function-pipe Documentation

Release 2.1.0

Christopher Ariza

May 20, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

The function-pipe Python module defines the class FunctionNode (FN) and decorators to create the derived-class PipeNode (PN). FNs are wrappers of callables that permit returning new FNs after applying operators, composing callables, or partialing. This supports the flexible combination of functions in a lazy and declarative manner.

PipeNodes (PNs) are FNs prepared for extended function composition or dataflow programming. PNs, through a decorator-provided, two-stage calling mechanism, expose to wrapped functions both predecessor output and a common initial input. Rather than strictly linear pipelines, sequences of PNs can be stored and reused; PNs can be provided as arguments to other PNs; and results from PNs can be stored for later recall in the same pipeline.

Code: https://github.com/InvestmentSystems/function-pipe

Docs: http://function-pipe.readthedocs.io

Packages: https://pypi.python.org/pypi/function-pipe

1.1 Getting Started

FunctionNode and PipeNode are abstract tools for linking Python functions, and are applicable in many domains. The best way to get to know them is to follow some examples and experiment. This documentation provide examples in a number of domains, including processing strings and Pandas DataFrames.

1.2 Installation

A standard setuptools installer is available via PyPI:

https://pypi.python.org/pypi/function-pipe

Or, install via pip3:

```
pip3 install function-pipe
```

Source code can be obtained here:

https://github.com/InvestmentSystems/function-pipe

1.3 History

The function-pipe tools were developed within Investment Systems, the core development team of Research Affiliates, LLC. Many of the approaches implemented were first created by Max Moroz in 2012. Christopher Ariza subsequently refined and extended those approaches into the current models of FunctionNode and PipeNode. The first public release of function-pipe was in January 2017. After that, Charles Burkland improved the quality and user-friendliness of the library, along with the addition of some more explicit functions. The second public release was made in April 2022.

1.4 Related

The function-pipe tools, consisting of one module, offers a very focused and light-weight approach to extended function composition in Python. There are many other tools that offer similar and/or broader resources. A few are listed below.

https://pypi.python.org/pypi/fn

https://pypi.python.org/pypi/functional

https://pypi.python.org/pypi/lusmu

https://github.com/manahl/mdf

https://pypi.python.org/pypi/pipe

https://pypi.python.org/pypi/pipetools

https://pypi.python.org/pypi/PyFunctional

https://pypi.python.org/pypi/PyMonad

CHAPTER

TWO

API

2.1 function_pipe

class FunctionNode(*function: Any*, **, doc_function: Optional[Callable] = None*, *doc_args: Tuple[Any, ...] = ()*, *doc_kwargs: Optional[Dict[str, Any]] = None*)

A wrapper for a callable that can reside in an expression of numerous FunctionNodes, or be modified with unary or binary operators.

__init__(*function: Any*, **, doc_function: Optional[Callable] = None*, *doc_args: Tuple[Any, ...] = ()*, *doc_kwargs: Optional[Dict[str, Any]] = None*) → None

Args:

- `function`: a callable or value. If given a value, will create a function that simply returns that value.
- `doc_function`: the function to display in the repr; will be set to `function` if not provided
- `doc_args`: the positional arguments to display in the repr
- `doc_kwargs`: the keyword arguments to display in the repr

property unwrap: Callable

The doc_function should be set to the core function being wrapped, no matter the level of wrapping.

__call__(**args: Any*, ***kwargs: Any*) → Any

Call the wrapped function with args and kwargs.

partial(**args: Any*, ***kwargs: Any*) → *function_pipe.core.function_pipe.FunctionNode*

Return a new FunctionNode with a partialed function with args and kwargs.

__neg__() → function_pipe.core.function_pipe.FN

Return a new FunctionNode that when evaulated, will negate the result of `self`

__invert__() → function_pipe.core.function_pipe.FN

Return a new FunctionNode that when evaulated, will invert the result of `self`

NOTE: This is generally expected to be a Boolean inversion, such as ~ (not) applied to a Numpy, Pandas, or Static-Frame objects.

__abs__() → function_pipe.core.function_pipe.FN

Return a new FunctionNode that when evaulated, will find the absolute value of the result of `self`

__eq__(*rhs: Any*) → function_pipe.core.function_pipe.FN

Return self==value.

__lt__(*rhs: Any*) → function_pipe.core.function_pipe.FN

Return self<value.

__le__(*rhs: Any*) → function_pipe.core.function_pipe.FN

Return self<=value.

__gt__(*rhs: Any*) → function_pipe.core.function_pipe.FN

Return self>value.

__ge__(*rhs: Any*) → function_pipe.core.function_pipe.FN

Return self>=value.

__ne__(*rhs: Any*) → function_pipe.core.function_pipe.FN

Return self!=value.

__rshift__(*rhs: Callable*) → function_pipe.core.function_pipe.FN

Composes a new FunctionNode will call `lhs` first, and then feed its result into `rhs`

__rrshift__(*lhs: Callable*) → function_pipe.core.function_pipe.FN

Composes a new FunctionNode will call `lhs` first, and then feed its result into `rhs`

__lshift__(*rhs: Callable*) → function_pipe.core.function_pipe.FN

Composes a new FunctionNode will call `rhs` first, and then feed its result into `lhs`

__rlshift__(*lhs: Callable*) → function_pipe.core.function_pipe.FN

Composes a new FunctionNode will call `rhs` first, and then feed its result into `lhs`

__or__(*rhs: function_pipe.core.function_pipe.FN*) → function_pipe.core.function_pipe.FN

Only implemented for PipeNode.

__ror__(*lhs: function_pipe.core.function_pipe.FN*) → function_pipe.core.function_pipe.FN

Only implemented for PipeNode.

class PipeNode(*function: Any*, *, *doc_function: Optional[Callable] = None*, *doc_args: Tuple[Any, ...] = ()*, *doc_kwargs: Optional[Dict[str, Any]] = None*, *call_state: Optional[function_pipe.core.function_pipe.PipeNode.State] = None*, *predecessor: Optional[function_pipe.core.function_pipe.PN] = None*)

This encapsulates the node that will be used in a pipeline.

It is not expected to be created directly, rather, through usage of `pipe_node` (and related) decorators.

PipeNodes will be in (or move between) one of three states, depending on where it was created, or what the current state of pipeline evaluation is

partial(**args: str*, ***kwargs: str*) → function_pipe.core.function_pipe.PN

Partialing PipeNodes is prohibited. Use `pipe_node_factory` (and related) decorators to pass in expression-level arguments.

property call_state: Optional[State]

The current call state of the Node

property predecessor: Optional[function_pipe.core.function_pipe.PN]

The PipeNode preceeding this Node in a pipeline. Can be None

__or__(*rhs: function_pipe.core.function_pipe.PN*) → function_pipe.core.function_pipe.PN

Invokes `rhs`, passing in `self` as the kwarg `PREDECESSOR_PN`.

__ror__(*lhs: function_pipe.core.function_pipe.PN*) → function_pipe.core.function_pipe.PN

Invokes `lhs`, passing in `self` as the kwarg `PREDECESSOR_PN`.

__getitem__(*pn_input: Any*) → Any

Invokes `self`, passing in `pn_input` as the kwarg `PN_INPUT`.

NOTE:

- If `None`, will evaluate self with a default `PipeNodeInput` instance
- If user desires for the initial input to be literally `None`, use `(**{PN_INPUT: None})` instead.

__call__(**args: Any, **kwargs: Any*) → Any

Call the wrapped function with args and kwargs.

class PipeNodeInput

PipeNode input to support store and recall; subclassable to expose other attributes and parameters.

store(*key: str, value: Any*) → None

Store `key` and `value` in the underlying store.

recall(*key: str*) → Any

Recall `key` from the underlying store. Can raise an `KeyError`

property store_items: ItemsView[str, Any]

Return an items view of the underlying store.

pipe_node(**key_positions: typing.Union[typing.Callable, str], core_decorator: typing.Callable[[typing.Any], typing.Callable] = <function _core_logger>, self_keyword: str = 'self'*) → Union[Callable, *function_pipe.core.function_pipe.PipeNode*]

Decorates a function to become a `PipeNode` that takes no expression-level args.

This can either be used as a decorator, or a decorator factory, similar to `functools.lru_cache`.

Examples:

```
>>> @pipe_node
>>> def func(**kwargs):
>>>     pass
```

```
>>> @pipe_node()
>>> def func():
>>>     pass
```

```
>>> @pipe_node(PN_INPUT)
>>> def func(pn_input):
>>>     pass
```

```
>>> class Example:
>>>     @pipe_node(PN_INPUT)
>>>     def method(self, pn_input):
>>>         pass
```

```
>>> from functools import partial
>>> español_pipe_node = partial(pipe_node, self_keyword="uno_mismo")
>>> ...
>>> class Ejemplo:
>>>     @español_pipe_node(PN_INPUT)
>>>     def método(uno_mismo, pn_input):
>>>         pass
```

Args:

- `key_positions`: either a single callable, or a list of keywords that will be positionally bound to the decorated function.
- `core_decorator`: a decorator that will be applied to the core_callable. This is typically a logger. By default, it will print to stdout.
- `self_keyword`: which keyword to look for when decorating instance methods.

`pipe_node_factory`(**key_positions: typing.Union[typing.Callable, str], core_decorator: typing.Callable[[typing.Any], typing.Callable] = <function _core_logger>, self_keyword: str = 'self'*) → Union[Callable, Callable[[Any], *function_pipe.core.function_pipe.PipeNode*]]

Decorates a function to become a pipe node factory, that when given *expression-level* arguments, will return a `PipeNode`

This can either be used as a decorator, or a decorator factory, similar to `functools.lru_cache`.

Examples:

```
>>> @pipe_node_factory
>>> def func(a, b, **kwargs):
>>>     pass
>>> ...
>>> func(1, 2) # This is now a PipeNode!
```

```
>>> @pipe_node_factory()
>>> def func(*, a, b):
>>>     pass
>>> ...
>>> func(a=1, b=2) # This is now a PipeNode!
```

```
>>> @pipe_node_factory(PN_INPUT, PREDECESSOR_RETURN)
>>> def func(pn_input, previous_value, a, *, b):
>>>     # pn_input will be given the PN_INPUT from the pipeline
>>>     # prev will be given the PREDECESSOR_RETURN from the pipeline
>>>     pass
>>> ...
>>> func(1, b=2) # This is now a PipeNode!
```

```
>>> class Example:
>>>     @pipe_node_factory(PN_INPUT, PREDECESSOR_RETURN)
>>>     def method(self, pn_input, previous_value, a, *, b):
>>>         pass
>>> ...
>>> Example().method(1, b=2) # This is now a PipeNode!
```

```
>>> from functools import partial
>>> español_pipe_node_factory = partial(pipe_node_factory, self_keyword="uno_mismo")
>>> ...
>>> class Ejemplo:
>>>     @español_pipe_node_factory(PN_INPUT, PREDECESSOR_RETURN)
>>>     def método(uno_mismo, pn_input, valor_anterior, a, *, b):
>>>         pass
```

(continues on next page)

(continued from previous page)

```
>>> ...
>>> Ejemplo().método(1, b=2) # Esto ahora es un PipeNode!
```

Args:

- `key_positions`: either a single callable, or a list of keywords that will be positionally bound to the decorated function.
- `core_decorator`: a decorator that will be applied to the core_callable. This is typically a logger. By default, it will print to stdout.
- `self_keyword`: which keyword to look for when decorating instance methods.

compose(**funcs: Callable*) → function_pipe.core.function_pipe.FN

Given a list of functions, execute them from right to left, passing the returned value of the right f to the left f. Store the reduced function in a FunctionNode

classmethod_pipe_node(**key_positions: typing.Union[typing.Callable, str], core_decorator: typing.Callable[[typing.Any], typing.Callable] = <function _core_logger>*) → Union[Callable, *function_pipe.core.function_pipe.PipeNode*]

Decorates a function to become a classmethod `PipeNode` that takes no expression-level args.

This can either be used as a decorator, or a decorator factory, similar to `functools.lru_cache`.

This is a convenience method, that is the mental equivalent to this pseudo-code:

```
>>> @classmethod
>>> @pipe_node(...)
>>> def func(...)
```

Examples:

```
>>> @classmethod_pipe_node
>>> def func(cls, **kwargs):
>>>     pass
```

```
>>> @classmethod_pipe_node()
>>> def func(cls):
>>>     pass
```

```
>>> @classmethod_pipe_node(PN_INPUT)
>>> def func(cls, pn_input):
>>>     pass
```

Args:

- `key_positions`: either a single callable, or a list of keywords that will be positionally bound to the decorated function.
- `core_decorator`: a decorator that will be applied to the core_callable. This is typically a logger. By default, it will print to stdout.

classmethod_pipe_node_factory(**key_positions: typing.Union[typing.Callable, str], core_decorator: typing.Callable[[typing.Any], typing.Callable] = <function _core_logger>*) → Callable

Decorates a function to become a classmethod pipe node factory, that when given *expression-level* arguments, will return a `PipeNode`

This can either be used as a decorator, or a decorator factory, similar to `functools.lru_cache`.

This is a convenience method, that is the mental equivalent to this pseudo-code:

```
>>> @classmethod
>>> @pipe_node_factory(...)
>>> def func(...)
```

Examples:

```
>>> @classmethod_pipe_node_factory
>>> def func(cls, a, b, **kwargs):
>>>     pass
>>> ...
>>> SomeClass.func(1, 2) # This is now a PipeNode!
```

```
>>> @classmethod_pipe_node_factory()
>>> def func(cls, *, a, b):
>>>     pass
>>> ...
>>> SomeClass.func(a=1, b=2) # This is now a PipeNode!
```

```
>>> @classmethod_pipe_node_factory(PN_INPUT, PREDECESSOR_RETURN)
>>> def func(cls, pn_input, previous_value, a, *, b):
>>>     # ``pn_input`` will be given the PN_INPUT from the pipeline
>>>     # ``previous_value`` will be given the PREDECESSOR_RETURN from the pipeline
>>>     pass
>>> ...
>>> SomeClass.func(1, b=2) # This is now a PipeNode!
```

Args:

- `key_positions`: either a single callable, or a list of keywords that will be positionally bound to the decorated function.
- `core_decorator`: a decorator that will be applied to the core_callable. This is typically a logger. By default, it will print to stdout.

staticmethod_pipe_node(**key_positions: typing.Union[typing.Callable, str], core_decorator: typing.Callable[[typing.Any], typing.Callable] = <function _core_logger>*) → Union[Callable, *function_pipe.core.function_pipe.PipeNode*]

Decorates a function to become a staticmethod `PipeNode` that takes no expression-level args.

This can either be used as a decorator, or a decorator factory, similar to `functools.lru_cache`.

This is a convenience method, that is the mental equivalent to this pseudo-code:

```
>>> @staticmethod
>>> @pipe_node(...)
>>> def func(...)
```

Examples:

```
>>> @staticmethod_pipe_node
>>> def func(**kwargs):
>>>     pass
```

```
>>> @staticmethod_pipe_node()
>>> def func():
>>>     pass
```

```
>>> @staticmethod_pipe_node(PN_INPUT)
>>> def func(pn_input):
>>>     pass
```

Args:

- `key_positions`: either a single callable, or a list of keywords that will be positionally bound to the decorated function.
- `core_decorator`: a decorator that will be applied to the core_callable. This is typically a logger. By default, it will print to stdout.

staticmethod_pipe_node_factory(**key_positions: typing.Union[typing.Callable, str], core_decorator: typing.Callable[[typing.Any], typing.Callable] = <function _core_logger>*) → Callable

Decorates a function to become a staticmethod pipe node factory, that when given *expression-level* arguments, will return a `PipeNode`

This can either be used as a decorator, or a decorator factory, similar to `functools.lru_cache`.

This is a convenience method, that is the mental equivalent to this pseudo-code:

```
>>> @staticmethod
>>> @pipe_node_factory(...)
>>> def func(...)
```

Examples:

```
>>> @staticmethod_pipe_node_factory
>>> def func(a, b, **kwargs):
>>>     pass
>>> ...
>>> SomeClass.func(1, 2) # This is now a PipeNode!
```

```
>>> @staticmethod_pipe_node_factory()
>>> def func(*, a, b):
>>>     pass
>>> ...
>>> SomeClass.func(a=1, b=2) # This is now a PipeNode!
```

```
>>> @staticmethod_pipe_node_factory(PN_INPUT, PREDECESSOR_RETURN)
>>> def func(pn_input, previous_value, a, *, b):
>>>     # ``pn_input`` will be given the PN_INPUT from the pipeline
>>>     # ``previous_value`` will be given the PREDECESSOR_RETURN from the pipeline
>>>     pass
```

(continues on next page)

(continued from previous page)

```
>>> ...
>>> SomeClass.func(1, b=2) # This is now a PipeNode!
```

Args:

- `key_positions`: either a single callable, or a list of keywords that will be positionally bound to the decorated function.
- `core_decorator`: a decorator that will be applied to the core_callable. This is typically a logger. By default, it will print to stdout.

store(*pni:* PipeNodeInput, *ret_val: tp.Any*, *label: str*) → tp.Any:

Store `ret_val` (the value returned from the previous `PipeNode`) to `pni` under `label`. Forward `ret_val`.

recall(*pni:* PipeNodeInput, *label: str*) → tp.Any:

Recall `label` from `pni``` and return it. Can raise an `KeyError`

call(**pns:* PipeNode) → tp.Any

Broadcasts `pns`, and returns the result of `pns[-1]`

Since `pns` are all `PipeNodes`, they will all be evaluated before passed in as values to this function.

pretty_repr(*f: Any*) → str

Provide a pretty string representation of a FN, PN, or anything. If the object is a FN or PN, it will recursively represent any nested FNs/PNs.

is_unbound_self_method(*core_callable: Union[classmethod, staticmethod, Callable]*, **, self_keyword: str*) → bool

Inspects a given callable to determine if it's both unbound, and the first argument in its signature is `self_keyword`

CHAPTER

THREE

INTUITIVELY UNDERSTANDING PIPELINES

Tutorial Alias

PN: pipe node

This tutorial will teach the foundational concepts of `function_pipe` PN pipelines through clear and intuitive steps. After reading, you will:

- Know how to build and use a PN and/or PN pipeline
- Understand the different between the **creation** and **evaluation** phase of PN
- Understand how to link PNs together
- Understand what a PN input is, and how to share data across PNs
- Be able to debug issues in your own PNs

3.1 Introduction

Function pipelines happen in two stages: **creation** & **evaluation**.

Creation is the step in which a pipeline is defined, understood as either a single PN, or multiple PNs chained together using the `|` operator. Here is a pseudo-code example of this:

```
pipeline = (pn_a | pn_b | pn_c | ...)

# OR

pipeline = pn
```

Evaluation is the step in which the pipeline is actually called, where the function code inside each PN is actually run:

```
pipeline["initial input"] # Evaluate the pipeline by using __getitem__, and passing in␣
↪some initial input
```

3.2 Visualizing the Distinction Between Creation & Evaluation

To get started, we will create two simple PNs, put them into a pipeline expression, and then evaluate that expression. **Creation** followed by **evaluation**.

To do this, we will use the `fpn.pipe_node` decorator, and define methods which take `**kwargs`. (`**kwargs` will be explained later!)

```
import function_pipe as fpn  # Import convention!

@fpn.pipe_node
def pipe_node_1(**kwargs):
   print("pipe_node_1 has been evaluated")

@fpn.pipe_node
def pipe_node_2(**kwargs):
   print("pipe_node_2 has been evaluated")

print("Start creation")
pipeline = (pipe_node_1 | pipe_node_2)
print("End creation")

print("Start pipeline evaluation")
pipeline[None]
print("End pipeline evaluation")
```

Now, let's see the output that happens were we to run the previous code.

```
Start creation
End creation
Start pipeline evaluation
| <function pipe_node_1 at 0x7f582c428ca0>
pipe_node_1 has been evaluated
| <function pipe_node_2 at 0x7f582c428b80>
pipe_node_2 has been evaluated
End pipeline evaluation
```

As you can see, none of the PNs are called (**evaluated**) until the pipeline expression itself was **created** and then invoked.

3.3 What Is The Deal With Kwargs

In the previous example, we used `**kwargs` on each function (if we hadn't, the code would have failed!) Why did we need this, and what are they? Let's investigate!

To investigate, we will build up a slightly longer pipeline, and expand the nodes to return some values

```
@fpn.pipe_node
def pipe_node_1(**kwargs):
   print(kwargs)
   return 1

@fpn.pipe_node
```

(continues on next page)

(continued from previous page)

```
def pipe_node_2(**kwargs):
    print(kwargs)
    return 2

@fpn.pipe_node
def pipe_node_3(**kwargs):
    print(kwargs)
    return 3

pipeline = (pipe_node_1 | pipe_node_2 | pipe_node_3)
assert pipeline["original_input"] == 3

print(f"repr(pipeline) = '{repr(pipeline)}'")
```

Running the above code will produce the following output:

```
| <function pipe_node_1 at 0x7f582cceb700>
{"pn_input": "original_input"}
| <function pipe_node_2 at 0x7f582c2d30d0>
{"pn_input": "original_input", "predecessor_pn": <PN: pipe_node_1>, "predecessor_return
↪": 1}
| <function pipe_node_3 at 0x7f582c33b820>
{"pn_input": "original_input", "predecessor_pn": <PN: pipe_node_1 | pipe_node_2>,
↪"predecessor_return": 2}
repr(pipeline) = '<PN: pipe_node_1 | pipe_node_2 | pipe_node_3>'
```

There are a few things happening here worth observing.

1) Every node is given the kwarg `pn_input`.

2) Each node (except the first), is given the kwargs `predecessor_pn` and `predecessor_return`

The first node is special. In the context of the pipeline it lives in, there are no PNs preceding it, hence `predecessor_pn` and `predecessor_return` are not passed in!

For every other node, it is initiutive what the values of `predecessor_pn` and `predecessor_return` will be. They contain the node instance of the one before, and the return value of that node once it's evaluated.

As we can observe on `pipe_node_3`, the repr of `predecessor_pn` shows how it's predecessor is actually a pipeline of PNs instead of a single PN. Additionally, printing the repr of `pipeline` shows how it is a pipeline of multiple PNs.

Note: From now on, we will refer to the three strings above by their symbolic constant handles in the **function_pipe** module. They are `fpn.PN_INPUT`, `fpn.PREDECESSOR_PN`, and `fpn.PREDECESSOR_RETURN`, respectively.

3.4 Using the Kwargs

Now that we know what will be passed in through each PN's `**kwargs` based on where it is in the pipeline, let's write some code that takes advantage of that.

```
@fpn.pipe_node
def multiply_input_by_2(**kwargs):
   return kwargs[fpn.PN_INPUT] * 2

@fpn.pipe_node
def add_7(**kwargs):
   return kwargs[fpn.PREDECESSOR_RETURN] + 7

@fpn.pipe_node
def divide_by_3(**kwargs):
   return kwargs[fpn.PREDECESSOR_RETURN] / 3

pipeline_1 = (multiply_input_by_2 | add_7 | divide_by_3)
assert pipeline_1[12] == (((12 * 2) + 7) / 3)

pipeline_2 = (multiply_input_by_2 | divide_by_3 | add_7)
assert pipeline_2[12] == (((12 * 2) / 3) + 7)
```

As you can see, PNs have the ability to use the return values from their predecessors, or the `fpn.PN_INPUT` whenever they need to.

You can also observe that `pipeline_2` reversed the order of the latter two PNs from their order in `pipeline_1`. This worked seamlessly, since each of the PNs was accessing information from the predecessor's return value. Had we tried something like:

```
pipeline_3 = (add_7 | multiply_input_by_2 | divide_by_3)
pipeline_3[12]
```

it would have failed, since the first PN is *never* given `fpn.PREDECESSOR_RETURN` as a kwarg.

Note: `fpn.PREDECESSOR_PN` is a kwarg that is almost never used in regular PNs or pipelines. If you are reaching for this kwarg, you are probably doing something wrong! It's primary purpose is to ensure the internals of the **function_pipe.PipeNode** module are working properly, not for use by end users.

3.5 Hiding the Kwargs

Now that we know how to use `**kwargs`, we can see that manually extracting the pipeline kwargs we care about each time is not good! On top of that, it's highly undesirable to require the signature of all PNs to accept arbitrary `**kwargs`.

Lucky for us, the `fpn.pipe_node` decorator can be optionally given the desired kwargs we want to positionally bind in the actual function signature.

```
# Bind the first positional argument
@fpn.pipe_node(fpn.PN_INPUT)
def multiply_input_by_2(pn_input):
   return pn_input * 2
```

(continues on next page)

(continued from previous page)

```
# Bind the first positional argument
@fpn.pipe_node(fpn.PREDECESSOR_RETURN)
def add_7(previous_value):
   return previous_value + 7

# Bind the first and second positional arguments
@fpn.pipe_node(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def divide_by_3_add_pn_input(pn_input, previous_value):
   return (previous_value / 3) + pn_input

@fpn.pipe_node() # Bind no arguments
def nothing_is_bound():
   pass

pipeline = (
   nothing_is_bound
   | multiply_input_by_2
   | add_7
   | divide_by_3_add_pn_input
)
assert pipeline[12] == ((((12 * 2) + 7) / 3) + 12)
```

Ah. That's much better. It clears up the function signature, and makes it clear what each PN function needs in order to process properly.

To restate what's happening, arguments given to the decorator will be extracted from the pipeline, and implicitly passed in as the first positional arguments defined in the function signature.

3.6 What About Other Arguments

So far, we have most of the basics. However, there is one essential use case missing: how do I define additional arguments on my function? Let's say instead of a PN called `add_7`, I want to have a PN called `add`, that takes an argument that will be added to the predecessor return value. Here's a pseudo-code example:

```
@fpn.pipe_node(fpn.PREDECESSOR_RETURN)
def add(previous_value, value_to_add):
   return previous_value + value_to_add

pipeline = (... | ... | add(13) | .. )
```

Ideally, there should be a mechanism that allows the user *bind* (or *partial*) custom args & kwargs to give their pipelines all the flexibility needed.

3.7 Welcome To the Factory

Thankfully, such a mechanism exists: it's called `fpn.pipe_node_factory`. This is the other key decorator we need to know for building PNs.

The previous example would work exactly as expected had we replaced the `fpn.pipe_node` decorator with the `fpn.pipe_node_factory` decorator!

```
@fpn.pipe_node(fpn.PN_INPUT)
def init(pn_input):
   return pn_input

@fpn.pipe_node_factory(fpn.PREDECESSOR_RETURN)
def add(previous_value, value_to_add):
   return previous_value + value_to_add

pipeline = (init | add(3) | add(4.2) | add(-2003))
assert pipeline[0] == (0 + 3 + 4.2 + -2003)
```

To reiterate what's happening here, the `fpn.pipe_node_factory` decorates the method in such way it can be thought of as a factory that builds PNs. This is essential, since every element in a pipeline **must** be a PN! The PN factories allow us to used *bound* (or *partialed*) PN with arbitrary args/kwargs.

3.8 A Common Factory Mistake

A common failure when using `fpn.pipe_node_factory` is forgetting to call the decorator before it's put into the pipeline!

Building on the previous example, let's see what happens if we forgot to add an argument to `add`.

```
@fpn.pipe_node(fpn.PN_INPUT)
def init(pn_input):
   return pn_input

@fpn.pipe_node_factory(fpn.PREDECESSOR_RETURN)
def add(previous_value, value_to_add):
   return previous_value + value_to_add

# Uh-oh! One of the `add` pn factories was not given its required argument!
pipeline = (init | add(3) | add(4.2) | add)
```

Let's see the failure message this will raise:

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
...
ValueError: Either you put a factory in a pipeline (i.e. not a pipe node), or your␣
↪factory was given a reserved pipeline kwarg ('pn_input', 'predecessor_pn',
↪'predecessor_return').
```

This failure should make sense now! Every node in a pipline **must** be a PN. Since `add` was not given a factory argument, it was a *PN factory*, **not** a PN.

3.9 PN Input (pni)

Code Alias

pni: pn_input (argument conventionally bound to `fpn.PN_INPUT`)

Up until now, the usage of `pni` (i.e. the argument conventionally bound to `fpn.PN_INPUT`) has been a relatively diverse. This is because `fpn.PN_INPUT` refers to the initial input to the pipeline, and as such, can be any value. For these simple examples, I have been providing integers, but real-world cases typically rely on the `fpn.PipeNodeInput` class.

`fpn.PipeNodeInput` is a subclassable object, which has the ability to:

1. Store results from previous PNs
2. Recall values from previous PNs
3. Share state across PNs.

Let's observe the following example, where we subclass `fpn.PipeNodeInput` in order to share some state accross PNs.

```
class PNI(fpn.PipeNodeInput):
   def __init__(self, state):
      super().__init__()
      self.state = state

pni_12 = PNI(12)

@fpn.pipe_node(fpn.PN_INPUT)
def pipe_node_1(pni):
   return pni.state * 2

@fpn.pipe_node(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def pipe_node_2(pni, previous_value):
   return (pni.state * previous_value) / 33

@fpn.pipe_node(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def pipe_node_3(pni, previous_value):
   return (previous_value ** pni.state) -16

pipeline = (pipe_node_1 | pipe_node_2 | pipe_node_3)
assert pipeline[pni_12] == ((((12 * (12 * 2)) / 33) ** 12) - 16)
```

This is also a good opportunity to highlight how pipeline expressions can be easily reused to provide different results when given different inital inputs. Using the above example, giving a different `pni` will give us a totally different result:

```
pni_99 = PNI(99)
assert pipeline[pni_99] == ((((99 * (99 * 2)) / 33) ** 99) - 16)
assert pipeline[pni_99] != pipeline[pni_12]
```

3.10 Store & Recall

One of the main benefits to using a `fpn.PipeNodeInput` subclass, is the ability to use `fpn.store` and `fpn.recall`. These utility methods will store & recall results from a cache privately stored on the `pni`.

```
@fpn.pipe_node()
def returns_12345():
   return 12345

@fpn.pipe_node(fpn.PREDECESSOR_RETURN)
def double_previous(previous_value):
   return previous_value * 2

@fpn.pipe_node(fpn.PREDECESSOR_RETURN)
def return_previous(previous_value):
   return previous_value

pni = fpn.PipeNodeInput()

pipeline_1 = (
   returns_12345
   | fpn.store("first_result")
   | double_previous
   | fpn.store("second_result")
)
pipeline_1[pni]

pipeline_2 = (fpn.recall("first_result") | return_previous)
assert pipeline_2[pni] == 12345

pipeline_3 = (fpn.recall("second_result") | return_previous)
assert pipeline_3[pni] == (12345 * 2)
```

As you can see, once results have been stored using `fpn.store`, they are retrievable using `fpn.recall` for any other pipeline **that is evaluated with that same pni**!

Additionally, you can see that `fpn.store` and `fpn.recall` simply forward along the previous return values so that they can be seamlessly inserted anywhere into a pipeline.

Note: `fpn.store` and `fpn.recall` only work when the initial input is a valid instance or subclass instance of `fpn.PipeNodeInput`.

3.11 Advanced - Instance/Class/Static Methods

The final section in this tutorial explains the tools needed for turning `classmethods` and `staticmethods` into PNs. To do this, we can take advantage of special classmethod/staticmethod tools built into the **function_pipe** library!

Note: Normal "instance" methods (i.e. functions that expect self (i.e. the instance) passed in as the first argument) work exactly as expected with the `fpn.pipe_node` and `fpn.pipe_node_factory` decorators, as long as the name of the argument is "self".

Building on everything we've seen so far, let's take a look at the class below, which demonstrates usage of `fpn.classmethod_pipe_node`, `fpn.classmethod_pipe_node_factory`, `fpn.staticmethod_pipe_node` and `fpn.staticmethod_pipe_node_factory`.

```
class Operations:
   STATE = 1

   def __init__(self, state):
      self.state = state

   @fpn.pipe_node
   def operation_1(self, **kwargs):
      # This works as expected, since the first argument is "self"
      return self.state + kwargs[fpn.PN_INPUT].state

   @fpn.classmethod_pipe_node
   def operation_2(cls, **kwargs):
      return cls.STATE + kwargs[fpn.PN_INPUT].state

   @fpn.staticmethod_pipe_node
   def operation_3(**kwargs):
      return kwargs[fpn.PN_INPUT].state

   @fpn.pipe_node_factory
   def operation_4(self, user_arg, *, user_kwarg, **kwargs):
      return (self.state + user_arg - user_kwarg) * kwargs[fpn.PN_INPUT].state

   @fpn.classmethod_pipe_node_factory
   def operation_5(cls, user_arg, *, user_kwarg, **kwargs):
      return (cls.STATE + user_arg - user_kwarg) * kwargs[fpn.PN_INPUT].state

   @fpn.staticmethod_pipe_node_factory
   def operation_6(user_arg, *, user_kwarg, **kwargs):
      return (user_arg - user_kwarg) * kwargs[fpn.PN_INPUT].state

   @fpn.pipe_node(fpn.PN_INPUT)
   def operation_7(self, pni):
      return (self.state + pni.state) * 2

   @fpn.classmethod_pipe_node_factory(fpn.PREDECESSOR_RETURN)
   def operation_8(cls, previous_value, user_arg, *, user_kwarg):
      return (cls.STATE + user_arg - user_kwarg) * previous_value
```

(continues on next page)

(continued from previous page)

```
   @fpn.staticmethod_pipe_node(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
   def operation_9(pni, previous_value):
      return (pni.state - previous_value) ** 2

class PNI(fpn.PipeNodeInput):
   def __init__(self, state):
      super().__init__()
      self.state = state

pni = PNI(-99)

op = Operations(2)

pipeline = (
      # The first three are PNs!
      op.operation_1
      | op.operation_2
      | op.operation_3
      # The second three are PN factories!
      | op.operation_4(10, user_kwarg=11)
      | op.operation_5(12, user_kwarg=13)
      | op.operation_6(14, user_kwarg=15)
      # The rest are PNs (except `operation_8`)
      | op.operation_7
      | op.operation_8(16, user_kwarg=17)
      | op.operation_9
)

assert pipeline[pni] == 9801 # Good luck figuring that one out ;)
```

To help explain the decorators a bit more, here is a quick pseudo-code example showing an alternative way to understand them:

```
@fpn.classmethod_pipe_node

# Behaves like you think this would:

@classmethod
@fpn.pipe_node

# ------------------------------------------------------------

@fpn.staticmethod_pipe_node_factory

# Behaves like you think this would:

@staticmethod
@fpn.pipe_node_factory

# etc...
```

3.12 Miscellaneous

3.12.1 __getitem__

For this entire tutorial, PNs and pipeline expressions have been evaluated using `__getitem__`. There is actually another way to do this. As we learned, the first node in a pipeline only receives `fpn.PN_INPUT` as a kwarg. Not only that, but it **must** receive that as a kwarg. The call that kicks off a PN/pipeline evaluation must give a single kwarg:`fpn.PN_INPUT`

Thus, we can actually evaluate a PN/pipeline expression this way:

```
some_pipe_node(**{fpn.PN_INPUT: pni})
```

Obviously, this approach is not very pretty, and it's quite a lot to type for the privilege of evaluation. Thus, the `__getitem__` syntactical sugar was introduced to make it so the user isn't required to unpack a single kwarg whenever they want to evaluate a pipeline.

Note: `__getitem__` has special handling for when the key is `None`. This will evaluate the PN/pipeline expression with a bare instance of `fpn.PipeNodeInput`. If the user desires to evaluate their expression with the literal value `None`, they must kwarg unpack like so: `pn(**{fpn.PN_INPUT: None})`.

3.12.2 Common Mistakes

1. Placing a bare factory in pipeline (see: A Common Factory Mistake).
2. Calling a PN directly (with the exception of unpacking the single kwarg `fpn.PN_INPUT`).
3. Partialing a method wrapped with `fpn.pipe_node` or `fpn.pipe_node_factory`.
4. Using `@classmethod` or `@staticmethod` decorators instead of the special decorators designed for working with classmethods/staticmethods.
5. Decorating a function with `fpn.pipe_node` whose signature expects args/kwargs outside either those bound from the pipeline, or `**kwargs`.

3.12.3 Broadcasting

A feature of `fpn.pipe_node_factory` is how it handles args/kwargs that are themselves PNs. For these types of arguments, it will evaluate them as isolated PNs with `fpn.PN_INPUT` forwarded, and then use the evaluated value in place of that PN. (This is referred to as broadcasting).

Example:

```
@fpn.pipe_node_factory()
def add_divide_exponentiate(*args, divide_by, to_power):
   return (sum(args) / divide_by) ** to_power

@fpn.pipe_node(fpn.PN_INPUT)
def multiply_input_by_2(pni):
   return pni * 2

@fpn.pipe_node(fpn.PN_INPUT)
def add_3_to_pni(pni):
```

(continues on next page)

(continued from previous page)

```
    return pni + 3

@fpn.pipe_node(fpn.PN_INPUT)
def forward_pni(pni):
    return pni

pipeline = add_divide_exponentiate(
    multiply_input_by_2,
    -4,
    forward_pni,
    divide_by=25,
    to_power=add_3_to_pni,
)

assert pipeline[12] == ((12 * 2 - 4 + 12) / 25) ** (12 + 3)
```

As we can see, when factories are given PNs as args/kwargs, they are evaluated with the `fpn.PN_INPUT` given to the original PN/expression being evaluated.

3.12.4 Arithmetic

A helpful feature of PNs, is the ability to perform arithmetic operations on the pipeline during creation. Supported operators are:

- Unary: `-`, `~`, and `abs()`
- Binary: `+`, `-`, `*`, `/`, `**`, `==`, `!=`, `>`, `<`, `<=`, and `>=`

```
@fpn.pipe_node(fpn.PN_INPUT)
def get_pni(pni):
    return pni

@fpn.pipe_node_factory(fpn.PREDECESSOR_RETURN)
def mul(prev, val):
    return prev*val

expr = ((get_pni + abs(-get_pni | mul(-0.9))) | mul(17) - 6 / get_pni) ** 23

assert expr[12] == ((12 + abs(-12 * -0.9)) * 17 - 6 / 12) ** 23
```

3.13 Conclusion

After going through this tutorial, you should now have an understanding of:

- The **creation** and **evaluation** stages of a pipeline
- The `fpn.pipe_node` decorator, and when to use it
- The `fpn.pipe_node_factory` decorator, and when to use it
- How to positionally bind the first argument(s) of a pipeline to `fpn.PN_INPUT` and/or `fpn.PREDECESSOR_RETURN`.

- How to use `fpn.store` and `fpn.recall` to store and recall results from a pipeline.
- How to use `fpn.PipeNodeInput`.
- How to make instance methods, classmethods, and staticmethods into PNs.
- Why `__getitem__` is used to evaluate a pipeline, and what an alternative calling method is
- How to identify and address the most common mistakes when using PNs.
- What broadcasting is and how to use it.
- How to use arithmetic unary/binary operators in a pipeline.

Here is all of the code examples we have seen so far:

```
import function_pipe as fpn  # Import convention!

@fpn.pipe_node
def pipe_node_1(**kwargs):
   print("pipe_node_1 has been evaluated")

@fpn.pipe_node
def pipe_node_2(**kwargs):
   print("pipe_node_2 has been evaluated")

print("Start creation")
pipeline = (pipe_node_1 | pipe_node_2)
print("End creation")

print("Start pipeline evaluation")
pipeline[None]
print("End pipeline evaluation")

# ---------------------------------------------------------------------------

@fpn.pipe_node
def pipe_node_1(**kwargs):
   print(kwargs)
   return 1

@fpn.pipe_node
def pipe_node_2(**kwargs):
   print(kwargs)
   return 2

@fpn.pipe_node
def pipe_node_3(**kwargs):
   print(kwargs)
   return 3

pipeline = (pipe_node_1 | pipe_node_2 | pipe_node_3)
assert pipeline["original_input"] == 3

print(f"repr(pipeline) = '{repr(pipeline)}'")

# ---------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
@fpn.pipe_node
def multiply_input_by_2(**kwargs):
   return kwargs[fpn.PN_INPUT] * 2

@fpn.pipe_node
def add_7(**kwargs):
   return kwargs[fpn.PREDECESSOR_RETURN] + 7

@fpn.pipe_node
def divide_by_3(**kwargs):
   return kwargs[fpn.PREDECESSOR_RETURN] / 3

pipeline_1 = (multiply_input_by_2 | add_7 | divide_by_3)
assert pipeline_1[12] == (((12 * 2) + 7) / 3)

pipeline_2 = (multiply_input_by_2 | divide_by_3 | add_7)
assert pipeline_2[12] == (((12 * 2) / 3) + 7)

# ---------------------------------------------------------------------------

pipeline_3 = (add_7 | multiply_input_by_2 | divide_by_3)

try:
   pipeline_3[12]
except KeyError as e:
   print(e)

# ---------------------------------------------------------------------------

# Bind the first positional argument
@fpn.pipe_node(fpn.PN_INPUT)
def multiply_input_by_2(pn_input):
   return pn_input * 2

# Bind the first positional argument
@fpn.pipe_node(fpn.PREDECESSOR_RETURN)
def add_7(previous_value):
   return previous_value + 7

# Bind the first and second positional arguments
@fpn.pipe_node(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def divide_by_3_add_pn_input(pn_input, previous_value):
   return (previous_value / 3) + pn_input

@fpn.pipe_node() # Bind no arguments
def nothing_is_bound():
   pass

pipeline = (
   nothing_is_bound
   | multiply_input_by_2
```

(continues on next page)

(continued from previous page)

```
    | add_7
    | divide_by_3_add_pn_input
)
assert pipeline[12] == ((((12 * 2) + 7) / 3) + 12)

# ---------------------------------------------------------------------------

@fpn.pipe_node(fpn.PN_INPUT)
def init(pn_input):
    return pn_input

@fpn.pipe_node_factory(fpn.PREDECESSOR_RETURN)
def add(previous_value, value_to_add):
    return previous_value + value_to_add

pipeline = (init | add(3) | add(4.2) | add(-2003))
assert pipeline[0] == (0 + 3 + 4.2 + -2003)

# ---------------------------------------------------------------------------

@fpn.pipe_node(fpn.PN_INPUT)
def init(pn_input):
    return pn_input

@fpn.pipe_node_factory(fpn.PREDECESSOR_RETURN)
def add(previous_value, value_to_add):
    return previous_value + value_to_add

# Uh-oh! One of the `add` pn factories was not given its required argument!
try:
    pipeline = (init | add(3) | add(4.2) | add)
except ValueError as e:
    print(e)

# ---------------------------------------------------------------------------

class PNI(fpn.PipeNodeInput):
    def __init__(self, state):
        super().__init__()
        self.state = state

pni_12 = PNI(12)

@fpn.pipe_node(fpn.PN_INPUT)
def pipe_node_1(pni):
    return pni.state * 2

@fpn.pipe_node(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def pipe_node_2(pni, previous_value):
    return (pni.state * previous_value) / 33

@fpn.pipe_node(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
```

(continues on next page)

(continued from previous page)

```
def pipe_node_3(pni, previous_value):
   return (previous_value ** pni.state) -16

pipeline = (pipe_node_1 | pipe_node_2 | pipe_node_3)
assert pipeline[pni_12] == ((((12 * (12 * 2)) / 33) ** 12) - 16)

# ---------------------------------------------------------------------------

pni_99 = PNI(99)
assert pipeline[pni_99] == ((((99 * (99 * 2)) / 33) ** 99) - 16)
assert pipeline[pni_99] != pipeline[pni_12]

# ---------------------------------------------------------------------------

@fpn.pipe_node()
def returns_12345():
   return 12345

@fpn.pipe_node(fpn.PREDECESSOR_RETURN)
def double_previous(previous_value):
   return previous_value * 2

@fpn.pipe_node(fpn.PREDECESSOR_RETURN)
def return_previous(previous_value):
   return previous_value

pni = fpn.PipeNodeInput()

pipeline_1 = (
   returns_12345
   | fpn.store("first_result")
   | double_previous
   | fpn.store("second_result")
)
pipeline_1[pni]

pipeline_2 = (fpn.recall("first_result") | return_previous)
assert pipeline_2[pni] == 12345

pipeline_3 = (fpn.recall("second_result") | return_previous)
assert pipeline_3[pni] == (12345 * 2)

# ---------------------------------------------------------------------------

class Operations:
   STATE = 1

   def __init__(self, state):
      self.state = state

   @fpn.pipe_node
   def operation_1(self, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        # This works as expected, since the first argument is "self"
        return self.state + kwargs[fpn.PN_INPUT].state

    @fpn.classmethod_pipe_node
    def operation_2(cls, **kwargs):
        return cls.STATE + kwargs[fpn.PN_INPUT].state

    @fpn.staticmethod_pipe_node
    def operation_3(**kwargs):
        return kwargs[fpn.PN_INPUT].state

    @fpn.pipe_node_factory
    def operation_4(self, user_arg, *, user_kwarg, **kwargs):
        return (self.state + user_arg - user_kwarg) * kwargs[fpn.PN_INPUT].state

    @fpn.classmethod_pipe_node_factory
    def operation_5(cls, user_arg, *, user_kwarg, **kwargs):
        return (cls.STATE + user_arg - user_kwarg) * kwargs[fpn.PN_INPUT].state

    @fpn.staticmethod_pipe_node_factory
    def operation_6(user_arg, *, user_kwarg, **kwargs):
        return (user_arg - user_kwarg) * kwargs[fpn.PN_INPUT].state

    @fpn.pipe_node(fpn.PN_INPUT)
    def operation_7(self, pni):
        return (self.state + pni.state) * 2

    @fpn.classmethod_pipe_node_factory(fpn.PREDECESSOR_RETURN)
    def operation_8(cls, previous_value, user_arg, *, user_kwarg):
        return (cls.STATE + user_arg - user_kwarg) * previous_value

    @fpn.staticmethod_pipe_node(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
    def operation_9(pni, previous_value):
        return (pni.state - previous_value) ** 2

class PNI(fpn.PipeNodeInput):
    def __init__(self, state):
        super().__init__()
        self.state = state

pni = PNI(-99)

op = Operations(2)

pipeline = (
        # The first three are PNs!
        op.operation_1
        | op.operation_2
        | op.operation_3
        # The second three are PN factories!
        | op.operation_4(10, user_kwarg=11)
        | op.operation_5(12, user_kwarg=13)
```

(continues on next page)

(continued from previous page)

```
      | op.operation_6(14, user_kwarg=15)
      # The rest are PNs (except `operation_8`)
      | op.operation_7
      | op.operation_8(16, user_kwarg=17)
      | op.operation_9
)

assert pipeline[pni] == 9801 # Good luck figuring that one out ;)

# ---------------------------------------------------------------------------

@fpn.pipe_node_factory()
def add_divide_exponentiate(*args, divide_by, to_power):
   return (sum(args) / divide_by) ** to_power

@fpn.pipe_node(fpn.PN_INPUT)
def multiply_input_by_2(pni):
   return pni * 2

@fpn.pipe_node(fpn.PN_INPUT)
def add_3_to_pni(pni):
   return pni + 3

@fpn.pipe_node(fpn.PN_INPUT)
def forward_pni(pni):
   return pni

pipeline = add_divide_exponentiate(
   multiply_input_by_2,
   -4,
   forward_pni,
   divide_by=25,
   to_power=add_3_to_pni,
)

assert pipeline[12] == ((12 * 2 - 4 + 12) / 25) ** (12 + 3)

# ---------------------------------------------------------------------------

@fpn.pipe_node(fpn.PN_INPUT)
def get_pni(pni):
   return pni

@fpn.pipe_node_factory(fpn.PREDECESSOR_RETURN)
def mul(prev, val):
   return prev*val

expr = ((get_pni + abs(-get_pni | mul(-0.9))) | mul(17) - 6 / get_pni) ** 23

assert expr[12] == ((12 + abs(-12 * -0.9)) * 17 - 6 / 12) ** 23
```

CHAPTER

FOUR

STRING PROCESSING WITH FUNCTIONNODE AND PIPENODE

4.1 Introduction

Simple examples of `FunctionNode` and `PipeNode` can be provided with string processing functions. While not serving any practical purpose, these examples demonstrate core features. Other usage examples will provide more practical demonstrations.

4.2 Importing function-pipe

Throughout these examples function-pipe will be imported as follows.

```
import function_pipe as fpn
```

This assumes the function_pipe.py module has been installed in `site-packages` or is otherwise available via `sys.path`.

4.3 FunctionNodes for Function Composition

FunctionNodes wrap callables. These callables can be lambdas, functions, or instances of callable classes. We can wrap them directly by calling `FunctionNode` or use `FunctionNode` as a decorator.

Using `lambda` callables for brevity, we can start with a number of simple functions that concatenate a string to an input string.

```
a = fpn.FunctionNode(lambda s: s + "a")
b = fpn.FunctionNode(lambda s: s + "b")
c = fpn.FunctionNode(lambda s: s + "c")
d = fpn.FunctionNode(lambda s: s + "d")
e = fpn.FunctionNode(lambda s: s + "e")
```

With or without the `FunctionNode` decorator, we can call and compose these in Python with nested calls, such that the return of the inner function is the argument to the outer function.

```
x = e(d(c(b(a("*")))))
assert x == "*abcde"
```

This approach does not return a new function we can use repeatedly with different inputs. To do so, we can wrap the same nested calls in a `lambda`. The *initial input* is the input provided to the resulting composed function.

```
f = lambda x: e(d(c(b(a(x)))))
assert f("*") == "*abcde"
```

While this works, it can be hard to maintain. By using FunctionNodes, we can make this composition more readable through it's linear >> or << operators.

Both of these operators return a `FunctionNode` that, when called, pipes inputs to outputs (>>: left to right, <<: left to right). As with the `lambda` example above, we can reuse the resulting `FunctionNode` with different inputs.

```
f = a >> b >> c >> d >> e
assert f("*") == "*abcde"
assert f("?") == "?abcde"
```

Depending on your perspective, a linear presentation from left to right may not map well to the nested presentation initially given. The << operator can be used to process from right to left:

```
f = a << b << c << d << e
assert f("*") == "*edcba"
```

And even though it is ill-advised on grounds of poor readability and unnecessary conceptual complexity, you can do bidirectional composition too:

```
f = a >> b >> c << d << e
assert f("*") == "*edabc"
```

The `FunctionNode` overloads standard binary and unary operators to produce new `FunctionNodes` that encapsulate operator operations. Operators can be mixed with composition to create powerful expressions.

```
f = a >> (b * 4) >> (c + "___") >> d >> e
assert f("*") == "*ab*ab*ab*abc___de"
```

We can create multiple FunctionNode expressions and combine them with operators and other compositions. Notice that the *initial input* "*" is made available to both *innermost* expressions, `p` and `q`.

```
p = c >> (b + "_") * 2
q = d >> e * 2
f = (p + q) * 2 + q
assert f("*") == "*cb_*cb_*de*de*cb_*cb_*de*de*de*de"
assert f("+") == "+cb_+cb_+de+de+cb_+cb_+de+de+de+de"
```

In the preceeding examples the functions took only the value of the *predecessor return* as their input. Each function thus has only one argument. Functions with additional arguments are much more useful.

As is common in approaches to function composition, we can partial multi-argument functions so as to compose them in a state where they only require the *predecessor return* as their input.

The `FunctionNode` exposes a `partial` method that simply calls `functools.partial` on the wrapped callable, and returns that new partialed function re-wrapped in a `FunctionNode`.

```
replace = fpn.FunctionNode(lambda s, src, dst: s.replace(src, dst))

p = c >> (b + "_") * 2 >> replace.partial(src="b", dst="B$")
q = d >> e * 2 >> replace.partial(src="d", dst="%D")
f = (p + q) * 2 + q

assert f("*") == "*cB$_*cB$_*%De*%De*cB$_*cB$_*%De*%De*%De*%De"
```

4.4 PipeNodes for Extended Function Composition

At higher level of complexity, `FunctionPipe` can start to become difficult to understand or maintain. The `PipeNode` class (a subclass of `FunctionNode`) and its associated decorators makes *extended function composition* practical, readable, and maintainable. Rather than using the >> or << operators used by `FunctionNode`, `PipeNode` uses only the | operator to express left-to-right composition.

We will build on the tutorial from earlier (LINK NEEDED), and now explore more complex string processing functions using `PipeNode`.

Using the function `a` from before, we will instead create it as a `PipeNode`, using the `pipe_node` decorator.

```
a = fpn.pipe_node(fpn.PREDECESSOR_RETURN)(lambda s: s + "a")
```

Recall that PNs that receive `fpn.PREDECESSOR_RETURN` must have a preceding PN. In our case, we want an initial PN that receives an *initial input* from the user. We will do this by positionally binding `fpn.PN_INPUT` to the first argument.

```
init = fpn.pipe_node(fpn.PN_INPUT)(lambda s: s)
```

Finally, we can generalize string concatenation with a `cat` function that, given an arbitrary string, concatenates it to its predecessor return value. Since this function takes an *expresion-level argument*, we must use the `pipe_node_factory` decorator.

```
cat = fpn.pipe_node_factory(fpn.PREDECESSOR_RETURN)(lambda s, chars: s + chars)
```

Now we can create a pipeline expression that evaluates to a single function `f`. In order to evaluate the pipeline, recall we must the `__getitem__` syntax with some initial input.

```
f = init | a | cat("b") | cat("c")
assert f["*"] == "*abc"
assert f["+"] == "+abc"
```

Each node in a `PipeNode` expression has access to the `fpn.PN_INPUT`. This can be used for many applications. A trivial application below replaces *initial input* characters found in the *predecessor return* with characters provided with the *expression-level argument* `chars`.

```
@fpn.pipe_node_factory(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def replace_init(pni, s, chars):
    return s.replace(pni, chars)

f = init | a | cat("b") | cat("c") * 2 | replace_init("+")
assert f["*"] == "+abc+abc"
```

As already shown, a callable decorated with `pipe_node_factory` can take *expression-level arguments*. With a `PipeNode` expression, these arguments can be `PipeNode` expressions. The following function interleaves *expression-level arguments* with those of the *predecessor return* value.

```
@fpn.pipe_node_factory(fpn.PREDECESSOR_RETURN)
def interleave(s, chars):
    post = []
    for i, c in enumerate(s):
        post.append(c)
        post.append(chars[i % len(chars)])
    return "".join(post)
```

(continues on next page)

(continued from previous page)

```
h = init | cat("@@") | cat("__") * 2

f = init | a | cat("b") | cat("c") * 3 | replace_init("+") | interleave(h)

assert f["*"] == "+*a@b@c_+_a*b@c@+_a_b*c@"
```

We can break `PipeNode` expressions into pieces by storing and recalling results. This requires that the *initial input* is a `PipeNodeInput` or a subclass. The following `PNI` class exposes the `__init__` based `chars` argument as an instance attribute. Alternative designs for `PipeNodeInput` subclasses can provide a range of input data preparation. Since our *initial input* has changed, we need a new *innermost* node. The `input_init` node defined below simply returns the `chars` attribute from the `PNI` instance passed as key-word argument `fpn.PN_INPUT`.

The function-pipe module provides `store` and `recall` nodes. The `store` node stores a predecessor value. The `recall` node returns a stored value as an output later in the expression. A `recall` node, for example, can be used as an argument to `pipe_node_factory` functions. The `call` `PipeNode`, also provided in the function-pipe module, will call any number of passed `PipeNode` expressions in sequence.

```
class PNI(fpn.PipeNodeInput):
    def __init__(self, chars):
        super().__init__()
        self.chars = chars

@fpn.pipe_node(fpn.PN_INPUT)
def input_init(pni):
    return pni.chars

p = input_init | cat("www") | fpn.store("p")
q = input_init | cat("@@") | cat("__") * 2 | fpn.store("q")
r = (
    input_init
    | a
    | cat(fpn.recall("p"))
    | cat("c") * 3
    | interleave(fpn.recall("q"))
)

f = fpn.call(p, q, r)
pni = PNI("x")

assert f[pni] == "xxa@x@w_w_wxc@x@a_x_wxw@w@c_x_axx@w@w_w_cx"
```

While these string processors do not do anything useful, they demonstrate common approaches in working with `FunctionNode` and `PipeNode`.

4.5 Conclusion

After going through this tutorial, you should now have an understanding of:

- How to use `fpn.FunctionNode` for function composition
- The directionality of `fpn.FunctionPipe` (i.e. >> and <<)
- How to partial expression-level arguments into `fpn.FunctionPipe`
- The `fpn.pipe_node` decorator, and when to use it
- The `fpn.pipe_node_factory` decorator, and when to use it
- How to use `fpn.PipeNode` for function composition

Here is all of the code examples we have seen so far:

```
import function_pipe as fpn

a = fpn.FunctionNode(lambda s: s + "a")
b = fpn.FunctionNode(lambda s: s + "b")
c = fpn.FunctionNode(lambda s: s + "c")
d = fpn.FunctionNode(lambda s: s + "d")
e = fpn.FunctionNode(lambda s: s + "e")

x = e(d(c(b(a("*")))))
assert x == "*abcde"

# ----------------------------------------------------------------------

f = lambda x: e(d(c(b(a(x)))))
assert f("*") == "*abcde"

# ----------------------------------------------------------------------

f = a >> b >> c >> d >> e
assert f("*") == "*abcde"
assert f("?") == "?abcde"

# ----------------------------------------------------------------------

f = a << b << c << d << e
assert f("*") == "*edcba"

# ----------------------------------------------------------------------

f = a >> b >> c << d << e
assert f("*") == "*edabc"

# ----------------------------------------------------------------------

f = a >> (b * 4) >> (c + "___") >> d >> e
assert f("*") == "*ab*ab*ab*abc___de"

# ----------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
p = c >> (b + "_") * 2
q = d >> e * 2
f = (p + q) * 2 + q
assert f("*") == "*cb_*cb_*de*de*cb_*cb_*de*de*de*de"
assert f("+") == "+cb_+cb_+de+de+cb_+cb_+de+de+de+de"

# ---------------------------------------------------------------------------

replace = fpn.FunctionNode(lambda s, src, dst: s.replace(src, dst))

p = c >> (b + "_") * 2 >> replace.partial(src="b", dst="B$")
q = d >> e * 2 >> replace.partial(src="d", dst="%D")
f = (p + q) * 2 + q

assert f("*") == "*cB$_*cB$_*%De*%De*cB$_*cB$_*%De*%De*%De*%De"

# ---------------------------------------------------------------------------

a = fpn.pipe_node(fpn.PREDECESSOR_RETURN)(lambda s: s + "a")

init = fpn.pipe_node(fpn.PN_INPUT)(lambda s: s)

cat = fpn.pipe_node_factory(fpn.PREDECESSOR_RETURN)(lambda s, chars: s + chars)

f = init | a | cat("b") | cat("c")
assert f["*"] == "*abc"
assert f["+"] == "+abc"

# ---------------------------------------------------------------------------

@fpn.pipe_node_factory(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def replace_init(pni, s, chars):
    return s.replace(pni, chars)

f = init | a | cat("b") | cat("c") * 2 | replace_init("+")
assert f["*"] == "+abc+abc"

# ---------------------------------------------------------------------------

@fpn.pipe_node_factory(fpn.PREDECESSOR_RETURN)
def interleave(s, chars):
    post = []
    for i, c in enumerate(s):
        post.append(c)
        post.append(chars[i % len(chars)])
    return "".join(post)

h = init | cat("@@") | cat("__") * 2

f = init | a | cat("b") | cat("c") * 3 | replace_init("+") | interleave(h)
```

(continues on next page)

(continued from previous page)

```
assert f["*"] == "+*a@b@c_+_a*b@c@+_a_b*c@"

# ----------------------------------------------------------------------

class PNI(fpn.PipeNodeInput):
    def __init__(self, chars):
        super().__init__()
        self.chars = chars

@fpn.pipe_node(fpn.PN_INPUT)
def input_init(pni):
    return pni.chars

p = input_init | cat("www") | fpn.store("p")
q = input_init | cat("@@") | cat("__") * 2 | fpn.store("q")
r = (
    input_init
    | a
    | cat(fpn.recall("p"))
    | cat("c") * 3
    | interleave(fpn.recall("q"))
)

f = fpn.call(p, q, r)
pni = PNI("x")

assert f[pni] == "xxa@x@w_w_wxc@x@a_x_wxw@w@c_x_axx@w@w_w_cx"
```

CHAPTER

FIVE

DATAFRAME PROCESSING WITH FUNCTIONNODE AND PIPENODE

5.1 Introduction

The `FunctionNode` and `PipeNode` were built in large part to handle data processing pipelines with Pandas `Series` and `DataFrame`. The following examples do simple things with data, but provide a framework that can be expanded to meet a wide range of needs.

5.2 Tutorial Data Source

Following an example in Wes McKinney's *Python for Data Analysis, 2nd Edition* (2017), these examples will use U.S. child birth name records from the Social Security Administration. Presently, this data is found at the following URL. We will write Python code to automatically download this data.

https://www.ssa.gov/oact/babynames/names.zip

5.3 DataFrame Processing with FunctionNode

`FunctionNode` wrapped functions can be used to link functions in linear compositions. What is passed to the nodes can change, as long as a node is prepared to receive the value of its predecessor. As before, *core callables* are called only after the complete composition expression is evaluated to a single function and called with the *initial input*.

We will use the follow imports throughout these examples. The `requests` and `pandas` third-party packages can be installed using `pip`.

```
import collections
import os
import webbrowser
import zipfile

import requests
import pandas as pd
import function_pipe as fpn
```

We will introduce the `FunctionNode` decorated functions one at a time. We start with a function that, given a destination file path, will download the dataset (if it does not already exist), read the zip, and load the data into an `OrderedDictionary` of `DataFrame` keyed by year. Each `DataFrame` has a column for "name", "gender", and "count". We will for now store the URL as a module-level constant.

```
URL_NAMES = "https://www.ssa.gov/oact/babynames/names.zip"
FP_ZIP = "unzipped_names.txt"

@fpn.FunctionNode
def load_data_dict(fp):

    if not os.path.exists(fp):
        r = requests.get(URL_NAMES)

        with open(fp, "wb") as f:
            f.write(r.content)

    data_dict = collections.OrderedDict()
    with zipfile.ZipFile(fp) as zf:
        for zip_info in sorted(zf.infolist(), key=lambda zip_info: zip_info.filename):
            filename = zip_info.filename

            if filename.startswith("yob"):
                year = int(filename[3:7])
                df = pd.read_csv(
                    zf.open(zip_info),
                    header=None,
                    names=("name", "gender", "count"),
                )
                data_dict[year] = df

    return data_dict
```

Next, we have a function that, given that same dictionary, produces a single `DataFrame` that lists, for each year, the total number of males and females recorded with columns for "M" and "F". Notice that the approach used below strictly requires the usage of an `OrderedDictionary`.

```
@fpn.FunctionNode
def gender_count_per_year(data_dict):
    records = []
    for year, df in data_dict.items():
        male = df[df["gender"] == "M"]["count"].sum()
        female = df[df["gender"] == "F"]["count"].sum()
        records.append((male, female))

    return pd.DataFrame.from_records(
        records,
        index=data_dict.keys(), # ordered
        columns=("M", "F"),
    )
```

Given row data that represent parts of whole, a utility function can be used to convert the previously created `DataFrame` into percent floats.

```
@fpn.FunctionNode
def percent(df):
    result = pd.DataFrame(index=df.index)
    total = df.sum(axis=1)
```

(continues on next page)

(continued from previous page)

```
    for column in df.columns:
        result[column] = df[column] / total
    return result
```

A utility function can be used to select a contiguous year range from a `DataFrame` indexed by integer year values. We expect the `start` and `end` parameters to provided through partialing, and the `DataFrame` to be provided from the *predecessor return* value:

```
@fpn.FunctionNode
def year_range(df, start, end):
    return df.loc[start:end]
```

We can plot any `DataFrame` using Pandas' interface to `matplotlib` (which will need to be installed and configured separately). The function takes an optional argument for destination file path and returns the same path after writing an image file.

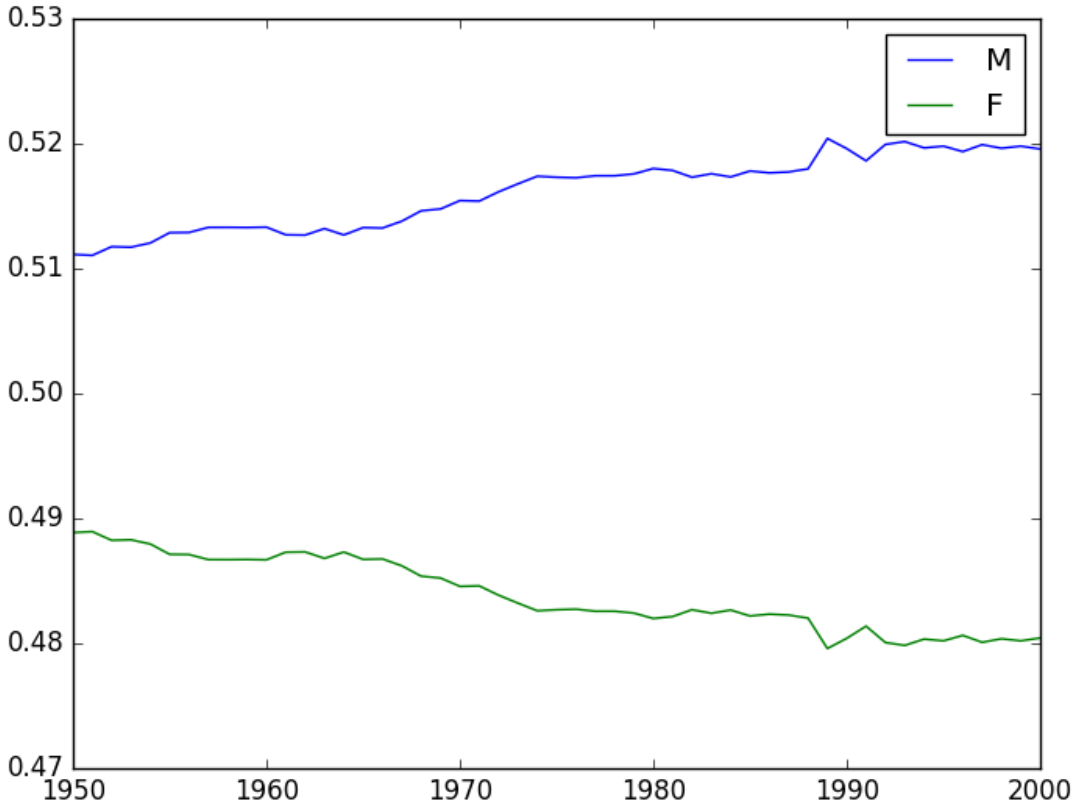
```
@fpn.FunctionNode
def plot(df, fp="/tmp/plot.png"):
    ax = df.plot()
    ax.get_figure().savefig(fp)
    return fp
```

Finally, to open the resulting plot for viewing, we will use Python's `webbrowser` module.

```
@fpn.FunctionNode
def open_plot(fp):
    webbrowser.open(fp)
```

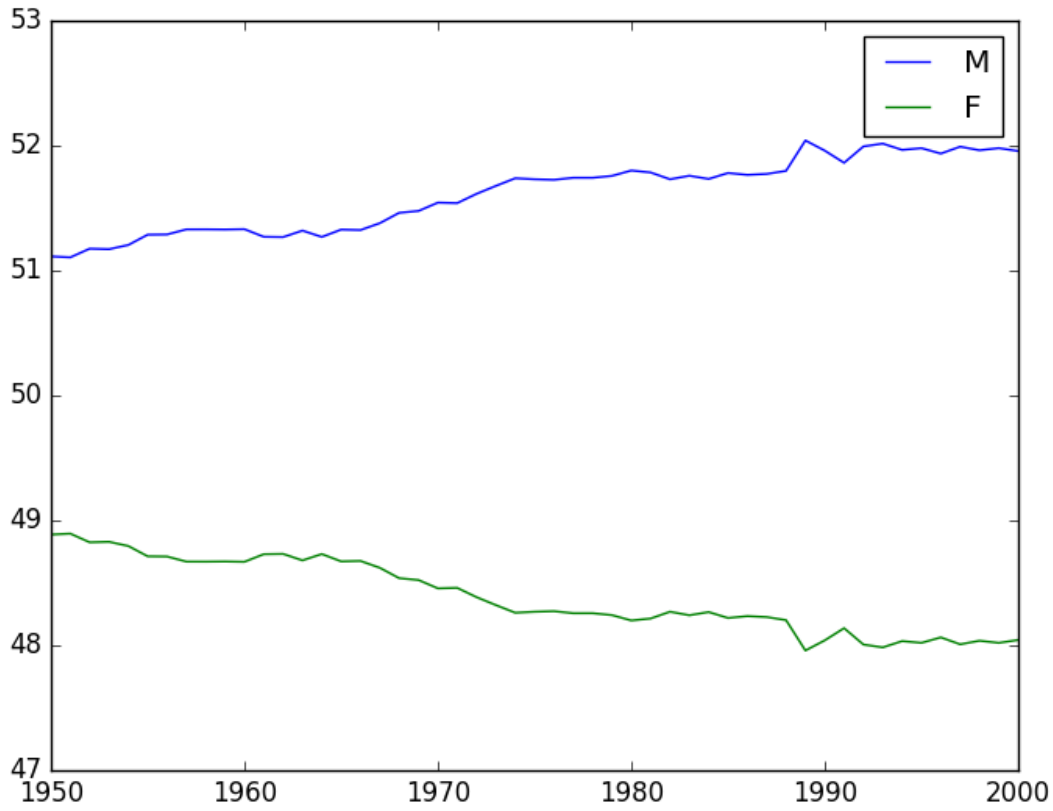
With all functions decorated as `FunctionNode`, we can create a composition expression. The partialed `start` and `end` arguments permit selecting different year ranges. Notice that the data passed between nodes changes, from an `OrderedDict` of `DataFrame`, to a `DataFrame`, to a file path string. To call the composition expression `f`, we simply pass the necessary argument of the *innermost* `load_data_dict` function.

```
f = (
    load_data_dict
    >> gender_count_per_year
    >> year_range.partial(start=1950, end=2000)
    >> percent
    >> plot
    >> open_plot
)

f(FP_ZIP)
```



If, for the sake of display, we want to convert the floating-point percents to integers before ploting, we do not need to modify the `FunctionNode` implementation. As `FunctionNode` support operators, we can simply scale the output of the `percent` `FunctionNode` by 100.

```
f = (
    load_data_dict
    >> gender_count_per_year
    >> year_range.partial(start=1950, end=2000)
    >> (percent * 100)
    >> plot
    >> open_plot
)

f(FP_ZIP)
```



While this approach is illustrative, it is limited. Using simple linear composition, as above, it is not possible with the same set of functions to produce multiple plots with the same data, or both write plots and output `DataFrame` data in Excel. This and more is possible with `PipeNode`.

5.4 DataFrame Processing with PipeNode

Building on the tutorial from earlier (LINK NEEDED), we will now explore processing dataframes using `PipeNode`.

While not required to use pipelines, is is useful to create a `PipeNodeInput` subclass that will share state across the pipeline.

The following implementation of a `PipeNodeInput` subclass stores the URL as the class attribute `URL_NAMES`, and stores the `output_dir` argument as an instance attribute. The `load_data_dict` function is essentially the same as before, though here it is a `classmethod` that reads `URL_NAMES` from the class. The resulting `data_dict` instance attribute is stored in the `PipeNodeInput`, making it available to every node.

```
class PNI(fpn.PipeNodeInput):

    URL_NAMES = "https://www.ssa.gov/oact/babynames/names.zip"

    @classmethod
    def load_data_dict(cls, fp):
```

(continues on next page)

(continued from previous page)

```
        if not os.path.exists(fp):
            r = requests.get(cls.URL_NAMES)
            with open(fp, "wb") as f:
                f.write(r.content)

        data_dict = collections.OrderedDict()
        with zipfile.ZipFile(fp) as zf:
            for zip_info in sorted(zf.infolist(), key=lambda zip_info: zip_info.
↪filename):
                filename = zip_info.filename

                if filename.startswith("yob"):
                    year = int(filename[3:7])
                    df = pd.read_csv(
                            zf.open(zip_info),
                            header=None,
                            names=("name", "gender", "count"))
                    data_dict[year] = df

        return data_dict

    def __init__(self, output_dir):
        super().__init__()
        self.output_dir = output_dir
        fp_zip = os.path.join(output_dir, "names.zip")
        self.data_dict = self.load_data_dict(fp_zip)
```

We can generalize the `gender_count_per_year` function from above to count names per gender per year. Names often have variants, so we can match names with a passed-in function `name_match`. As this node takes an *expression-level argument*, we decorate it with `pipe_node_factory`. Setting this function to `lambda n: True` results in exactly the same functionality as the `gender_count_per_year` function. Recall how we can access `data_dict` from the positionally bound `pni` argument.

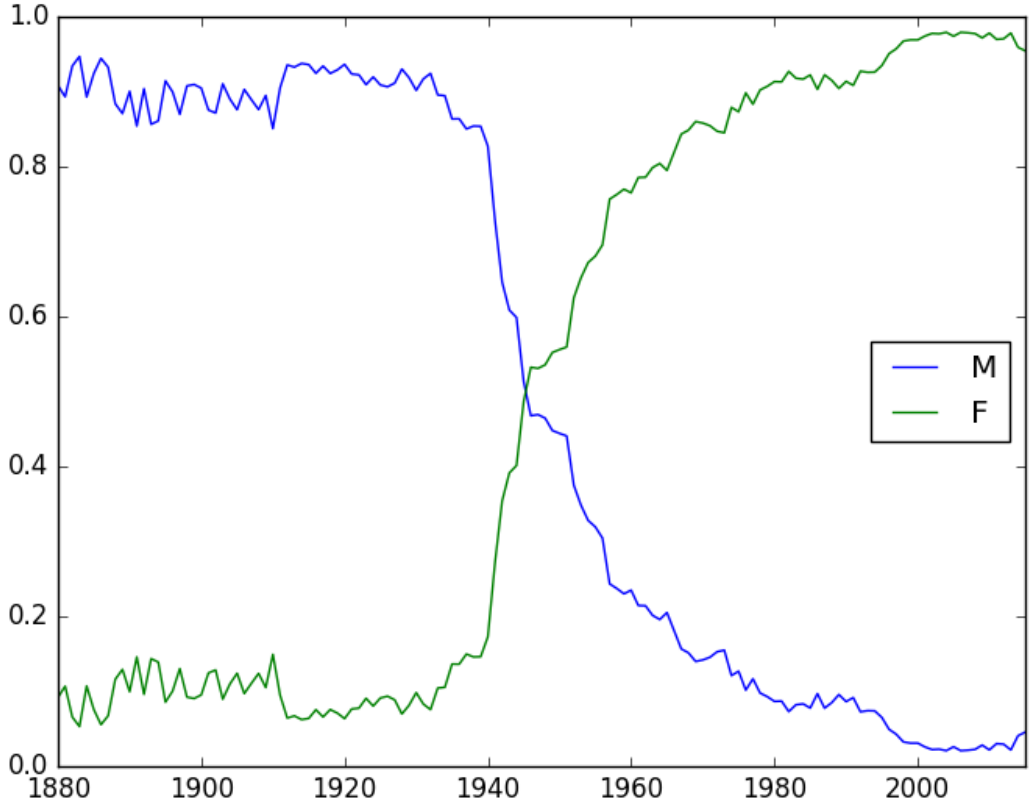
```
@fpn.pipe_node_factory(fpn.PN_INPUT)
def name_count_per_year(pni, name_match):
    records = []

    for year, df in pni.data_dict.items():
        counts = collections.OrderedDict()
        name_selection = df["name"].apply(name_match)

        for gender in ("M", "F"):
            gender_selection = (df["gender"] == gender) & name_selection
            counts[gender] = df[gender_selection]["count"].sum()

        records.append(tuple(counts.values()))

    return pd.DataFrame.from_records(
        records,
        index=pni.data_dict.keys(), # ordered
        columns=("M", "F"),
    )
```

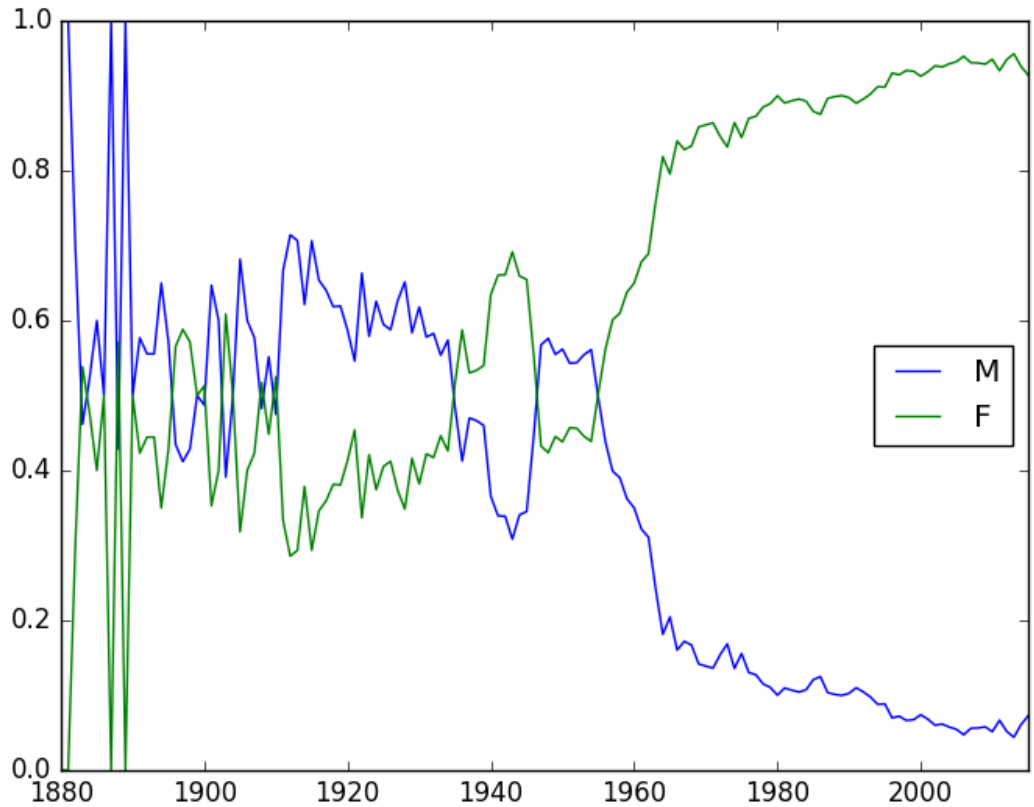
A number of functions used above as `FunctionNode` can be recast as `PipeNode` by simpy binding `fpn.PREDECESSOR_RETURN` as the first positional argument. Recall that PNs that need *expression-level arguments* are decorated with `pipe_node_factory`. The `plot` node now takes a `file_name` argument, to be combined with the output directory set in the `PipeNodeInput` instance.

```
@fpn.pipe_node(fpn.PREDECESSOR_RETURN)
def percent(df):
    result = pd.DataFrame(index=df.index)
    total = df.sum(axis=1)

    for column in df.columns:
        result[column] = df[column] / total

    return result

@fpn.pipe_node_factory(fpn.PREDECESSOR_RETURN)
def year_range(df, start, end):
    return df.loc[start:end]

@fpn.pipe_node_factory(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def plot(pni, df, file_name): # now we can pass a file name
    fp = os.path.join(pni.output_dir, file_name)
    ax = df.plot()
    ax.get_figure().savefig(fp)
    return fp

@fpn.pipe_node(fpn.PREDECESSOR_RETURN)
def open_plot(fp):
    webbrowser.open(fp)
```

With these nodes defined, we can create many differnt processing pipelines. For example, to plot two graphs, one each for the distribution of names that start with "lesl" and "dana", we can create the following expression. Notice that, for maximum efficiency, `load_data_dict` is called only once in the `PipeNodeInput`. Further, now that `plot` takes a file name argument, we can uniquely name our plots.

```
f = (
    name_count_per_year(lambda n: n.lower().startswith("lesl"))
    | percent
    | plot("lesl.png")
    | open_plot
    | name_count_per_year(lambda n: n.lower().startswith("dana"))
    | percent
    | plot("dana.png")
    | open_plot
)

f[PNI("/tmp")]
```


1.0
0.8
0.6
0.4
0.2
0.0
1880
1900
1920
1940
1960
1980
2000
M
F



To support graphing the gender distribution for multiple names simultaneously, we can create a specialized node to merge `PipeNode` expressions passed as key-word arguments. We will then merge all those `DataFrame` key-value pairs.

```
@fpn.pipe_node_factory(fpn.PN_INPUT)
def merge_gender_data(pni, **kwargs):
    df = pd.DataFrame(index=pni.data_dict.keys())
    for k, v in kwargs.items():
        for gender in ("M", "F"):
            df[k + "_" + gender] = v[gender]
    return df
```

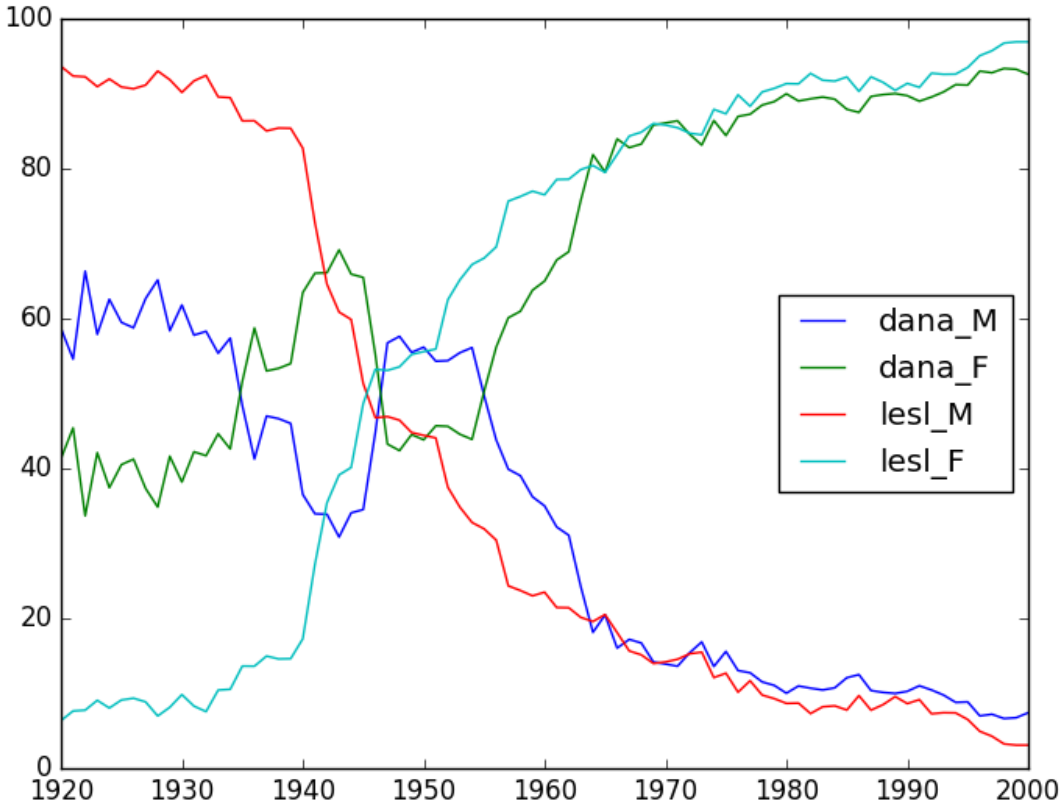
Now we can create two expressions for each name we are investigating. These are then passed to `merge_gender_data` as key-word arguments. In all cases the raw data `DataFrame` is now retained with the `store` `PipeNode`. After plotting and viewing, we can retrieve and iterate over stored keys and `DataFrame` by accessing the `store_items` property of `PipeNodeInput`. In this example, we load each `DataFrame` into a sheet of an Excel workbook.

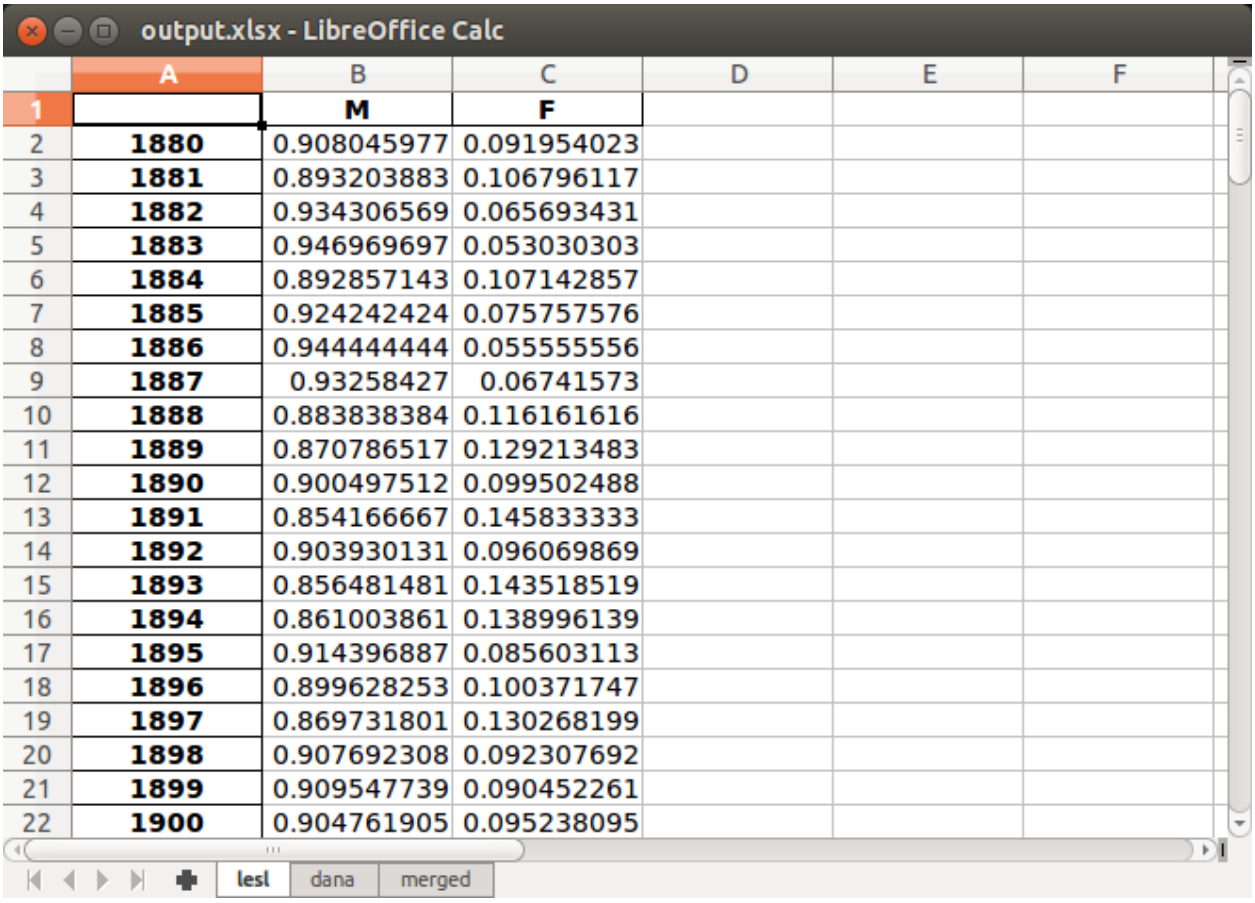
```
lesl_pipeline = (
    name_count_per_year(lambda n: n.lower().startswith("lesl"))
    | percent
    | fpn.store("lesl")
)

dana_pipeline = (
```

(continues on next page)

(continued from previous page)

```
    name_count_per_year(lambda n: n.lower().startswith("dana"))
    | percent
    | fpn.store("dana")
)

f = (
    merge_gender_data(lesl=lesl_pipeline, dana=dana_pipeline)
    | year_range(1920, 2000)
    | fpn.store("merged") * 100
    | plot("gender.png")
    | open_plot
)

pni = PNI("/tmp")
f[pni]

xlsx = pd.ExcelWriter(os.path.join(pni.output_dir, "output.xlsx"))
for k, df in pni.store_items:
    df.to_excel(xlsx, k)
xlsx.save()
```





	A	B	C	D	E	F
1		M	F			
2	1880	0.908045977	0.091954023			
3	1881	0.893203883	0.106796117			
4	1882	0.934306569	0.065693431			
5	1883	0.946969697	0.053030303			
6	1884	0.892857143	0.107142857			
7	1885	0.924242424	0.075757576			
8	1886	0.944444444	0.055555556			
9	1887	0.93258427	0.06741573			
10	1888	0.883838384	0.116161616			
11	1889	0.870786517	0.129213483			
12	1890	0.900497512	0.099502488			
13	1891	0.854166667	0.145833333			
14	1892	0.903930131	0.096069869			
15	1893	0.856481481	0.143518519			
16	1894	0.861003861	0.138996139			
17	1895	0.914396887	0.085603113			
18	1896	0.899628253	0.100371747			
19	1897	0.869731801	0.130268199			
20	1898	0.907692308	0.092307692			
21	1899	0.909547739	0.090452261			
22	1900	0.904761905	0.095238095			

These examples demonstrate organizing data processing routines with `PipeNode` expressions. Using `PipeNodeInput` sublcasses, data acesss routines can be centralized and made as efficient as possible. Further, `PipeNodeInput` sublcasses can provide common parameters, such as output directories, to all nodes. Finally, the results of sub-expressions can be stored and recalled within `PipeNode` expressions, or extracted after `PipeNode` execution for writing to disk.

5.5 Conclusion

After going through this tutorial, you should now have an understanding of:

- How to use `fpn.FunctionNode` to do DataFrame processing
- How to use `fpn.PipeNode` to do DataFrame processing

Here is all of the code examples we have seen so far:

```
import collections
import os
import webbrowser
import zipfile

import requests
import pandas as pd
import function_pipe as fpn
```

(continues on next page)

(continued from previous page)

```
URL_NAMES = "https://www.ssa.gov/oact/babynames/names.zip"
FP_ZIP = "unzipped_names.txt"

@fpn.FunctionNode
def load_data_dict(fp):

    if not os.path.exists(fp):
        r = requests.get(URL_NAMES)

        with open(fp, "wb") as f:
            f.write(r.content)

    data_dict = collections.OrderedDict()
    with zipfile.ZipFile(fp) as zf:
        for zip_info in sorted(zf.infolist(), key=lambda zip_info: zip_info.filename):
            filename = zip_info.filename

            if filename.startswith("yob"):
                year = int(filename[3:7])
                df = pd.read_csv(
                    zf.open(zip_info),
                    header=None,
                    names=("name", "gender", "count"),
                )
                data_dict[year] = df

    return data_dict

@fpn.FunctionNode
def gender_count_per_year(data_dict):
    records = []
    for year, df in data_dict.items():
        male = df[df["gender"] == "M"]["count"].sum()
        female = df[df["gender"] == "F"]["count"].sum()
        records.append((male, female))

    return pd.DataFrame.from_records(
        records,
        index=data_dict.keys(), # ordered
        columns=("M", "F"),
    )

@fpn.FunctionNode
def percent(df):
    result = pd.DataFrame(index=df.index)
    total = df.sum(axis=1)
    for column in df.columns:
        result[column] = df[column] / total
    return result

@fpn.FunctionNode
def year_range(df, start, end):
```

(continues on next page)

(continued from previous page)

```
    return df.loc[start:end]

@fpn.FunctionNode
def plot(df, fp="/tmp/plot.png"):
    ax = df.plot()
    ax.get_figure().savefig(fp)
    return fp

@fpn.FunctionNode
def open_plot(fp):
    webbrowser.open(fp)

# Example 1:

f = (
    load_data_dict
    >> gender_count_per_year
    >> year_range.partial(start=1950, end=2000)
    >> percent
    >> plot
    >> open_plot
)

f(FP_ZIP)

# Example 2:

f = (
    load_data_dict
    >> gender_count_per_year
    >> year_range.partial(start=1950, end=2000)
    >> (percent * 100)
    >> plot
    >> open_plot
)

f(FP_ZIP)

# Example 3:

class PNI(fpn.PipeNodeInput):

    URL_NAMES = "https://www.ssa.gov/oact/babynames/names.zip"

    @classmethod
    def load_data_dict(cls, fp):

        if not os.path.exists(fp):
            r = requests.get(cls.URL_NAMES)
            with open(fp, "wb") as f:
                f.write(r.content)
```

(continues on next page)

(continued from previous page)

```
        data_dict = collections.OrderedDict()
        with zipfile.ZipFile(fp) as zf:
            for zip_info in sorted(zf.infolist(), key=lambda zip_info: zip_info.
↪filename):
                filename = zip_info.filename

                if filename.startswith("yob"):
                    year = int(filename[3:7])
                    df = pd.read_csv(
                            zf.open(zip_info),
                            header=None,
                            names=("name", "gender", "count"))
                    data_dict[year] = df

        return data_dict

    def __init__(self, output_dir):
        super().__init__()
        self.output_dir = output_dir
        fp_zip = os.path.join(output_dir, "names.zip")
        self.data_dict = self.load_data_dict(fp_zip)

@fpn.pipe_node_factory(fpn.PN_INPUT)
def name_count_per_year(pni, name_match):
    records = []

    for year, df in pni.data_dict.items():
        counts = collections.OrderedDict()
        name_selection = df["name"].apply(name_match)

        for gender in ("M", "F"):
            gender_selection = (df["gender"] == gender) & name_selection
            counts[gender] = df[gender_selection]["count"].sum()

        records.append(tuple(counts.values()))

    return pd.DataFrame.from_records(
        records,
        index=pni.data_dict.keys(), # ordered
        columns=("M", "F"),
    )

@fpn.pipe_node(fpn.PREDECESSOR_RETURN)
def percent(df):
    result = pd.DataFrame(index=df.index)
    total = df.sum(axis=1)

    for column in df.columns:
        result[column] = df[column] / total

    return result
```

(continues on next page)

(continued from previous page)

```
@fpn.pipe_node_factory(fpn.PREDECESSOR_RETURN)
def year_range(df, start, end):
    return df.loc[start:end]

@fpn.pipe_node_factory(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def plot(pni, df, file_name): # now we can pass a file name
    fp = os.path.join(pni.output_dir, file_name)
    ax = df.plot()
    ax.get_figure().savefig(fp)
    return fp

@fpn.pipe_node(fpn.PREDECESSOR_RETURN)
def open_plot(fp):
    webbrowser.open(fp)

f = (
    name_count_per_year(lambda n: n.lower().startswith("lesl"))
    | percent
    | plot("lesl.png")
    | open_plot
    | name_count_per_year(lambda n: n.lower().startswith("dana"))
    | percent
    | plot("dana.png")
    | open_plot
)

f[PNI("/tmp")]

# Example 4:

@fpn.pipe_node_factory(fpn.PN_INPUT)
def merge_gender_data(pni, **kwargs):
    df = pd.DataFrame(index=pni.data_dict.keys())
    for k, v in kwargs.items():
        for gender in ("M", "F"):
            df[k + "_" + gender] = v[gender]
    return df

lesl_pipeline = (
    name_count_per_year(lambda n: n.lower().startswith("lesl"))
    | percent
    | fpn.store("lesl")
)

dana_pipeline = (
    name_count_per_year(lambda n: n.lower().startswith("dana"))
    | percent
    | fpn.store("dana")
)

f = (
    merge_gender_data(lesl=lesl_pipeline, dana=dana_pipeline)
```

(continues on next page)

(continued from previous page)

```
    | year_range(1920, 2000)
    | fpn.store("merged") * 100
    | plot("gender.png")
    | open_plot
)

pni = PNI("/tmp")
f[pni]

xlsx = pd.ExcelWriter(os.path.join(pni.output_dir, "output.xlsx"))
for k, df in pni.store_items:
    df.to_excel(xlsx, k)
xlsx.save()
```

CHAPTER

SIX

NUMPY ARRAY PROCESSING WITH PIPENODE

6.1 Introduction

This example will present a complete command-line program to print an equal-space, bitmap / pixel-font display of the current Python version (or, with extension, something else more useful). The display will be configurable with (1) a scaling factor and (2) a variable character to be used per pixel. For example:

```
% python3 pyv.py --scale=1 --pixel=*

****  *   * *****       *****        **   *****
*   * *   *     *       *   *         *       *
***** *****   ***       *****         *   *****
*       *       *       *   *         *   *
*       *   *****   *   *****   *     *   *****



% python3 pyv.py --scale=2 --pixel=.

........    ..      ..  ..........              ..........                ....      .....
↪.....
........    ..      ..  ..........              ..........                ....      .....
↪.....
..      ..  ..      ..          ..              ..      ..                  ..          ␣
↪   ..
..      ..  ..      ..          ..              ..      ..                  ..          ␣
↪   ..
..........  ..........      ......              ..........                  ..      .....
↪.....
..........  ..........      ......              ..........                  ..      .....
↪.....
..              ..              ..              ..      ..                  ..      ..
..              ..              ..              ..      ..                  ..      ..
..              ..      ..........      ..      ..........      ..          ..      .....
↪.....
..              ..      ..........      ..      ..........      ..          ..      .....
↪.....
```

6.2 Tutorial

Rather than explicitly defining each character as a fixed bit map, we can use simple `PipeNode` functions to define characters as pipeline operations on Boolean NumPy arrays. Operations include creating an empty frame, drawing horizontal or vertical lines, shifting those lines, selectively inverting specific pixels, and taking the union or intersection of any number of frames. Since we want to model linear pipelining of frames through transformational nodes, but also need to expose a `scale` parameter to numerous nodes, we will use `PipeNode` functions and a `PipeNodeInput` instance rather than simple function composition.

We will use the follow imports throughout these examples. The `numpy` third-party package can be installed with `pip`.

```
import argparse
import functools
import sys

import numpy as np
import function_pipe as fpn
```

In order to minimize the number of `function_pipe` stdout logs, we will partial in a forwarding lambda that does not print.

```
fpn.pipe_node = functools.partial(
    fpn.pipe_node,
    core_decorator=lambda f: f,
)
fpn.pipe_node_factory = functools.partial(
    fpn.pipe_node_factory,
    core_decorator=lambda f: f,
)
```

A derived `PipeNodeInput` class can specify fixed (as class attributes) or configurable (as arguments passed at initialization and set to instance attributes) parameters, available to all `PipeNode` functions when called. For this example, we set a fixed frame shape of 5 by 5 pixels as `SHAPE`, and expose `scale` and `pixel` as instance attributes.

```
class PixelFontInput(fpn.PipeNodeInput):

    SHAPE = (5,5)

    def __init__(self, pixel="*", scale=1):
        super().__init__()
        self.scale = scale
        self.pixel = pixel
```

Next, we define `pipe_node` decorated functions (that that take no *expression-level arguments*) for creating an empty matrix, a vertical line, and a horizontal line. The `frame` function serves in the *innermost* position to provide an empty two-dimensional NumPy array filled with False. In the *innermost* position it only has access to the `fpn.PN_INPUT` key-word argument. From the `fpn.PN_INPUT` it can read the `SHAPE` and `scale` attributes to correctly construct the frame. The `v_line` and `h_line` functions expect a frame passed via `fpn.PREDECESSOR_RETURN`, and use the `scale` attribute from `fpn.PN_INPUT` to write correctly sized Boolean True values in a vertical or horizontal line through the origin (index 0, 0, or the upper left corner) on that frame.

```
@fpn.pipe_node(fpn.PN_INPUT)
def frame(pixel_font_input):
    shape = tuple(v * pixel_font_input.scale for v in pixel_font_input.SHAPE)
```

(continues on next page)

(continued from previous page)

```
    return np.zeros(shape=shape, dtype=bool)

@fpn.pipe_node(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def v_line(pixel_font_input, matrix):
    matrix = matrix.copy()
    matrix[:, slice(0, pixel_font_input.scale)] = True
    return matrix

@fpn.pipe_node(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def h_line(pixel_font_input, matrix):
    matrix = matrix.copy()
    matrix[slice(0, pixel_font_input.scale), :] = True
    return matrix
```

Next, we can create some transformation functions that, given a frame via `fpn.PREDECESSOR_PN`, transform and return a new frame. The `pipe_node_factory` decorated functions `v_shift` and `h_shift` use the NumPy roll function to shift the two-dimensional array vertically or horizontally by the `steps` argument, passed via *expression-level arguments*. The `steps` passed are interpreted at the unit level, and are thus multipled by `scale` via `fpn.PN_INPUT`. As a convenience to users (and catching an error made developing these tools), we check and raise an Exception if we try to do a meaningless shift, such as vertically shifting a vertical line, or horizontall shifting a horizontal line. The `PipeNode.unwrap` attribute exposes the *core callable* wrapped by the `PipeNode`, permitting direct comparison regardless of `PipeNode` state.

```
@fpn.pipe_node_factory(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN, fpn.PREDECESSOR_PN)
def v_shift(pixel_font_input, matrix, predecessor, steps):
    if predecessor.unwrap == v_line.unwrap:
        raise Exception("cannot v_shift a v_line")
    return np.roll(matrix, pixel_font_input.scale * steps, axis=0)

@fpn.pipe_node_factory(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN, fpn.PREDECESSOR_PN)
def h_shift(pixel_font_input, matrix, predecessor, steps):
    if predecessor.unwrap == h_line.unwrap:
        raise Exception("cannot h_shift an h_line")
    return np.roll(matrix, pixel_font_input.scale * steps, axis=1)
```

We will need at times to draw points directly, either setting a False pixel to True or vice versa. The `pipe_node_factory` decorated function `flip` will, given coordinate pairs in positional arguments, invert the Boolean value found. Again, we use the `fpn.PN_INPUT` to get the `scale` argument so coordinates can be passed at the unit level, independent of the scale.

```
@fpn.pipe_node_factory(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def flip(pixel_font_input, matrix, *coords):
    matrix = matrix.copy()
    for coord in coords: # x, y pairs
        start = [i * pixel_font_input.scale for i in coord]
        end = [i + pixel_font_input.scale for i in start]
        iloc = slice(start[1], end[1]), slice(start[0], end[0])
        matrix[iloc] = ~matrix[iloc]
    return matrix
```

The following `pipe_node_factory` decorated functions combine variable numbers of `PipeNode` instances passed via positional arguments. The `` union`` and `intersect` functions perform logical OR and logical AND, respectively, on all positional arguments. The `concat` function concatenates frames into a longer frame, inserting a unit-width space

bewteen frames.

```
@fpn.pipe_node_factory()
def union(*args):
    return functools.reduce(np.logical_or, args)

@fpn.pipe_node_factory()
def intersect(*args):
    return functools.reduce(np.logical_and, args)

@fpn.pipe_node_factory(fpn.PN_INPUT)
def concat(pixel_font_input, *args):
    space = np.zeros(
        shape=(
            pixel_font_input.SHAPE[0] * pixel_font_input.scale,
            1 * pixel_font_input.scale
        ),
        dtype=bool,
    )
    concat = lambda x, y: np.concatenate((x, space, y), axis=1)
    return functools.reduce(concat, args)
```

We will need a function to print any frame to standard out. For this, we can create a `pipe_node` decorated function that, given a frame via `fpn.PREDECESSOR_RETURN`, simply walks over the rows and prints the `fpn.PN_INPUT` defined `pixel` when a frame value is True, a space otherwise. Since this node returns the `fpn.PREDECESSOR_RETURN` unchanged, it can be used anywhere in an expression to view a frame mid-pipeline.

```
@fpn.pipe_node(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def display(pixel_font_input, matrix):
    for row in matrix:
        for pixel in row:
            if pixel:
                print(pixel_font_input.pixel, end="")
            else:
                print(end=" ")
        print()
    return matrix
```

We have the tools now to define pipelines to produce the individual characters we need. We will define these in a dictionary, named `chars`, so that we can map string characters to `PipeNode` expressions, pass them to `concat`, and then pipe the results to `display`. For brevity, we will not define a complete alphabet. For most characters the process involves taking the union of a number of lines (some shifted) and then flipping a few pixels. The font here is based on the Visitor font:

http://www.dafont.com/visitor.font

```
chars = {
    "_" : frame,
    "." : frame | flip((2,4)),
    "p" : (
        union(
            frame | v_line,
            frame | h_line,
            frame | h_line | v_shift(2),
```

(continues on next page)

(continued from previous page)

```
        )
        | flip((4,0), (4,1))
    ),
    "y" : (
        frame
        | h_line
        | v_shift(2)
        | flip((0,0), (0,1), (2,3), (2,4), (4,0), (4,1))
    ),
    "0" : union(
        frame | v_line,
        frame | v_line | h_shift(-1),
        frame | h_line,
        frame | h_line | v_shift(-1),
    ),
    "1" : frame | v_line | h_shift(2) | flip((1,0)),
    "2" : (
        union(
            frame | h_line,
            frame | h_line | v_shift(2),
            frame | h_line | v_shift(4),
        )
        | flip((4, 1), (0, 3))
    ),
    "3" : (
        union(
            frame | h_line,
            frame | h_line | v_shift(-1),
            frame | v_line | h_shift(4),
        )
        | flip((2, 2), (3, 2))
    ),
    "4" : (
        union(
            frame | h_line | v_shift(2),
            frame | v_line | h_shift(-1),
        )
        | flip((0, 0), (0, 1))
    ),
    "5" : (
        union(
            frame | h_line,
            frame | h_line | v_shift(2),
            frame | h_line | v_shift(-1),
        )
        | flip((0, 1), (4, 3))
    ),
    "6" : (
        union(
            frame | h_line,
            frame | h_line | v_shift(2),
            frame | h_line | v_shift(-1),
```

(continues on next page)

(continued from previous page)

```
            frame | v_line,
        )
        | flip((4, 3))
    ),
    "7" : (
        (
            frame | h_line
        )
        | flip((2, 4), (2, 3), (3, 2), (4, 1))
    ),
    "8" : (
        union(
            frame | h_line,
            frame | h_line | v_shift(2),
            frame | h_line | v_shift(-1),
            frame | v_line,
            frame | v_line | h_shift(4)
        )
    ),
    "9" : (
        union(
            frame | h_line,
            frame | h_line | v_shift(2),
            frame | h_line | v_shift(-1),
            frame | v_line,
            frame | v_line | h_shift(4)
        )
        | flip((0, 3), (0, 4), (1, 4), (2, 4), (3, 4))
    ),
}
```

We need a function to produce the final `PipeNode` expression. The `msg_display_pipeline` function, given a string message, will return the `PipeNode` expression combining `concat` and `display`, where `concat` is called with PipeNode positional arguments, mapped from `chars`, for each character passed in `msg`. We map the "_" character for any characters not defined in `chars`.

```
def msg_display_pipeline(msg):
    get_char = lambda char: chars.get(char.lower(), chars["_"])
    return concat(*tuple(map(get_char, msg))) | display
```

Finally, we can define the outer-most application function, which will parse command-line arguments for `pixel` and `scale` with `argparse.ArgumentParser`. The `msg_display_pipeline` function is called with the prepared `msg` string, returning `f`, a `PipeNode` function configured to generate and display the `msg` as a banner. A `PixelFontInput` instance is created with the `pixel` and `scale` arguments received from the command line. At last, all *core callables* are called with the evocation of `f` with the `__getitem__` syntax, passing the `PixelFontInput` instance `pixel_font_input`.

```
def version_banner(args):

    p = argparse.ArgumentParser(
        description="Display the Python version in a banner",
    )
```

(continues on next page)

(continued from previous page)

```
    p.add_argument(
        "--pixel",
        default="*",
        help="Set the character used for each pixel of the banner.",
    )
    p.add_argument(
        "--scale",
        default=1,
        type=int,
        help="Set the pixel scale for the banner.",
    )
    namespace = p.parse_args(args)
    assert len(namespace.pixel) == 1
    assert namespace.scale > 0

    msg = "py%s.%s.%s" % sys.version_info[:3]
    f = msg_display_pipeline(msg)

    pixel_font_input = PixelFontInput(pixel=namespace.pixel, scale=namespace.scale)
    f[pixel_font_input]


if __name__ == "__main__":
    version_banner(sys.argv[1:])
```

6.3 Conclusion

After going through this tutorial, you should now have an understanding of:

- How to use `fpn.PipeNode` to do complex numpy array data pipline processing.

Here is all of the code examples we have seen so far:

```
import argparse
import functools
import sys

import numpy as np
import function_pipe as fpn

fpn.pipe_node = functools.partial(
    fpn.pipe_node,
    core_decorator=lambda f: f,
)
fpn.pipe_node_factory = functools.partial(
    fpn.pipe_node_factory,
    core_decorator=lambda f: f,
)

class PixelFontInput(fpn.PipeNodeInput):
```

(continues on next page)

(continued from previous page)

```
    SHAPE = (5,5)

    def __init__(self, pixel="*", scale=1):
        super().__init__()
        self.scale = scale
        self.pixel = pixel

@fpn.pipe_node(fpn.PN_INPUT)
def frame(pixel_font_input):
    shape = tuple(v * pixel_font_input.scale for v in pixel_font_input.SHAPE)
    return np.zeros(shape=shape, dtype=bool)

@fpn.pipe_node(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def v_line(pixel_font_input, matrix):
    matrix = matrix.copy()
    matrix[:, slice(0, pixel_font_input.scale)] = True
    return matrix

@fpn.pipe_node(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def h_line(pixel_font_input, matrix):
    matrix = matrix.copy()
    matrix[slice(0, pixel_font_input.scale), :] = True
    return matrix

@fpn.pipe_node_factory(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN, fpn.PREDECESSOR_PN)
def v_shift(pixel_font_input, matrix, predecessor, steps):
    if predecessor.unwrap == v_line.unwrap:
        raise Exception("cannot v_shift a v_line")
    return np.roll(matrix, pixel_font_input.scale * steps, axis=0)

@fpn.pipe_node_factory(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN, fpn.PREDECESSOR_PN)
def h_shift(pixel_font_input, matrix, predecessor, steps):
    if predecessor.unwrap == h_line.unwrap:
        raise Exception("cannot h_shift an h_line")
    return np.roll(matrix, pixel_font_input.scale * steps, axis=1)

@fpn.pipe_node_factory(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def flip(pixel_font_input, matrix, *coords):
    matrix = matrix.copy()
    for coord in coords: # x, y pairs
        start = [i * pixel_font_input.scale for i in coord]
        end = [i + pixel_font_input.scale for i in start]
        iloc = slice(start[1], end[1]), slice(start[0], end[0])
        matrix[iloc] = ~matrix[iloc]
    return matrix

@fpn.pipe_node_factory()
def union(*args):
    return functools.reduce(np.logical_or, args)

@fpn.pipe_node_factory()
def intersect(*args):
```

(continues on next page)

(continued from previous page)

```
    return functools.reduce(np.logical_and, args)

@fpn.pipe_node_factory(fpn.PN_INPUT)
def concat(pixel_font_input, *args):
    space = np.zeros(
        shape=(
            pixel_font_input.SHAPE[0] * pixel_font_input.scale,
            1 * pixel_font_input.scale
        ),
        dtype=bool,
    )
    concat = lambda x, y: np.concatenate((x, space, y), axis=1)
    return functools.reduce(concat, args)

@fpn.pipe_node(fpn.PN_INPUT, fpn.PREDECESSOR_RETURN)
def display(pixel_font_input, matrix):
    for row in matrix:
        for pixel in row:
            if pixel:
                print(pixel_font_input.pixel, end="")
            else:
                print(end=" ")
        print()
    return matrix

chars = {
    "_" : frame,
    "." : frame | flip((2,4)),
    "p" : (
        union(
            frame | v_line,
            frame | h_line,
            frame | h_line | v_shift(2),
        )
        | flip((4,0), (4,1))
    ),
    "y" : (
        frame
        | h_line
        | v_shift(2)
        | flip((0,0), (0,1), (2,3), (2,4), (4,0), (4,1))
    ),
    "0" : union(
        frame | v_line,
        frame | v_line | h_shift(-1),
        frame | h_line,
        frame | h_line | v_shift(-1),
    ),
    "1" : frame | v_line | h_shift(2) | flip((1,0)),
    "2" : (
        union(
            frame | h_line,
```

(continues on next page)

(continued from previous page)

```
            frame | h_line | v_shift(2),
            frame | h_line | v_shift(4),
        )
        | flip((4, 1), (0, 3))
    ),
    "3" : (
        union(
            frame | h_line,
            frame | h_line | v_shift(-1),
            frame | v_line | h_shift(4),
        )
        | flip((2, 2), (3, 2))
    ),
    "4" : (
        union(
            frame | h_line | v_shift(2),
            frame | v_line | h_shift(-1),
        )
        | flip((0, 0), (0, 1))
    ),
    "5" : (
        union(
            frame | h_line,
            frame | h_line | v_shift(2),
            frame | h_line | v_shift(-1),
        )
        | flip((0, 1), (4, 3))
    ),
    "6" : (
        union(
            frame | h_line,
            frame | h_line | v_shift(2),
            frame | h_line | v_shift(-1),
            frame | v_line,
        )
        | flip((4, 3))
    ),
    "7" : (
        (
            frame | h_line
        )
        | flip((2, 4), (2, 3), (3, 2), (4, 1))
    ),
    "8" : (
        union(
            frame | h_line,
            frame | h_line | v_shift(2),
            frame | h_line | v_shift(-1),
            frame | v_line,
            frame | v_line | h_shift(4)
        )
    ),
```

(continues on next page)

(continued from previous page)

```
    "9" : (
        union(
            frame | h_line,
            frame | h_line | v_shift(2),
            frame | h_line | v_shift(-1),
            frame | v_line,
            frame | v_line | h_shift(4)
        )
        | flip((0, 3), (0, 4), (1, 4), (2, 4), (3, 4))
    ),
}

def msg_display_pipeline(msg):
    get_char = lambda char: chars.get(char.lower(), chars["_"])
    return concat(*tuple(map(get_char, msg))) | display

def version_banner(args):

    p = argparse.ArgumentParser(
        description="Display the Python version in a banner",
    )
    p.add_argument(
        "--pixel",
        default="*",
        help="Set the character used for each pixel of the banner.",
    )
    p.add_argument(
        "--scale",
        default=1,
        type=int,
        help="Set the pixel scale for the banner.",
    )
    namespace = p.parse_args(args)
    assert len(namespace.pixel) == 1
    assert namespace.scale > 0

    msg = "py%s.%s.%s" % sys.version_info[:3]
    f = msg_display_pipeline(msg)

    pixel_font_input = PixelFontInput(pixel=namespace.pixel, scale=namespace.scale)
    f[pixel_font_input]


if __name__ == "__main__":
    version_banner(sys.argv[1:])
```

CHAPTER

SEVEN

INDICES AND TABLES

- genindex

INDEX

Symbols

C

F

I

P

R

S

U