# kingston

***Release 0.7.8***

**May 25, 2021**

# Contents:

Funcy Kingston, or "Kingston" for short is a Python library with many extras that I myself find useful. It depends on the excellent library funcy. Funcy is a library to make Python programming more functional.

The best way to get started quickly is to read the *README*, below:

*Release:* v0.7.8. (*Install instructions*) *Development:* v0.7.9.

CHAPTER 1

# Installing Kingston

*pip install kingston*

CHAPTER 2

# Kingston README

I use the excellent Funcy library for Python a lot. This is my collection of extras that I have designed to work closely together with funcy. Funcy Kingston (Reference, see here).

Run on Repl.it

Kingston is auto-formatted using yapf.

## 2.1 Pattern matching using extended `dict`'s

`match.Match` objects are callable objects using a `dict` semantic that also matches calls based on the type of the calling parameters:

```
>>> from kingston import match
>>> foo = match.TypeMatcher({
...     int: lambda x: x*100,
...     str: lambda x: f'Hello {x}'
... })
>>> foo(10)
1000
>>> foo('bar')
'Hello bar'
>>>
```

```
>>> from kingston import match
>>> foo = match.TypeMatcher({
...     int: lambda x: x * 100,
...     str: lambda x: f'Hello {x}',
...     (int, int): lambda a, b: a + b
... })
>>> foo(10)
1000
>>> foo('bar')
```

(continues on next page)

(continued from previous page)

```
'Hello bar'
>>>
>>> foo(1, 2)
3
>>>
```

You can use `typing.Any` as a wildcard:

```
>>> from typing import Any
>>> from kingston import match
>>> foo = match.TypeMatcher({
...     int: lambda x: x * 100,
...     str: (lambda x: f"Hello {x}"),
...     (int, Any): (lambda num, x: num * x)
... })
>>> foo(10)
1000
>>> foo('bar')
'Hello bar'
>>> foo(3, 'X')
'XXX'
>>> foo(10, 10)
100
>>>
```

You can also subclass type matchers and use a decorator to declare cases as methods:

```
>>> from kingston.match import Matcher, TypeMatcher, case
>>> from numbers import Number
>>> class NumberDescriber(TypeMatcher):
...     @case
...     def describe_one_int(self, one:int) -> str:
...         return "One integer"
...
...     @case
...     def describe_two_ints(self, one:int, two:int) -> str:
...         return "Two integers"
...
...     @case
...     def describe_one_float(self, one:float) -> str:
...         return "One float"
>>> my_num_matcher:Matcher[Number, str] = NumberDescriber()
>>> my_num_matcher(1)
'One integer'
>>> my_num_matcher(1, 2)
'Two integers'
>>> my_num_matcher(1.0)
'One float'
>>>
```

### 2.1.1 Typing pattern matchers

`match.Match` objects can be typed using Python's standard typing mechanism. It is done using Generics:

The two subtypes are *[argument type, return type]*.

```
>>> from kingston import match
>>> foo:match.Matcher[int, int] = match.TypeMatcher({
...     int: lambda x: x+1,
...     str: lambda x: 'hello'})
>>> foo(10)
11
>>> foo('bar')  # fails on mypy but would be ok at runtime
'hello'
>>>
```

### 2.1.2 Match by value(s)

`match.ValueMatcher` will use the *values* of the parameters to do the same as as `match.Match`:

```
>>> from kingston import match
>>> foo = match.ValueMatcher({'x': (lambda: 'An x!'), ('x', 'y'): (lambda x,y:␣
↪3*(x+y))})
>>> foo('x')
'An x!'
>>> foo('x', 'y')
'xyxyxy'
>>>
```

Same as with the type matcher above, `typing.Any` works as a wildcard with the value matcher as well:

```
>>> from kingston import match
>>> from typing import Any
>>> foo = match.ValueMatcher({
...     'x': lambda x: 'An X!',
...     ('y', Any): lambda x, y: 3 * (x + y)
... })
>>> foo('x')
'An X!'
>>> foo('y', 'x')
'yxyxyx'
>>>
```

You can also declare cases as methods in a custom `ValueMatcher` subclass.

Use the function `value_case()` to declare value cases. **Note:** *imported as a shorthand*:

```
>>> from kingston.match import Matcher, ValueMatcher
>>> from kingston.match import value_case as case
>>> class SimplestEval(ValueMatcher):
...     @case(Any, '+', Any)
...     def _add(self, a, op, b) -> int:
...         return a + b
...
...     @case(Any, '-', Any)
...     def _sub(self, a, op, b) -> int:
...         return a - b
>>> simpl_eval = SimplestEval()
>>> simpl_eval(1, '+', 2)
3
>>> simpl_eval(10, '-', 5)
```

(continues on next page)

(continued from previous page)

```
5
>>>
```

## 2.2 Aspect Oriented Programming with terse syntax

Kingston also implement a technique to do AOP with an opinionated terse syntax that I like. It lives in the `kingston.aop` module.

It's used in two main ways:

### 2.2.1 With decorators

Define an =`Aspects`= object as an empty object:

```
>>> from kingston.aop import Aspects
>>> when = Aspects()
>>>
```

Then declare your aspects using the object as a decorator:

```
>>> @when(lambda x: x == 1, y=lambda y: y == 1)
... def labbo(x, y=1):
...     return 11
>>> @when(lambda x: x == 1, z=lambda z: z == 2)
... def labbo(x, z=2):
...     return 12
>>>
```

Aspect 1 above will be triggered if you call it with positional parameter 0 as `1` and a keyword parameter `y=1`:

```
>>> labbo(1, y=1)
11
>>>
```

Aspect 2 is triggered by parameters `1, z=2`:

```
>>> labbo(1, z=2)
12
>>>
```

Any other combination of parameters will raise a `AspectNotFound` exception:

```
>>> labbo(123)
Traceback (most recent call last):
AspectNotFound
>>>
>>>
```

### 2.2.2 With a mapping of aspects

You might find this better if you want brievity and/or point free style.

```
>>> given = Aspects({
...     (lambda x: x == 1,): lambda x: 1,
...     (lambda x: x > 1,): lambda x: x * x
... })
>>>
```

Calls work the same as above:

```
>>> given(1)
1
>>> given(2)
4
>>> given(0)
Traceback (most recent call last):
AspectNotFound
>>>
```

## 2.3 Nice things

### 2.3.1 dig()

Deep value grabbing from almost any object. Somewhat inspired by CSS selectors, but not very complete. This part of the API is unstable — it will (hopefully) be developed further in the future.

```
>>> from kingston import dig
>>> dig.xget((1, 2, 3), 1)
2
>>> dig.xget({'foo': 'bar'}, 'foo')
'bar'
>>> dig.dig({'foo': 1, 'bar': [1,2,3]}, 'bar.1')
2
>>> dig.dig({'foo': 1, 'bar': [1,{'baz':'jox'},3]}, 'bar.1.baz')
'jox'
>>>
```

The difference between `dig.dig()` and `funcy.get_in()` is that you can use shell-like blob patterns to get several values keyed by similar names:

```
>>> from kingston import dig
>>> res = dig.dig({'foo': 1, 'foop': 2}, 'f*')
>>> res
[foo=1:int, foop=2:int]
>>> # (textual representation of an indexable object)
>>> res[0]
foo=1:int
>>> res[1]
foop=2:int
>>>
```

## 2.4 Testing tools

Kingston has some testing tools as well. Also, due to Kingston's opinionated nature, they are only targeted towards pytest.

### 2.4.1 Shortform for pytest.mark.parametrize

I tend to use pytest.mark.parametrize in the same form everywhere. Thus I have implemented this short-form:

```
>>> from kingston.testing import fixture
>>> @fixture.params(
...     "a, b",
...     (1, 1),
...     (2, 2),
... )
... def test_dummy_compare(a, b):
...     assert a == b
>>>
```

### 2.4.2 Doctests as fixtures

There is a test decorator that generates pytest fixtures from a function or an object. Use it like this:

```
>>> def my_doctested_func():
...   """
...   >>> 1 + 1
...   2
...   >>> mystring = 'abc'
...   >>> mystring
...   'abc'
...   """
...   pass
>>> from kingston.testing import fixture
>>> @fixture.doctest(my_doctested_func)
... def test_doctest_my_doctested(doctest):  # fixture name always 'doctest'
...     res = doctest()
...     assert res == '', res
>>>
```

CHAPTER 3

# Kingston Changelog

## 3.1 0.7.8

- Slight refactor / yak shave & fix version
- Redesigns kingston.match.matches() to ensure that marker values ‘Any‘ and ‘... ‘ are never sent to the comparision function. Fixes seldom triggered edge-case eg when you want to feed values straight to the ‘re‘ module.
- Implements sensible ‘str()‘ and ‘repr()‘ handlers for ‘kingston.match.Matcher‘ objects.
- Shaves off 1 iteration when matching against ‘... ‘ marker values.
- A few internal naming improvements.
- Filled in missing typing stub for ‘kingston.aop‘

## 3.2 0.7.7

- Fix version, fixes an edge case when type matching against sequences with exactly 1 value.

## 3.3 0.7.6

- Require that recursive `match.TypeMatcher`’s are declared explicitly case by case.

## 3.4 0.7.5

- Implements a mechanism for AOP with terse syntax
- Small internal refinements

## 3.5 0.7.4

- Implements subtype matching in match.TypeMatcher
- More options for devtool.PrintfDebugging
- Tiny style fixes

## 3.6 0.7.3

- More readable error messages in case of `Mismatch` exception from matchers.
- Implemented new notation for matchers as subclasses where cases are declared using a decorator.

## 3.7 0.7.2

- Fix version, incorrect imports in `kingston.testing` could cause false positives in CI settings.

## 3.8 0.7.1

- Fix version, invalid metadata in `setup.py`

## 3.9 0.7.0

- The module `kingston.match` can now do a wildcard match using `...` (`Ellipsis`) objects, i.e. an arbitrary long group of `Any` matchings.
- Many more refactoring and fixes in `kingston.match`
- Testing utility `kingston.testing.fixture` and extract and run doctests as pytest fixtures.
- Dropped the homegrown pipe operator overloading mechanism, use SSPipe instead.
- Can build as a Conda Package.
- Dropped obsolete dependencies, e.g. `pysistence`.
- More extensive usage of MyPy gradual typing mechanism.

## 3.10 0.6.8

- Fixes for `kingston.match`
- `kingston.testing.trial()` / `kingston.testing.retryit()`, moved to `kingston.devtool`.

## 3.11 0.6.7

- Bugfix in `kingston.match.Match.case()`

## 3.12 0.6.6

- Polish release, mosly QA work
- Smaller bugfixes
- Coverage analysis with pytest-cov
- Trimmed code base after coverage analysis

## 3.13 0.6.5

- New module `kingston.match`, a mechanism for *"pattern matching"* using subclasses of `dict`'s to store patterns and references to `callable`'s.

## 3.14 0.6.4

- Built a more formal project structure.
- Started to use light-weight CI in the form of a GitHub action invoking Tox.

## 3.15 0.6.3

- Project renamed to *"Kingston"* and re-licenced under LGPL v3

CHAPTER 4

# Examples

## 4.1 Tiny AST to code generation thingy

A key feature of Kingston is the pattern matching in kingston.match.

To get an idea of what you can build, let's create a rudimentary AST to Python code generator. This tends to be a quite daunting exercise.

### 4.1.1 Imports needed

You will want to import `ast`, `kingston.match.Matcher` and `kingston.match.TypeMatcher`:

```
>>> import ast
>>> from kingston.match import Matcher, TypeMatcher
```

### 4.1.2 Configure a `Matcher` to convert *ast.AST* nodes to strings

```
>>> nodeRep:Matcher[ast.AST, str] = TypeMatcher({
...     ast.Interactive: lambda: '',
...     ast.FunctionDef: lambda node: f"def {node.name}(",
...     ast.arguments: (lambda node: ','.join(arg.arg
...                                           for arg in node.args) +
...                     '):\n'),
...     ast.BinOp: lambda node: '',
...     ast.Constant: lambda node: str(node.value),
...     ast.Return: lambda node: '    return ',
...     ast.Add: lambda: ' + ',
... })
```

As you can see, this isn't recursive and will be very primitive. We will borrow `ast.walk()` for brevity. Note the typing declaration at the top line. This is to be able to use MyPy to type check our matchings. Here the declaration means that the matcher accepts `ast.AST` nodes as parameters and will return `str`.

### 4.1.3 Compile a small AST

```
>>> topnode = compile("""
... def helo():
...     return 1 + 1
... """, 'examples.ormsnackis', 'single', ast.PyCF_ONLY_AST)
```

Given the `Matcher` we just created we could only support the most minimal AST.

### 4.1.4 Test it

We use the linear list of nodes you get from AST's walk function to test without too much work:

```
>>> def test(tree):
...     print(''.join(nodeRep(node) for node in ast.walk(tree)))

>>> if __name__ == '__main__':
...     test(topnode)
```

Running gives the following output:

```
$ python examples/ormsnackis.py
def helo():
    return 1 + 1
$
```

CHAPTER 5

# API Reference

To read about the nitty-gritty inside Kingston, you can explore the API documentation:

## 5.1 API Reference

### 5.1.1 Matching module

#### The Match module

This module implements a technique for pattern matching.

IDEA: Statement-oriented matcher within the same scope: with ValueMatcher('sum', 1, 2, 3) as m:

**with m.case('sum', Any, Any, Any) as _, one, two, three:** print(one + two + three)

**with m.case('avg', ...) as _, *values:** print(sum(*values))

**with.m.miss() as *missed:** print('MISS!')

#### High-level functions

`kingston.match.`**`matches`**(*values: Sequence[T_co], patterns: Sequence[T_co], matchfn: Callable = <function match>*) → Sequence[T_co]

Tries to match `values` from `patterns`.

**Parameters**

- **`values`** – A sequence of values to match.
- **`patterns`** – A sequence of patterns that may match `values`.

**Returns** The pattern that was matched or `Miss`.

**Return type** Union[Sequence, Type[Miss]]

### High-level classes

**class** kingston.match.**Matcher**
Common base for all matcher classes.

Since `Matcher` is also `Generic`, you use it to subtype concrete instances of matchers you implement.

**class** kingston.match.**TypeMatcher**
Concrete implementation of a type matcher instance.

If you want to type a type matcher, use standard technique when using `Generic` types:

```
>>> from kingston.match import Matcher, TypeMatcher
>>> my_int_matcher:Matcher[int, int] = TypeMatcher({
...     int: lambda x: x+1,
...     str: lambda x: 'str'})
>>> my_int_matcher(10)
11
>>> my_int_matcher(20)
21
>>> my_int_matcher('foo')  # ok at runtime but fails mypy
'str'
>>>
```

It will try to give a reasonably human representation when inspected:

```
>>> my_int_matcher
<TypeMatcher: (int)->λ, (str)->λ >
>>>
```

You can also subclass type matchers and use a decorator to declare cases as methods:

```
>>> from kingston.match import Matcher, TypeMatcher, case
>>> from numbers import Number
>>> class NumberDescriber(TypeMatcher):
...     @case
...     def describe_one_int(self, one:int) -> str:
...         return "One integer"
...
...     @case
...     def describe_two_ints(self, one:int, two:int) -> str:
...         return "Two integers"
...
...     @case
...     def describe_one_float(self, one:float) -> str:
...         return "One float"
>>> my_num_matcher:Matcher[Number, str] = NumberDescriber()
>>> my_num_matcher(1)
'One integer'
>>> my_num_matcher(1, 2)
'Two integers'
>>> my_num_matcher(1.0)
'One float'
>>>
```

**class** kingston.match.**ValueMatcher**
Concrete implementation of a value matching instance.

If you want to type a type matcher, use standard technique when using `Generic` types:

```
>>> from kingston.match import ValueMatcher, Miss
>>> my_val_matcher:Matcher[int, str] = ValueMatcher({
...     1: lambda x: 'one!',
...     2: lambda x: 'two!',
...     Miss: lambda x: 'many!'})
>>> my_val_matcher(1)
'one!'
>>> my_val_matcher(2)
'two!'
>>> my_val_matcher(3)
'many!'
>>> my_val_matcher('x')  # ok at runtime but fails mypy (& missleading..)
'many!'
>>>
```

It will try to give a reasonably human representation when inspected:

```
>>> my_val_matcher
<ValueMatcher: (1)->λ, (2)->λ, (<class 'kingston.match.Miss_'>)->λ >
>>>
```

You can also declare cases as methods in a custom `ValueMatcher` subclass.

Use the function `value_case()` to declare value cases. **Note:** *imported as a shorthand*:

```
>>> from kingston.match import Matcher, ValueMatcher
>>> from kingston.match import value_case as case
>>> class SimplestEval(ValueMatcher):
...     @case(Any, '+', Any)
...     def _add(self, a, op, b) -> int:
...         return a + b
...
...     @case(Any, '-', Any)
...     def _sub(self, a, op, b) -> int:
...         return a - b
>>> simpl_eval = SimplestEval()
>>> simpl_eval(1, '+', 2)
3
>>> simpl_eval(10, '-', 5)
5
```

### Exceptions and symbols

**exception** kingston.match.**Mismatch**
Exception to signal matching error in Matcher objects.

**exception** kingston.match.**Conflict**
Exception raised if a pattern to be matched already have been applied to a *Matcher* instance.

kingston.match.**Miss**

**class** kingston.match.**NoNextValue**
Symbol signifying that no more values are available to pattern check for.

**class** kingston.match.**NoNextAnchor**
Symbol signifying that no more anchor values exist in a pattern.

#### Low-level functions

kingston.match.**match**(*cand: Any*, *pattern: Any*) → bool
*"Primitive"* function that checks an individual value against another. The `match()` function is *only* responsible for checking two values, matching markers *Any* and *Ellipsis* are handled elsewhere.

kingston.match.**move**(*matched: Sequence[T_co]*, *pending: Sequence[T_co]*, *matchfn: Callable = <function match>*) → Tuple[Sequence[T_co], Sequence[T_co]]

### 5.1.2 Dig - fetch/query in live objects

This module if for reading values from live objects. The general idea is that you use a function `kingston.dig.dig()` with an object and a string. The string is (somewhat) inspired by a CSS selector. It specifies how to get a certain sub-element.

The goal is that these spec-strings should be storable for future re-use.

This module is a work in progress module and thus subject to change.

#### High-level functions

kingston.dig.**dig**(*obj: Any*, *path: str*) → Any
Dig after object content from object content based on a string spec.

**Parameters**

- **obj** – A live object that values should be digged from.
- **path** – String representation of the *"path"*

kingston.dig.**xget**(*obj: Any*, *idx: Any*) → Any
Single point of entry function to fetch a value / attribute / element from an object.

**Parameters**

- **obj** – The object to find attribute / value in.
- **idx** – Symbolic index.

CHAPTER 6

# Indices and tables

- *Kingston README*
- search
- *Installing Kingston*

# Python Module Index

## k

# Index