spype Documentation

Release 0.0.0

Derrick Chambers

Jun 16, 2019

Contents

Spype is a [s]imple [py]thon [p]ipelin[e] library.

It facilitates the creation of lightweight, expressive data pipelines.

Spype has three main design goals:

- Independent

  Spype has no required dependencies. It does, however, require graphviz. for visualization and pytest for running the test suite.

- Simple, Declarative API

  Spype provides an intuitive, declarative API for hooking together python callables in order to create arbitrarily complex data pipelines.

- Disciplined

  Spype can (optionally) provide runtime type-checking and compatibility validation in order to help you find bugs faster, and give you more confidence in your data pipelines.

Liscence: BSD

WARNING: spype is brand new and experimental, dont use it in production until it matures a little, and expect frequent API changes.

Documentation:

CHAPTER 1

Quickstart/Installation

1.1 Installation

spype can be installed using pip from pypi:

```
pip install spype
```

You can also clone the repo and run the setup.py.

```
clone https://github.com/d-chambers/spype
cd spype
pip install .
```

1.2 Defining tasks

Tasks define some simple or complex computation. They are most easily created by decorating python callables:

[1]:
```
import spype


@spype.task
def add_two(num: int) -> int:
    return num + 2


@spype.task
def raise_two(num: int) -> int:
    return num ** 2


@spype.task
```

(continues on next page)

(continued from previous page)

```
def divide_two(num: int) -> int:
    return num // 2


@spype.task
def multiply_two(num: int) -> int:
    return num * 2


@spype.task
def split_str(some_str: str) -> str:
    return some_str.split()


@spype.task
def add_together(num1, num2):
    return num1 + num2
```

1.3 Hooking tasks together

Tasks can be create by hooking tasks together using the | operator, starting with `spype.pype_input`.

[2]:
```
pype = spype.pype_input | add_two | raise_two | divide_two
```

Now the pype is a simple callable we can call to push data through the tasks. We will also turn on the print_flow options so that each task will print the data it receives and sends.

[3]:
```
spype.set_options(print_flow=True)  # print out flow of data
pype(2)
```

```
PypeInput got ((2,), {}) and returned (2,)
add_two got ((2,), {}) and returned 4
raise_two got ((4,), {}) and returned 16
divide_two got ((16,), {}) and returned 8
```

[3]:
```
8
```

1.4 Plotting pypes

Pype instances can be visualized if you have graphviz installed.

[4]:
```
pype.plot()
```

[4]:

[5]:
```
pype2 = spype.pype_input | (add_two, raise_two, add_two) | (divide_two, multiply_two)␣
↪| add_two
pype2.print_flow = True
pype2(0)
```

```
PypeInput got ((0,), {}) and returned (0,)
add_two got ((0,), {}) and returned 2
divide_two got ((2,), {}) and returned 1
add_two got ((1,), {}) and returned 3
```

(continues on next page)

(continued from previous page)

```
multiply_two got ((2,), {}) and returned 4
add_two got ((4,), {}) and returned 6
raise_two got ((0,), {}) and returned 0
divide_two got ((0,), {}) and returned 0
add_two got ((0,), {}) and returned 2
multiply_two got ((0,), {}) and returned 0
add_two got ((0,), {}) and returned 2
add_two got ((0,), {}) and returned 2
divide_two got ((2,), {}) and returned 1
add_two got ((1,), {}) and returned 3
multiply_two got ((2,), {}) and returned 4
add_two got ((4,), {}) and returned 6
```

[5]:
```
6
```

[6]:
```
pype2.plot()
```

[6]:

1.5 Hooking pypes in series

[7]:
```
pype3 = pype | pype2  # hook together in series
pype3.plot()
```

[7]:

1.6 Hooking pypes in parallel

[8]:
```
pype4 = pype & pype2  # hook together in parallel
pype4.plot()
```

[8]:

1.7 Hooking pypes at any point

[9]:
```
pype5 = pype2[raise_two] | pype
pype5.plot()
```

[9]:

1.8 Checking compatibility

If you try to hook pypes together that do not have compatible input/outputs (assuming type hints are correct) an exception will be raised before pushing data through the pype.

[10]:
```
pype = spype.pype_input | split_str | add_two

try:
    pype.validate()
except Exception as e:
    print(e)
```

```
output of (some_str:str) -> str is not valid input to (num:int) -> int
```

1.9 Type checking

If an invalid input is given, an exception will be raised right away.

[11]:
```
pype = spype.pype_input | split_str

try:
    pype(2)
except Exception as e:
    print(e)
```

```
PypeInput got ((2,), {}) and returned (2,)
(2,) and {} are not valid inputs for <function split_str at 0x7fe5cc11b268> which␣
↪expects a signature of (some_str:str) -> str
```

The same thing happens if the task returns and incorrect type.

[12]:
```
@spype.task
def str_to_int(obj: str) -> str:
    return int(obj)


pype = spype.pype_input | str_to_int

try:
    pype('1')
except TypeError as e:
    print(f'spype raise exception: {e}')
```

```
PypeInput got (('1',), {}) and returned ('1',)
str_to_int got (('1',), {}) and returned 1
spype raise exception: task: <function str_to_int at 0x7fe5cc14c950> returned: 1␣
↪which is not consistent with expected output type of: <class 'str'>
```

1.10 Constant dependencies

Input arguments for a task can come from the previous task, or can be set to a constant value using the partial method on the task, or the set_item method on the pype.

[13]:
```
pype = spype.pype_input | add_together.partial(num1=1)
pype(4)
```

```
PypeInput got ((4,), {}) and returned (4,)
add_together got ((1, 4), {}) and returned 5
```

[13]:
```
5
```

[14]:
```
pype = spype.pype_input | add_together
pype['num1'] = 1
pype(4)
```

```
PypeInput got ((4,), {}) and returned (4,)
add_together got ((1, 4), {}) and returned 5
```

[14]:
```
5
```

1.11 Task dependencies

Tasks can also depend on other resolvable tasks (not just the prior task in the graph).

[15]:
```
pype = spype.pype_input | add_together.partial(num1=raise_two)
pype &= raise_two  # add raise to in parallel

pype.plot()
```

[15]:

The output of `raise_two` will be passed to `add_togher` as parameter num1.

[16]:
```
pype(2)
```

```
PypeInput got ((2,), {}) and returned (2,)
raise_two got ((2,), {}) and returned 4
add_together got ((4, 2), {}) and returned 6
```

[16]:
```
6
```

Learning more

Spype has many other features, be sure to read through the *tutorial*.

CHAPTER 2

Should I Use Spype?

WARNING: spype is brand new. Don't use it in production until it matures a little.

The following questions will help you decide if Spype is the right library for you, or if you are better of using something else.

2.1 Can you use python 3.6 or greater?

Spype only runs on python 3.6+, so if you are still stuck on 2.7 you cannot use spype. Also, I feel for you.

2.2 Does my data fit on a single machine?

Although spype is designed to play nice multiprocessing and multithreading, it is not really designed to run across a network. If the data you are trying to process do not fit on a single machine I recommend you look elsewhere.

2.3 Do you want to limit external dependencies?

Spype does not have any required external dependencies, it runs on pure python.

2.4 Do you value expressiveness and maintainability over short execution time?

Spype provides a rich, concise API, and provides several features to help you discipline your data flows. However, these features come at performance cost that may or may not be significant depending on your application.

2.5 Do you want to "push" or "pull" you data?

If you have a piece of data and you can describe the steps that need to be performed on it, you probably want to use a pipeline system that implements a "push" paradigm (like spype). If, however, you can describe the data you want to generate, and steps required to do so, you probably want a "pull" system (like dask's custom graphs).

CHAPTER 3

Similar Projects

The most similar project to spype that I have encountered is consecution. It looks like an excellent package, be sure to take a look at it.

There are also other libraries that somewhat fit into the same space as spype. Here are a few (in no particular order): luigi, airflow, pinball, dagobah, celery, dask, streamz.

There is also an awesome list of data pipelines you should checkout.

CHAPTER 4

Tutorial

4.1 Tasks

Tasks are the basic unit of spype. Each task encapsulates some unit of work. Tasks can be defined either through inheritance or using a decorator. For example, the following statements are (roughly) equivalent:

4.1.1 Task through function decorator

[1]:
```
import spype

@spype.task
def to_int(obj):
    return int(obj)
```

4.1.2 Task through inheritance

[2]:
```
class ToInt(spype.Task):

    def __init__(self):
        pass  # init some state when needed

    def __call__(self, obj):
        return int(obj)

to_int = ToInt()
```

Note: A call method must be defined.

Now calling the function, or an instance of the task class, will behave exactly like before. There is however, an added `run` method which will hook in all the spype machinery which includes optional type checking and callbacks.

4.1.3 Type checking

Tasks support run-time type checking using type hints. Defining a type hint will both validate data input to a task and ensure tasks are hooked together correctly (see pypes). They are completely optional, but I recommend you use them on real-world spype applications to help you not screw up.

[3]:
```
@spype.task
def str_to_int(obj: str) -> int:
    return int(obj)

try:
    str_to_int.run(1)
except TypeError:
    print('TypeError raised')
```

```
TypeError raised
```

Although spype's type system is not perfect, the use of most of the classes defined in the `typing` module, such as `Optional`, `Union`, `List`, etc. are supported.

[4]:
```
from typing import Union, TypeVar, Optional


# -------------- using Optional, Union
@spype.task
def is_even(num: int) -> Optional[Union[bool, int]]:
    """ bad function to return True if even, else the number if odd.
    Return None if num == 13 """
    if num == 13:  # this will cause a type error
        return 'This is a mistake in the code'
    elif num % 2 == 0:
        return True
    else:
        return num

is_even(1)  # returns 1
is_even(2)  # returns True

# this raises TypeError as a str is not part of specified output
try:
    is_even.run(13)
except TypeError:
    print('TypeError raised')
```

```
TypeError raised
```

[5]:
```
# ------------------ using typevar

TV = TypeVar('TV', int, float, str)

@spype.task
def add_to_self(obj: TV) -> TV:
    """ add an object to itself, return """
    if obj == 13:  # unlucky number, here is another bug
        return float(13 + 13)
    return obj + obj
```

(continues on next page)

(continued from previous page)

```
# these are all good
add_to_self(10)  # returns 20
add_to_self('_java_sucks_')  # returns '_java_sucks__java_sucks_'
add_to_self(13.)  # returns 26.0


# this raises because 13 is an int and 26.0 is a float (not the same type as input)
try:
    add_to_self.run(13)
except TypeError:
    print('TypeError raised')
```

```
TypeError raised
```

4.1.4 Callbacks

Each task supports any number of one of four types of callbacks:

Callback	Description
on_start	Called before running the task
on_failure	Called if the task raises an unhandled exception
on_success	Called if the task did not raise an unhandled exception
on_finish	Called when the task finishes, regardless of exceptions

The callbacks can be defined using keywords in the decorator call or implemented as class methods. These are useful for customizing how failures are handled (e.g. dropping into a debugger), applying data validators, implementing hooks, etc.

Generic Callback Fixtures

Rather than requiring a specific signature for each type of callback, you can control the data the callbacks receive based on the argument names defined in the functions. This is known as a fixture (this concept was stolen from pytest).

spype.TASK_FIXTURES is a mapping that contains the supported task fixtures and a brief description of each:

[6]:
```
spype.TASK_FIXTURES
```

[6]:
```
mappingproxy({'args': 'A tuple of arguments passed to current task',
              'e': 'The exception object if one was raised, else None',
              'inputs': 'A tuple of (args, kwargs) passed as input to current task',
              'kwargs': 'A dict of keywork arguments passed to current task',
              'outputs': 'The outputs of calling a task, or None',
              'self': 'A reference to the current task object',
              'signature': "The signature of the task's run method",
              'task': 'A reference to the current task object'})
```

Here are a few simple examples of callbacks:

[7]:
```
def on_failure(task, e, inputs):
    """ e is the exception raised, the inputs to the task are bound to args and␣
↪kwargs """

def on_success(task, outputs):
    """ return values of the task are bound to args and kwargs """
```

Callbacks can be used with both class-based tasks and decorator tasks like so:

[8]:
```
# attach callbacks in task class
class StrToInt(spype.Task):
    def __call__(self, obj: str) -> int:
        if obj == '13':
            raise ValueError('unlucky numbers not accepted')
        return int(obj)

    def on_failure(self, e, inputs):
        print(f'{self} raised: {e}\n')
        print(f'inputs are: {inputs}')


str_to_int = StrToInt()

str_to_int.run('13') # prints name and exception, then input args and kwargs.
```

```
StrToInt instance raised: unlucky numbers not accepted

inputs are: (('13',), {})
```

[9]:
```
# this is the equivelent function-based approach:

def on_failure(task, e, inputs):
        print(f'{task} raised {e}')
        print(f'input args: {inputs}')


@spype.task(on_failure=on_failure)
def str_to_int(obj: str) -> int:
    if obj == '13':
        raise ValueError('unlucky numbers not accepted')
    return None

str_to_int.run('13')
```

```
<function str_to_int at 0x7efe4c395d90> raised unlucky numbers not accepted
input args: (('13',), {})
```

Task Specific Callbacks

In addition to the task fixtures, callbacks can also request any of the named parameters in the task signature. However, if the task does not have the requested parameter this will raise an exception.

[10]:
```
def on_start(a, c):
    print(f'on_start callback got a: {a}, c: {c}')


@spype.task(on_start=on_start)
def some_task(a, b, c):
    pass


some_task.run(1, 2, 3)
```

```
on_start callback got a: 1, c: 3
```

Replacing Task Output

If a callback returns any value other than `None`, the task will return that value. The value the callback returns will be output immediately without running the other callbacks.

[11]:
```
def on_start(num1):
    return float(num1)

def on_finish():
    print('This line will never get printed')

@spype.task(on_start=on_start, on_finish=on_finish)
def return_float(num1) -> float:
    return num1

return_float.run(1)
```

[11]:
```
1.0
```

Terminating Task Execution

If a task should stop execution, and return None (which will prevent any downstream processing carried out by pypes) `ExitTask` exception can be raised.

[12]:
```
def stop_it():
    print('terminating task execution')
    raise spype.ExitTask

@spype.task(on_start=stop_it)
def some_task(a):
    return a

some_task.run(1)
```

```
terminating task execution
```

Replacing Callbacks

Callbacks can also be attached after task definition, or replace permanently or temporarily.

[13]:
```
def dont_care(e):
    """ use this callback when you are beyond caring, maybe a friday afternoon? """
    print(f'just swallowed: "{e}"')


@spype.task
def naming_is_hard():
    raise ValueError('bad function call')


# You can also set callbacks default callbacks for all tasks using options
with spype.options(on_failure=dont_care):
    naming_is_hard.run()  # wont raise
```

```
just swallowed: "bad function call"
```

[14]:
```
# callbacks can be overwritten, but don't do this unless you have a good reason
str_to_int.on_failure = dont_care  # replaces old callback (be careful with this one)

# you can also define a sequence of callbacks to be executed in order
str_to_int.on_failure = (dont_care, on_failure)  # calls dont_care first then on_
↪failure
```

4.2 Pypes

The `Pype` class is used to hook tasks together, and to controll data flow from task to task (yes, "pype" is a bad pun for "python pipes"). The basic idea; you can just hook tasks together in an intuitive way and use the resulting pype as a callable to kick the whole thing off.

Some operators for hooking tasks/pypes together:

Operator	Use
|	Hook tasks/pypes in series
&	Hook tasks/pypes in parallel
|=	Hook tasks/pypes in serial in place
&=	Hook task/pypes in parallel in place
<<	Fan outputs of tasks to next task
>>	Aggregate outputs of previous task

Now let's look at some examples, but first let's define some stupid simple tasks:

[15]:
```
import spype
from spype import pype_input


@spype.task
def add_two(num: int) -> int:
    return num + 2


@spype.task
def raise_two(num: int) -> int:
    return num ** 2


@spype.task
def divide_two(num: int) -> int:
    return num // 2


@spype.task
def divide_numbers(num1: int, num2: int) -> int:
    return num1 // num2


@spype.task
def multiply_two(num: int) -> int:
    return num * 2
```

(continues on next page)

(continued from previous page)

```
@spype.task
def split_str(some_str: str) -> str:
    return some_str.split()


@spype.task
def add_together(num1, num2):
    return num1 + num2
```

Tasks can be hooked together in series using the | operator, in which case the output of one task will be fed directly as the input to the next task.

[16]:
```
pype = spype.pype_input | add_two | multiply_two | raise_two
```

4.2.1 Pype Visualization

The plot function is used to visualize the relationships between tasks in a pype. You need graphviz and python package called graphviz installed on your system.

[17]:
```
pype.plot()
```

[17]:

Notice the pype starts with a special task called `PypeInput`. This will always be the case.

4.2.2 Print Pype Outputs

We can also turn on printing to better understand how data are flowing from task to task. To turn this feature on for the remainder of this python session this we will use spype.options. It can also be used as a context manager if you wanted to enable this feature temporarily (eg for debugging).

[18]:
```
spype.options(print_flow=True)
pype(2)
```

```
PypeInput got ((2,), {}) and returned (2,)
add_two got ((2,), {}) and returned 4
multiply_two got ((4,), {}) and returned 8
raise_two got ((8,), {}) and returned 64
```

[18]:
```
64
```

Notice the structure of the data as it moves between tasks is (args, kwargs) where args is tuple of positional arguments and kwargs is a dict of named arguments. For example the input was ((2,), {}). This SO question explains args and kwargs nicely if you are not familiar.

4.2.3 Mixing pypes and tasks

We can also hook pypes together, or any combinations of pypes and tasks.

[19]:
```
pype = spype.pype_input | add_two | multiply_two
pype |= raise_two  # modfies pype1 by attaching raise_two at the end
# pype1 and pype are equivelent
```

(continues on next page)

(continued from previous page)

```
pype(0)
pype.plot()
```

```
PypeInput got ((0,), {}) and returned (0,)
add_two got ((0,), {}) and returned 2
multiply_two got ((2,), {}) and returned 4
raise_two got ((4,), {}) and returned 16
```

[19]:

4.2.4 Hooking pypes in parallel

Pypes/tasks can be hooked in parallel using the `&` operator.

[20]:
```
pype2 = pype & divide_two
pype2.plot()
```

[20]:

[21]:
```
pype3 = pype & pype
pype3.plot()
```

[21]:

4.2.5 Broadcasting and Merging tasks

If you would like a pype/tasks output to be given as the input to several pypes/tasks (known as broadcasting), simply use a tuple.

[22]:
```
pype = spype.pype_input | (multiply_two, divide_two)
pype.plot()
```

[22]:

Which results in the following execution:

[23]:
```
pype(2)
```

```
PypeInput got ((2,), {}) and returned (2,)
multiply_two got ((2,), {}) and returned 4
divide_two got ((2,), {}) and returned 1
```

[23]:
```
1
```

And the merging can also be done via a tuple

[24]:
```
pype = spype.pype_input | (multiply_two, raise_two) | divide_two
pype.plot()
```

[24]:

and the excution looks like this:

[25]:
```
pype(2)
```

```
PypeInput got ((2,), {}) and returned (2,)
raise_two got ((2,), {}) and returned 4
divide_two got ((4,), {}) and returned 2
multiply_two got ((2,), {}) and returned 4
divide_two got ((4,), {}) and returned 2
```

[25]:
```
2
```

Here is an unnecessarily complex example:

[26]:
```
# define any pypes
p1 = spype.pype_input | add_two | raise_two | raise_two | add_two
p2 = spype.pype_input | add_two | add_two
# hook pypes to the end of add_two and feed outputs into divide_two
complex_pype = (spype.pype_input | add_two | (p1, p2, spype.pype_input | add_two, p2,␣
↪p1)
                | (divide_two, add_two) | (divide_two, add_two))
complex_pype &= (add_two, divide_two)
complex_pype.plot()
```

[26]:

4.2.6 Attaching tasks/pypes anywhere

In some cases it may be necessary to access a particular task directly when adding new tasks. This can be done using the get_item interface on a pype object. eg:

[27]:
```
p1 = spype.pype_input | add_two | raise_two | divide_two
```

Now once this is defined it might be a bit tricky to attach something to the middle task but using the get_item interface makes it simple

[28]:
```
pype = p1[add_two] | multiply_two
pype(2)
pype.plot()
```

```
PypeInput got ((2,), {}) and returned (2,)
add_two got ((2,), {}) and returned 4
raise_two got ((4,), {}) and returned 16
divide_two got ((16,), {}) and returned 8
multiply_two got ((4,), {}) and returned 8
```

[28]:

4.2.7 Getting intermediate values

If we want to see the last value a task returned we can use the get_item interface (square brackets) on the pypes's `outputs` attribute like so:

[29]:
```
print(pype.outputs[raise_two])
print(pype.outputs[multiply_two])
```

```
((16,), {})
((8,), {})
```

If you want the given to the pype object when called, you can use `pype_input`

[30]:
```
print(pype.outputs[spype.pype_input])
```

```
((2,), {})
```

WARNING: Accessing values in this way is mainly for debugging as this is not reliable when running multiple threads or processes. Also, tasks used as keys must be unique in the pype, see the gotchas section.

4.2.8 Conditionals

If you want to trigger tasks only under certain conditions you can use the `iff` attribute of a task. `iff` takes a callable returns a single boolean. Like callbacks, you can use any of the fixtures or named arguments which are part of the task signature. For example:

[31]:
```
def is_even(num):
    return (num % 2) == 0

pype = pype_input | add_two | multiply_two.iff(is_even) | raise_two

# if number is not even only add_two gets run
pype(3)
```

```
PypeInput got ((3,), {}) and returned (3,)
add_two got ((3,), {}) and returned 5
multiply_two got ((5,), {}) and returned None
```

[31]:
```
5
```

Notice how execution occurred only through `multiply_two`, and stopped before `raise_two` when None was returned.

[32]:
```
pype(2)  # if the number is even it will go all the way through
```

```
PypeInput got ((2,), {}) and returned (2,)
add_two got ((2,), {}) and returned 4
multiply_two got ((4,), {}) and returned 8
raise_two got ((8,), {}) and returned 64
```

[32]:
```
64
```

Asking for an Spype fixtures also works

[33]:
```
def print_pype(pype, task, wrap):
    print(f'called predicate with {(pype, task, wrap)}')
    return True

pype = pype_input | add_two.iff(print_pype)
pype(1)
```

```
PypeInput got ((1,), {}) and returned (1,)
called predicate with (Pype isntance

NODES:

[PypeInput instance, <function add_two at 0x7efe4c354598>]

 EDGES:

[(PypeInput instance, <function add_two at 0x7efe4c354598>)]

DEPENDENCIES:

 []
, <function add_two at 0x7efe4c354598>, task wrap of <function add_two at␣
↪0x7efe4c354598>)
add_two got ((1,), {}) and returned 3
```

[33]:
```
3
```

4.2.9 Fanning out

A single task may produce a sequence of objects, each of which need to be handled independently for the rest of the processing pypeline. Spype calls fanning out. It can be done using the `fan` attribute or by using the left bytes shift operator (<<, think of it as widening the outputs), although this can be a bit tricky due to byte shift occurring before the pype.

An example:

[34]:
```
from typing import List

@spype.task
def split_on_space(obj: str) -> List[str]:
    """ yield the string split on spaces """
    return obj.split(' ')


@spype.task
def print_input(obj: str):
    print(f'just got {obj}')

# note the parentheses
pype = (spype.pype_input | split_on_space) << print_input
pype.plot()
```

[34]:

[35]:
```
pype('szechuan sauce snafu')
```

```
PypeInput got ((’szechuan sauce snafu’,), {}) and returned (’szechuan sauce snafu’,)
split_on_space got ((’szechuan sauce snafu’,), {}) and returned [’szechuan’, ’sauce’,␣
↪’snafu’]
just got szechuan
print_input got ((’szechuan’,), {}) and returned None
just got sauce
print_input got ((’sauce’,), {}) and returned None
just got snafu
print_input got ((’snafu’,), {}) and returned None
```

[35]:
```
['szechuan', 'sauce', 'snafu']
```

which is equivalent to

[36]:
```
pype = spype.pype_input | split_on_space | print_input.fan()
pype('szechuan sauce snafu')
```

```
PypeInput got ((’szechuan sauce snafu’,), {}) and returned (’szechuan sauce snafu’,)
split_on_space got ((’szechuan sauce snafu’,), {}) and returned [’szechuan’, ’sauce’,␣
↪’snafu’]
just got szechuan
print_input got ((’szechuan’,), {}) and returned None
just got sauce
print_input got ((’sauce’,), {}) and returned None
just got snafu
print_input got ((’snafu’,), {}) and returned None
```

[36]:
```
['szechuan', 'sauce', 'snafu']
```

4.2.10 Aggregating

Aggregate does the opposite of fanning out. It stores all the outputs of a task (until it is done executing) then sends them as a tuple to the downstream tasks. This can be activated using the `agg` attribute or the right byte shift operator (>> think of it as "funneling" outputs into a task).

[37]:
```
from typing import Tuple, TypeVar

tv = TypeVar('tv')

@spype.task
def join_str(obj: List[str]) -> str:
    return ' '.join(obj)


@spype.task
def pass_through(x: tv) -> tv:
    return x


pype = (spype.pype_input | split_on_space) << pass_through.agg() >> join_str | print_
↪input
pype('a full string')
```

```
PypeInput got (('a full string',), {}) and returned ('a full string',)
split_on_space got (('a full string',), {}) and returned ['a', 'full', 'string']
pass_through got (('a',), {}) and returned a
pass_through got (('full',), {}) and returned full
pass_through got (('string',), {}) and returned string
join_str got ((['a', 'full', 'string'],), {}) and returned a full string
just got a full string
print_input got (('a full string',), {}) and returned None
```

[37]:
```
'a full string'
```

[38]:
```
pype.plot()
```

[38]:

4.2.11 Dependency injection

If a task needs data that are not passed from the previous task you can use the `partial` method. For example, if the `num2` parameter in the `divide_two` function should always be 2:

[39]:
```
pype = spype.pype_input | add_two | raise_two | divide_numbers.partial(num2=2)
```

If the value of num2 depends on the output of another task, you can assign the task in the partial statement. Then, the value returned by the task will be assigned to the parameter.

[40]:
```
pype = spype.pype_input | add_two | (raise_two, divide_numbers.partial(num1=raise_
↪two))
pype(2)
```

```
PypeInput got ((2,), {}) and returned (2,)
add_two got ((2,), {}) and returned 4
raise_two got ((4,), {}) and returned 16
divide_numbers got ((16, 4), {}) and returned 4
```

```
[40]: 4
```

You can also define an argument's value using the set_item interface of the pype. Then, all unmet arguments with the same name used in the set_item call will be given that value or, the the value returned by a task if a task instance is used. For example:

```
[41]: pype = spype.pype_input | add_two | raise_two | divide_numbers
      # if we called pype now it would raise because divide needs two arguments
      # however, we can tell the pype what the value should be.
      pype['num2'] = 2
      # now 2 will automatically be passed to divide when no other value for num2 is found
      pype(2)  # returns 8

      PypeInput got ((2,), {}) and returned (2,)
      add_two got ((2,), {}) and returned 4
      raise_two got ((4,), {}) and returned 16
      divide_numbers got ((16, 2), {}) and returned 8

[41]: 8
```

```
[42]: pype = spype.pype_input | add_two | raise_two | divide_numbers
      # we could also assign a task, indicating the output of the task should be used
      pype['num2'] = add_two
      pype(2)  # returns 4 because num2 = the result of add_two (4)

      PypeInput got ((2,), {}) and returned (2,)
      add_two got ((2,), {}) and returned 4
      raise_two got ((4,), {}) and returned 16
      divide_numbers got ((16, 4), {}) and returned 4

[42]: 4
```

You need to be a little careful with this, if you define circular dependencies an InvalidPype exception will be raised the first time you try calling a malformed pype.

4.2.12 Debugging

Admittedly, it can be a bit challenging to navigate spype's working to get to debugging your tasks. For this reason, the pype Class has a `debug` method, which will essentially drop you into a debugger before each task or selected tasks get called.

```
[43]: pype = pype_input | multiply_two | raise_two

      with pype.debug() as p:
          pass  # calling the pype will drop you into a debugger at the start of every task
```

4.2.13 Gotchas

Spype still has a few warts which I hope to fix in the future. For now, there are mainly two things that might trip you up:

Task uniqueness

As demonstrated, tasks can be used as keys to set dependencies and check results. If a single task is used multiple times in a pype, however, some of these features will not be reliable, and will probably raise Exceptions in future

versions.

For example, the following code will return the latest value returned by the `add_two` task.

[44]:
```
pype = pype_input | add_two | add_two
pype(2)
print(pype.outputs[add_two])
```

```
PypeInput got ((2,), {}) and returned (2,)
add_two got ((2,), {}) and returned 4
add_two got ((4,), {}) and returned 6
((6,), {})
```

If the pype was more complex, however, it would be difficult to know which `add_two` ouputs were being referred to.

[45]:
```
pype = pype_input | (add_two, add_two, (pype_input | add_two | add_two))
pype &= add_two
pype(2)
print(pype.outputs[add_two])
```

```
PypeInput got ((2,), {}) and returned (2,)
add_two got ((2,), {}) and returned 4
add_two got ((2,), {}) and returned 4
add_two got ((2,), {}) and returned 4
add_two got ((4,), {}) and returned 6
add_two got ((2,), {}) and returned 4
((4,), {})
```

Task uniqueness: best practices

There are a few options for easily avoiding the issue if you do need to use a pype with multiple identical tasks (in order of preference):

1. Don't depend on features that require unique tasks
2. Use a class based task and instantiate different objects
3. Copy tasks using the copy function

The last option will probably break the ability of the pickle module to serialize the pype.

option 1

Most of the pype features will work fine, so if you can rethink how the pype is being used to avoid needing any of the tasks to be unique it is probably the most prudent course of action.

option 2

Instances of class based tasks will work as expected:

[46]:
```
class AddTwo(spype.Task):
    def __call__(self, num: int) -> int:
        return num + 2

add2 = [AddTwo() for _ in range(3)]
```

(continues on next page)

(continued from previous page)

```
pype = pype_input | add2[0] | add2[1] | add2[2]
pype(2)

for add in add2:
    print(pype.outputs[add])
```

```
PypeInput got ((2,), {}) and returned (2,)
AddTwo got ((2,), {}) and returned 4
AddTwo got ((4,), {}) and returned 6
AddTwo got ((6,), {}) and returned 8
((4,), {})
((6,), {})
((8,), {})
```

option 3

You can also simply copy a task using the `copy` method, though the resulting task will probably not be serializable using vanilla pickle so it may mess up multiprocessing.

[47]:
```
add2 = add_two.copy(), add_two.copy(), add_two.copy()

pype = pype_input | add2[0] | add2[1] | add2[2]
pype(2)

for add in add2:
    print(pype.outputs[add])
```

```
PypeInput got ((2,), {}) and returned (2,)
add_two got ((2,), {}) and returned 4
add_two got ((4,), {}) and returned 6
add_two got ((6,), {}) and returned 8
((4,), {})
((6,), {})
((8,), {})
```

CHAPTER 5

Contributing

Contributions to spype are welcome.

If you find a problem, would like to propose a change, or have a question, use the issue tracker.

I will be making some rather rapid changes until version 0.1.0, after which I more or less plan to follow the obspy branching model. Basically, if you want to fix a bug use branch "maintenance_version" where version is the last released version. If you want to add a new feature just branch off master.

CHAPTER 6

API

6.1 Task

class spype.core.task.**Forward**
A task for simply forwarding inputs to the next task.

class spype.core.task.**PypeInput**
A singleton Task that is used, explicitly or implicitly, to begin each pype.

class spype.core.task.**Task**
An abstract class whose subclasses encapsulate a unit of work.

copy()
Return a deep copy of task.

Return type *Task*

get_name()
Return the name of task.

Return type str

get_option(*option*)
Returns an option defined in self or defer to Task MRO.

Parameters **option** (str) – A supported Spype option. See spype.options.

Return type Any

get_signature()
return signature of bound run method

Return type Signature

run(**args*, *_fixtures=None*, *_callbacks=None*, *_predicate=None*, ***kwargs*)
Call the task's __call__ and handle spype magic in the background.

Run essentially performs the following steps:

1. Try to bind args and kwawrgs to the task signature
2. If bind raises, look for missing arguments in _fixtures
3. Rebind args and kwargs to signature with new args if needed
4. Run on_start callback, if defined
5. Run task call method (or function)
6. Run on_failure callback if defined and an exception was raised
7. Run on_success callback if defined and no exception was raised
8. Run on_finish callback, if defined
9. Return output of call method, or output of any callback if any non-None values were returned.

Parameters

- **_fixtures** (Optional[Mapping[str, Any]]) – A dict of fixtures. Keys are parameters that might be used by callbacks and values are the values to substitute.
- **_callbacks** (Optional[Mapping[str, Union[Callable, Sequence[Callable]]]]) – A dict of callbacks. Keys must be supported callback names (str) and values must be callables.
- **_predicate** (Union[Callable[..., bool], Sequence[Callable[..., bool]], None]) – A single function, or sequence of functions, that return a bool. Standard fixtures can be used, just like in callbacks.

validate_callback(*callback*)
Raise TypeError if callback is not a valid callback for this task.

Parameters **callback** (Callable) – Any callable

Return type None

validate_callbacks()
Iterate over all attached callbacks and raise TypeError if any problems are detected.

Return type None

wrap(**args*, ***kwargs*)
Instantiate a Wrap instance from this task.

Args and kwargs are passed to Wrap constructor.

Returns

Return type *Wrap*

spype.core.task.**task**(*func=None*, ***, *on_start=None*, *on_failure=None*, *on_success=None*, *on_finish=None*, *predicate=None*, ***kwargs*)
Decorator for registering a callable as a tasks.

This essentially adds the Task class attributes to a function and returns the function. This means the function will behave as before, but will have the Task class attributes attached. This approach is needed so that the tasks are pickable, else returning Task instances would work.

Parameters

- **func** (Optional[Callable]) – A callable to use as a task
- **on_start** (Union[Callable, Sequence[Callable], None]) – Callable which is called before running task

- **on_failure** (Union[Callable, Sequence[Callable], None]) – Callable which will be called when a task fails
- **on_success** (Union[Callable, Sequence[Callable], None]) – Callable that gets called when a task succeeds
- **on_finish** (Union[Callable, Sequence[Callable], None]) – Callable that gets called whenever a task finishes

Returns An instance of Task

Return type *Task*

6.2 Wrap

Wrap class. Used to wrap tasks defined in pype processing lines.

class spype.core.wrap.**Wrap**(*task*, ***kwargs*)

Class to encapsulate a task.

agg(*scope='object'*)

Mark wrap as aggregating output from input tasks.

This will store all outputs of previous task in a list then feed to this task when it is done.

Return type *Wrap*

compatible(*other*, *extra_params=None*)

Return True if self (current wrap) provides valid inputs for other.

Parameters

- **other** (Union[*Task*, *Wrap*]) – Another task or wrap
- **extra_params** (Optional[Mapping[~KT, +VT_co]]) – A mapping of extra parameters

Returns

Return type bool

conditional_name

return the name of the predicate, else None

fan()

Mark Wrap as fanning out.

This will cause it to iterate output and queue one item at a time.

Return type *Wrap*

fit(**args*)

Method to adapt order/name of the outputs.

This is useful if the output order/name needs to be adjusted to work with the next Wrap in the Pype.

Parameters **args** – A sequence of ints/strings for mapping output into args and kwargs

Returns

Return type Wrap instance

iff(*predicate=None*)

Register a condition that must be true for data to continue in pype.

Parameters **predicate** (Union[Callable[..., bool], Sequence[Callable[..., bool]], None]) – A function that takes the same inputs as the task and returns a boolean.

Returns

Return type *Wrap*

par(***kwargs*)
Set values for paramters.

If this task does not receive all the required arguments the ones set with this function will be used.

Return type *Wrap*

partial(***kwargs*)
Set values for paramters.

If this task does not receive all the required arguments the ones set with this function will be used.

Return type *Wrap*

signature
return signature which indicates the arguments expected as input, excluding partials

Return type Signature

task_name
return the short name of the wrapped task

6.3 Pype

Pype class and supporting functions.

class spype.core.pype.**Pype**(*arg=None*, *name=None*)
Class to control the data flow between tasks.

Parameters **arg** (Union[*Task*, *Wrap*, *Pype*, str, None]) – A task, pype, or any hashable that has been registered as a pype. If hashable, a copy of the registered pype will be returned.

add_callback(*callback*, *callback_type*, *tasks=None*)
Add a callback to all, or some, tasks in the pype. Return new Pype.

Parameters

- **callback** (*<built-in function callable>*) – The callable to attach to the tasks in the pype
- **callback_type** (str) –

  The type of callback: supported types are: on_start, on_failure, on_success, and on_exception

- **tasks** (Optional[Sequence[*Task*]]) – A sequence of tasks to apply callback to, else apply to all tasks.

Returns

Return type A copy of Pype

debug(*tasks=None*, *callback_type='on_start'*)
Return of copy of Pype with debugging callbacks set.

Optionally, a list of tasks to set debug on can be defined to limit the breakpoints to only include those tasks. The callback where to debugger is called is also configurable.

Parameters

- **tasks** (Optional[Sequence[*Task*]]) – If not None a task or sequence of tasks to debug. If None debug all tasks.
- **callback_type** – The callback to set debug function. Controls where in the execution cycle debugging tasks place

Returns

Return type A copy of this pype.

iff(*predicate*, *inplace=False*)

Run data through the pype only if predicate evaluates to True.

Parameters

- **predicate** (Callable[[Any], bool]) – A callable that returns a boolean and takes the same inputs as the first task in the pype (excluding pype_input)
- **inplace** – If True modify the pype in place, else return a pype with iff applied.

Returns

Return type *Pype*

plot(*file_name=None*, *view=True*)

Plot the graph :type file_name: Optional[str] :param file_name: The name of the graph viz file output. :type view: bool :param view: If True display the graph network.

Returns

Return type Instance of graphviz.Digraph

register(*name*)

Register a pype under name.

Allows accessing the pype, or copies of it, later.

Parameters **name** (Hashable) – Any Non-None hashable

Return type None

validate()

Run checks on the pype to detect potential problems.

Will raise an InvalidPype exception if compatibility issues are found, or a TypeError if any invalid callbacks are found.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index