# pyfpm Documentation

***Release 0.1.2***

**Martin Blech**

August 04, 2012

# CONTENTS

The name `pyfpm` stands for PYthon Functional Pattern Matching. It's an attempt to bring Scala-like functional pattern matching to Python with a similar syntax.

CHAPTER

ONE

# PYFPM.MATCHER

Matchers are the main user-facing API for *pyfpm*.

This module lets you unpack objects:

```
>>> unpacker = Unpacker()
>>> unpacker('head :: tail') << (1, 2, 3)
>>> unpacker.head
1
>>> unpacker.tail
(2, 3)
```

or function parameters:

```
>>> @match_args('[x:str, [y:int, z:int]]')
... def match(x, y, z):
...     return (x, y, z)

>>> match('abc', (1, 2))
('abc', 1, 2)
```

You can also create simple matchers with lambda expressions:

```
>>> what_is_it = Matcher([
...     ('_:int', lambda: 'an int'),
...     ('_:str', lambda: 'a string'),
...     ('x', lambda x: 'something else: %s' % x),
...     ])

>>> what_is_it(10)
'an int'
>>> what_is_it('abc')
'a string'
>>> what_is_it({})
'something else: {}'
```

or more complex ones using a decorator:

```
>>> parse_options = Matcher()
>>> @parse_options.handler("['-h'|'--help', None]")
... def help():
...     return 'help'
>>> @parse_options.handler("['-o'|'--optim', level:int] if 1<=level<=5")
... def set_optimization(level):
...     return 'optimization level set to %d' % level
>>> @parse_options.handler("['-o'|'--optim', bad_level]")
```

```
... def bad_optimization(bad_level):
...     return 'bad optimization level: %s' % bad_level
>>> @parse_options.handler('x')
... def unknown_options(x):
...     return 'unknown options: %s' % repr(x)

>>> parse_options(('-h', None))
'help'
>>> parse_options(('--help', None))
'help'
>>> parse_options(('-o', 3))
'optimization level set to 3'
>>> parse_options(('-o', 0))
'bad optimization level: 0'
>>> parse_options(('-v', 'x'))
"unknown options: ('-v', 'x')"
```

**class** pyfpm.matcher.**Matcher**(*bindings*=[ ], *context*=*None*)

Maps patterns to handler functions.

**Parameters**

- **bindings** (*iterable*) – an optional list of pattern-handler pairs. String patterns are automatically parsed.
- **context** (*dict*) – an optional context for the `Parser`. If absent, it uses the caller's *globals()*

**handler**(*pattern*)

Decorator for registering handlers. It's an alternate syntax with the same effect as `register()`:

```
>>> m = Matcher()
>>> @m.handler('x:int')
... def int_(x):
...     return 'an int: %d' % x
>>> @m.handler('_')
... def any():
...     return 'any'
>>> m(1)
'an int: 1'
>>> m(None)
'any'
```

**match**(*obj*, **args*)

Match the given object against the registerd patterns until the first match. The corresponding handler gets called with *args* as positional arguments and the match context as keyword arguments.

**Parameters**

- **obj** – the object to match the patterns with
- **args** – the extra positional arguments that the handler function will get called with

**Raises** NoMatch – if none of the patterns can match de object

Example:

```
>>> m = Matcher([
... ('head :: tail', lambda extra, head, tail: (extra, head, tail)),
... ('x', lambda extra, x: (extra, 'got something! %s' % x)),
... ])
>>> m.match('hello', 'yo!')
('yo!', 'got something! hello')
```

```
>>> m.match((1, 2, 3), 'numbers')
('numbers', 1, (2, 3))
```

**register**(*pattern*, *handler*)
Register a new pattern-handler pair. If the pattern is a string, it will be parsed automatically.

**Parameters**

- **pattern** – Pattern or str – the pattern
- **handler** – callable – the handler function for the pattern

pyfpm.matcher.**match_args**(*pattern*, *context=None*)
Decorator for matching a function's arglist.

**Parameters**

- **pattern** – Pattern or str – the pattern
- **context** – dict – an optional context for the pattern parser. If absent, it defaults to the caller's *globals()*.

Usage:

```
>>> @match_args('head::tail')
... def do_something(head, tail):
...     return (head, tail)
>>> do_something(1, 2, 3, 4)
(1, (2, 3, 4))
```

**class** pyfpm.matcher.**Unpacker**
Inline object unpacker. Usage:

```
>>> unpacker = Unpacker()
>>> unpacker('[x, [y, z]]') << (1, (2, 3))
>>> unpacker.x
1
>>> unpacker.y
2
>>> unpacker.z
3
```

**exception** pyfpm.matcher.**NoMatch**
Thrown by matchers when no registered pattern could match the given object.

CHAPTER

TWO

# PYFPM.PARSER

Scala-like pattern syntax parser.

pyfpm.parser.**Parser**(*context=None*)

Create a parser.

**Parameters** **context** (*dict*) – optional context, defaults to the caller's *globals()*

> **Warning:** creating a parser is expensive!

Usage and syntax examples:

```
>>> parser = Parser()
```

match anything anonymously:

```
>>> parser('_') << 'whatever'
Match({})
```

match anything and bind to a name:

```
>>> parser('x') << 1
Match({'x': 1})
```

match instances of a specific type:

```
>>> parser('_:str') << 1
>>> parser('_:int') << 1
Match({})
>>> parser('x:str') << 'abc'
Match({'x': 'abc'})
```

match int, float, str and bool constants:

```
>>> parser('1') << 1
Match({})
>>> parser('1.618') << 1.618
Match({})
>>> parser('"abc"') << 'abc'
Match({})
>>> parser('True') << True
Match({})
```

match lists:

```
>>> parser('[]') << ()
Match({})
>>> parser('[x:int]') << [1]
Match({'x': 1})
>>> parser('[a, b, _]') << [1, 2, 3]
Match({'a': 1, 'b': 2})
```

split head vs. tail:

```
>>> parser('a::b') << (1, 2, 3)
Match({'a': 1, 'b': (2, 3)})
>>> parser('a::b::c') << (0, 1, 2, 3, 4)
Match({'a': 0, 'c': (2, 3, 4), 'b': 1})
```

match named tuples (as if they were Scala case classes)

```
>>> try:
...     from collections import namedtuple
...     Case3 = namedtuple('Case3', 'a b c')
...     parser = Parser() # Case3 has to be in the context
...     parser('Case3(x, y, z)') << Case3(1, 2, 3)
... except ImportError:
...     from pyfpm.pattern import Match
...     Match({'y': 2, 'x': 1, 'z': 3}) # no namedtuple in python < 2.6
Match({'y': 2, 'x': 1, 'z': 3})
```

boolean or between expressions:

```
>>> parser('a:int|b:str') << 1
Match({'a': 1})
>>> parser('a:int|b:str') << 'hello'
Match({'b': 'hello'})
```

nest expressions:

```
>>> parser('[[[x:int]]]') << [[[1]]]
Match({'x': 1})
>>> parser('[_:int|[], 2, 3]') << (1, 2, 3)
Match({})
>>> parser('[_:int|[], 2, 3]') << ([], 2, 3)
Match({})
>>> parser('[_:int|[], 2, 3]') << ([1], 2, 3)
```

CHAPTER

THREE

# PYFPM.PATTERN

This module holds the actual pattern implementations.

End users should not normally have to deal with it, except for constructing patterns programatically without making use of the pattern syntax parser.

pyfpm.pattern.**build**(**args*, ***kwargs*)

Shorthand pattern factory.

Examples:

```
>>> build() == AnyPattern()
True
>>> build(1) == EqualsPattern(1)
True
>>> build('abc') == EqualsPattern('abc')
True
>>> build(str) == InstanceOfPattern(str)
True
>>> build(re.compile('.*')) == RegexPattern('.*')
True
>>> build(()) == build([]) == ListPattern()
True
>>> build([1]) == build((1,)) == ListPattern(EqualsPattern(1),
...     ListPattern())
True
>>> build(int, str, 'a') == ListPattern(InstanceOfPattern(int),
...     ListPattern(InstanceOfPattern(str),
...         ListPattern(EqualsPattern('a'))))
True
>>> try:
...     from collections import namedtuple
...     MyTuple = namedtuple('MyTuple', 'a b c')
...     build(MyTuple(1, 2, 3)) == NamedTuplePattern(MyTuple, 1, 2, 3)
... except ImportError:
...     True
True
```

**class** pyfpm.pattern.**Match**(*ctx=None*, *value=None*)

Represents the result of matching successfully a pattern against an object. The *ctx* attribute is a `dict` that contains the value for each bound name in the pattern, if any.

**class** pyfpm.pattern.**Pattern**

Base Pattern class. Abstracts the behavior common to all pattern types, such as name bindings, conditionals and operator overloading for combining several patterns.

**bind**(*name*)
Bind this pattern to the given name. Operator: %.

**head_tail_with**(*other*)
Head-tail concatenate this pattern with the other. The lhs pattern will be the head and the other will be the tail. Operator: +.

Example:

```
>>> p = InstanceOfPattern(int).head_tail_with(ListPattern())
>>> p.match([1])
Match({})
>>> p.match([1, 2])
```

**if_**(*condition*)
Add a boolean condition to this pattern. Operator: /.

**Parameters** **condition** (*callable*) – must accept the match context as keyword arguments and return a boolean-ish value.

**match**(*other*, *ctx=None*)
Match this pattern against an object. Operator: <<.

**Parameters**

- **other** – the object this pattern should be matched against.
- **ctx** (*dict*) – optional context. If none, an empty one will be automatically created.

**Returns** a `Match` if successful, *None* otherwise.

**multiply**(*n*)
Build a `ListPattern` that matches *n* instances of this pattern. Operator: *.

Example:

```
>>> p = EqualsPattern(1).multiply(3)
>>> p.match((1, 1, 1))
Match({})
```

**or_with**(*other*)
Build a new `OrPattern` with this or the other pattern. Operator: |.

Example:

```
>>> p = EqualsPattern(1).or_with(InstanceOfPattern(str))
>>> p.match('hello')
Match({})
>>> p.match(1)
Match({})
>>> p.match(2)
```

**class** pyfpm.pattern.**AnyPattern**
Pattern that matches anything.

**class** pyfpm.pattern.**EqualsPattern**(*obj*)
Pattern that only matches objects that equal the given object.

**class** pyfpm.pattern.**InstanceOfPattern**(*cls*)
Pattern that only matches instances of the given class.

**class** pyfpm.pattern.**RegexPattern**(*regex*)
Pattern that only matches strings that match the given regex.

**class** pyfpm.pattern.**ListPattern**(*head_pattern=None*, *tail_pattern=None*)
Pattern that only matches iterables whose head matches *head_pattern* and whose tail matches *tail_pattern*

**class** pyfpm.pattern.**NamedTuplePattern**(*casecls*, **initpatterns*)
Pattern that only matches named tuples of the given class and whose contents match the given patterns.

**class** pyfpm.pattern.**OrPattern**(**patterns*)
Pattern that matches whenever any of the inner patterns match.

CHAPTER

FOUR

# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

# PYTHON MODULE INDEX

## p