Functional Error Handling in Python

Moiz Merchant

Imperative Python

```
def parse_int_field(data, length):

    if len(data) == length:
        if re.match("\s+", data):
            return None
        if re.match("[0-9]+", data):
            try:
                return int(data)
            except:
                raise Exception("Failed to parse data: %s." % data)
        else:
            raise Exception("Invalid data found: %s" % data)
    else:
        raise Exception("Invalid length of data: %s" % data)
```

There must be something more than this

Functional Python

```
def parse_field(data, length, validator, parser):

    parse_empty = lambda d: check_spaces(d).map(lambda _: None)
    parse_value = lambda: validator(data).flatmap(parser)

    return (Right(data)
        .flatmap(lambda d: check_length(length, d))
        .flatmap(lambda d: parse_empty(d).or_elsef(parse_value)))
```

How did I get here?

Journey to Functional

c#	c/c++	python
obj-c	ruby	java
python	php	scala
clojure	python	

Journey to Functional

c# c/c++ python

obj-c ruby java

python php scala

clojure python

Journey to Functional

c#	c/c++	python
obj-c	ruby	java
python	php	scala
clojure	python	

Scala

```
request.body.asJson.map { jsonBody =>
  DB.withSession { implicit s =>
    val offset   = (page - 1) * count
    val queries = buildQueries(jsonBody)
    val entries = queries.sorted
      .drop(offset)
      .take(count)
      .run
      .map(userToUserListEntry)
    Ok(Json.obj(
      "result" -> Json.toJson(entries),
      "total" -> queries.filtered.length.run))
  }
} getOrElse {
  BadRequest
}
```

Clojure

```
(letfn [(turn-seq []
          (cycle [go-right go-down go-left go-up]))
        (spiral-coords [coord n]
          (reduce (fn [a f] (concat a (f (last a))))
            [coord]
            (map (fn [i f] (partial f i))
                 (reverse (range (inc n))) (turn-seq))))
        (matrix [n]
          (vec (take n (repeat (vec (take n (repeat " "))))))))]
  (defn spiral [n]
    (map println
      (reduce
        (fn [a coord] (assoc-in a coord "*"))
        (matrix n)
        (spiral-coords [0 0] n)))))
```

Imperative Chaos

- Inheritance 7 layers deep
- Base Class with Feature Flags
- Weak Encapsulation

Functional or Bust

Imperative Error

```
def parse_field(data, length, validator, parser):

    if check_length(length, data):
        if check_spaces(data):
            return None
        if validator(data):
            try:
                return parser(data)
            except:
                raise Exception("Failed to parse data: %s." % data)
        else:
            raise Exception("Invalid data found: %s" % data)
    else:
        raise Exception("Invalid length of data: %s" % data)
```

Existing Monadic Libraries

- pyMonad
- OSlash
- fn.py

pyfnz

- biased Either
- Try
- do Notation
- clj.core

Why Either?

```
class Either(object):
    """Modeled after scalaz's right-biased implementation."""

    def __init__(self, value):
        self._value = value

class Left(Either):
    """Represent a failure state."""
    __slots__ = ('_value',)

class Right(Either):
    """Represent a successful state."""
    __slots__ = ('_value',)
```

Pythonic Either

```
def map[D](g: B => D): (A \/ D) =
  this match {
    case \/-(b)     => \/-(g(b))
    case a @ -\/(_) => a.coerceRight
  }
```

```
def map(self, f):
    if type(self) is Left:
        return self
    elif type(self) is Right:
        return Right(f(self._value))
```

Pythonic Either

```scala
def flatMap[D](g: B => (A \/ D)): (A \/ D) =
  this match {
    case a @ -\/(_) => a.coerceRight
    case \/-(b) => g(b)
  }
```

```python
def flatmap(self, g):
    if type(self) is Left:
        return self
    elif type(self) is Right:
        return g(self._value)
```

Without Either

```
if (len(data) != 0):
    if (re.match(r"[a-z]+", data)):
        if (data == "".join(reversed(data))):
            q, r = divmod(len(data), 2)
            return data[:q+r]
        else:
            raise Exception("Data must be palindrome.")
    else:
        raise Exception("Data must be lower case.")
else:
    raise Exception("Data cannot be empty.")
```

Railway-oriented

Pythonic Either

```
return (Right(data)
    .flatmap(check_len)
    .flatmap(check_lower_alpha)
    .flatmap(check_palin)
    .map(get_half))
```

Pythonic Either

Pythonic Either

```
check_len = (lambda d:
    Right(d) if not is_empty(d) else Left("Data cannot be empty."))

check_lower_alpha = (lambda d:
    Right(d) if re.match(r"[a-z]+", d) else Left("Data must be lower case alpha."))

check_palin = (lambda d:
    Right(d) if d == "".join(reversed(d)) else Left("Data must be palindrome."))

def get_half(d):
    q, r = divmod(len(d), 2)
    return d[:q+r]

return (Right(data)
    .flatmap(check_len)
    .flatmap(check_lower_alpha)
    .flatmap(check_palin)
    .map(get_half))
```

Why Try?

Why Try?

```
class Try(object):
    """Modeled after scala's Try."""

    def __new__(cls, f, *args, **kwargs):
        try:
            value = f(*args, **kwargs)
            instance = object.__new__(Success)
            instance._value = value
        except Exception, e:
            instance = object.__new__(Failure)
            instance._value = e
        return instance

class Failure(Try):
    def __new__(cls, *args, **kargs):
        return object.__new__(Failure)

class Success(Try):
    def __new__(cls, *args, **kargs):
        return object.__new__(Success)
```

Try.Fail

Try Fail

```
Try(value.toInt)
```

```
Try(lambda: int(value))
```

Pythonic Try

Pythonic Try

```
Try(value.toInt)
```

```
Try(int, value)
```

Without Try

Without Try

```
try:
    dividend = int(readline('dividend'))
except Exception, e:
    dividend = 1

try:
    divisor = int(readline('divisor'))
except Exception, e:
    divisor = 1

try:
    return dividend / divisor
except Exception, e:
    return 0
```

Pythonic Try

Pythonic Try

```
one                 = constantly(1)
read_int            = comp(int, readline)
maybe_dividend = Try(read_int, 'dividend').recover(one)
maybe_divisor   = Try(read_int, 'divisor').recover(one)
return maybe_dividend.flatmap(
    lambda x: maybe_divisor.map(lambda y: x/y))
        .getOrElse(0)
```

Combining Constructs

Combining Constructs

```
maybe_name = parse_name(name)
maybe_type = parse_type(type)
maybe_range = parse_range(range)
```

Flatmapping

Flatmapping

```
maybe_name = parse_name(name)
maybe_type = parse_type(type)
maybe_range = parse_range(range)

maybe_name.flatmap(lambda n:
    maybe_type.flatmap(lambda t:
        maybe_range.flatmap(lambda r:
            make_parser(n, t, r))))
```

Reduce Ap Partials

Reduce Ap Partials

```
maybe_name = parse_name(name)
maybe_type = parse_type(type)
maybe_range = parse_range(range)

reduce(lambda f, m : m.ap_partial(f),
    [maybe_name,
     maybe_type,
     maybe_range],
    Right(make_parser))
  .map(lambda x: x())
```

List Comprehension

```
maybe_name = parse_name(name)
maybe_type = parse_type(type)
maybe_range = parse_range(range)

try:
    [make_parser(n, t, r)
     for n in maybe_name
     for t in maybe_type
     for r in maybe_range]
except EitherIterExcept, e:
    e.obj
```

Pythonic Do Notation

```
maybe_name = parse_name(name)
maybe_type = parse_type(type)
maybe_range = parse_range(range)

Either.do(make_parser(n, t, r)
          for n in maybe_name
          for t in maybe_type
          for r in maybe_range)
```

Pyjure

```
if (coll is None) or (len(coll) == 0):
    return 'empty'
```

```
if is_empty(coll):
    return 'empty'
```

Pyjure

```
if (coll is None) or (len(coll) == 0):
    return None
else:
    return coll[0]
```

```
return first(coll)
```

Pyjure

- is_some
- is_empty
- first
- second
- last
- butlast
- nxt
- rest
- some
- identity

Life may be sweeter for this, I don't know

More pyfnz

- Option
- Observables
- Transducers
- more clj.core

github.com/
papaver/pyfnz

pip install pyfnz