Python functions and then add them to the Dispatcher.

\[ \text{def inc}(x): \]
\[ \quad \text{return} \ x + 1 \]

\[ \text{def dec}(x): \]
\[ \quad \text{return} \ x - 1 \]

\[ f.\text{add}((\text{int,}), \text{inc}) \]  # f increments integers
\[ f.\text{add}((\text{float,}), \text{dec}) \]  # f decrements floats

\[ f(1) \]
2

\[ f(1.0) \]
0.0

Internally Dispatcher.resolve selects the function implementation.

\[ f.\text{resolve}((\text{int,})) \]
<function __main__.inc>
For notational convenience dispatchers leverage Python’s decorator syntax to register functions as we define them.

```python
>>> f.resolve((float,))
<function __main__.dec>
```

This is equivalent to the form above. It also adheres to the standard implemented by `functools.singledispatch` in Python 3.4.

### 1.3 Namespaces and dispatch

The `dispatch` decorator hides the creation and manipulation of `Dispatcher` objects from the user.

```python
# f = Dispatcher('f')  # no need to create Dispatcher ahead of time

@dispatch(int)
def inc(x):
    return x + 1

@dispatch(float)
def dec(x):
    return x - 1
```

The `dispatch` decorator uses the name of the function to select the appropriate `Dispatcher` object to which it adds the new signature/function. When it encounters a new function name it creates a new `Dispatcher` object and stores name/Dispatcher pair in a namespace for future reference.

```python
# This creates and stores a new Dispatcher('g')
# namespace['g'] = Dispatcher('g')
# namespace['g'].add((int,), g)
@dispatch(int)
def g(x):
    return x ** 2
```

We store this new `Dispatcher` in a `namespace`. A namespace is simply a dictionary that maps function names like ‘g’ to dispatcher objects like `Dispatcher('g')`.

By default `dispatch` uses the global namespace in `multipledispatch.core.global_namespace`. If several projects use this global namespace unwisely then conflicts may arise, causing difficult to track down bugs. Users who desire additional security may establish their own namespaces simply by creating a dictionary.

```
my_namespace = dict()

@dispatch(int, namespace=my_namespace)
def f(x):
    return x + 1
```

To establish a namespace for an entire project we suggest the use of `functools.partial` to bind the new namespace to the `dispatch` decorator.
from multipledispatch import dispatch
from functools import partial

my_namespace = dict()
dispatch = partial(dispatch, namespace=my_namespace)

@dispatch(int)  # Uses my_namespace rather than the global namespace
def f(x):
    return x + 1
Method Resolution

Multiple dispatch selects the function from the types of the inputs.

```python
@dispatch(int)
def f(x):
    # increment integers
    return x + 1

@dispatch(float)
def f(x):
    # decrement floats
    return x - 1
```

```python
>>> f(1)  # 1 is an int, so increment
2
>>> f(1.0)  # 1.0 is a float, so decrement
0.0
```

## 2.1 Union Types

Similarly to the built-in `isinstance` operation you specify multiple valid types with a tuple.

```python
@dispatch((list, tuple))
def f(x):
    """ Apply 'f' to each element in a list or tuple ""
    return [f(y) for y in x]
```

```python
>>> f([1, 2, 3])
[2, 3, 4]
>>> f((1, 2, 3))
[2, 3, 4]
```

## 2.2 Abstract Types

You can also use abstract classes like `Iterable` and `Number` in place of union types like `(list, tuple)` or `(int, float).

```python
from collections import Iterable

# @dispatch((list, tuple))
```
2.3 Selecting Specific Implementations

If multiple valid implementations exist then we use the most specific one. In the following example we build a function to flatten nested iterables.

```python
@dispatch(Iterable)
def flatten(L):
    return sum([flatten(x) for x in L], [])

@dispatch(object)
def flatten(x):
    return [x]

>>> flatten([1, 2, 3])
[1, 2, 3]

>>> flatten([1, [2], 3])
[1, 2, 3]

>>> flatten([1, 2, {3, 4}, [[5]], {(6, 7), (8, 9)}])
[1, 2, 3, 4, 5, 6, 7, 8, 9]

Because strings are iterable they too will be flattened

>>> flatten([1, 'hello', 3])
[1, 'h', 'e', 'l', 'l', 'o', 3]

We avoid this by specializing flatten to str. Because str is more specific than Iterable this function takes precedence for strings.

@dispatch(str)
def flatten(s):
    return s

>>> flatten([1, 'hello', 3])
[1, 'hello', 3]

The multiple dispatch project depends on Python’s issubclass mechanism to determine which types are more specific than others.

2.4 Multiple Inputs

All of these rules apply when we introduce multiple inputs.

```
2.5 Ambiguities

However ambiguities arise when different implementations of a function are equally valid.

```python
@dispatch(float, object)
def f(x, y):
    """ Square left hand side if it is a float """
    return x**2 + y

>>> f(2.0, 10.0)
?  
Which result do we expect, 2.0**2 + 10.0 or 2.0 + 10.0**2? The types of the inputs satisfy three different implementations, two of which have equal validity:

- input types: float, float
- Option 1: object, object
- Option 2: object, float
- Option 3: float, object

Option 1 is strictly less specific than either options 2 or 3 so we discard it. Options 2 and 3 however are equally specific and so it is unclear which to use.

To resolve issues like this `multipledispatch` inspects the type signatures given to it and searches for ambiguities. It then raises a warning like the following:

```
multipledispatch/dispatcher.py:74: AmbiguityWarning: Ambiguities exist in dispatched function f
The following signatures may result in ambiguous behavior:
    [object, float], [float, object]
```

Consider making the following additions:

```python
@dispatch(float, float)
def f(...)
```

This warning occurs when you write the function and guides you to create an implementation to break the ambiguity. In this case, a function with signature `(float, float)` is more specific than either options 2 or 3 and so resolves the issue. To avoid this warning you should implement this new function `before` the others.

```python
@dispatch(float, float)
def f(x, y):
    ...

@dispatch(float, object)
```
def f(x, y):
    ...

@dispatch(object, float)
def f(x, y):
    ...

If you do not resolve ambiguities by creating more specific functions then one of the competing functions will be selected pseudo-randomly.