pythonflow Documentation

Release 0.3.0

Till Hoffmann

Oct 29, 2019

Contents

Pythonflow is a simple implementation of dataflow programming for python. Users of Tensorflow will immediately be familiar with the syntax.

At Spotify, we use Pythonflow in data preprocessing pipelines for machine learning models because

- it automatically caches computationally expensive operations,
- any part of the computational graph can be easily evaluated for debugging purposes,
- it allows us to distribute data preprocessing across multiple machines.

CHAPTER 1

Installation of Pythonflow

This section guides you through the installation of Pythonflow, which only supports python 3.

1.1 Installation for users

You can install Pythonflow from the command line like so.

```
$ pip install pythonflow
```

If you don't have `pip` installed, have a look at this guide.

1.2 Installation for developers

If you want to contribute to Pythonflow, we recommend you fork the GitHub repository of Pythonflow and clone your fork like so.

```
$ git clone git@github.com:<your username>/pythonflow.git
```

Next, create a virtual environment or set up a Conda environment and install Pythonflow in development mode like so.

```
$ pip install -e /path/to/your/clone/of/pythonflow
```

1.2.1 Testing your installation

To make sure your local installation of Pythonflow works as expected (and to make sure your changes don't break a test when you submit a pull request), you can run Pythonflow's unit tests. First, install the requirements needed for tests like so.

```
$ pip install -r requirements/test.txt
```

Then run the tests like so.

```
$ pylint pythonflow
$ py.test --cov --cov-fail-under=100 --cov-report=term-missing
```

We have also provided a `Makefile` so you don't have to remember the commands for testing and you can run the following instead.

```
$ make tests
```

For more details on how to contribute, have a look at GitHub's developer guide and make sure you have read our guidelines for contributing.

1.2.2 Building the documentation

If you want to see what the documentation of your local changes looks like, first install the requirements needed for compiling it like so.

```
$ pip install -r dev-requirements.txt
```

Then compile the documentation like so.

```
$ make docs
```

CHAPTER 2

Using Pythonflow

Pythonflow is a pure python library for dataflow programming. In contrast to the usual control flow paradigm of python, dataflow programming requires two steps. First, we set up a directed acyclic graph (DAG) of operations that represents the computation we want to perform. Second, we evaluate the operations of the DAG by providing input values for some of the nodes of the graph. But learning by example is usually easier than theory. So here we go.

```
import pythonflow as pf


with pf.Graph() as graph:
    a = pf.constant(4)
    b = pf.constant(38)
    x = a + b
```

The code above creates a simple graph which adds two numbers. Importantly, none of the computation has been performed yet, and *x* is an operation that must be evaluated to obtain the result of the addition:

```
>>> x
<pf.func_op '...' target=<built-in function add> args=<2 items> kwargs=<0 items>>
```

Although a little cryptic, the output tells us that `x` is an operation that wraps the built-in function `add`, has two positional arguments, and no keyword arguments. We can evaluate the operation to perform the computation by calling the `graph` like so.

```
>>> graph(x)
42
```

pythonflow supports all operations of the standard python data model that do not change the state of the operation (think *:code:‘const* member functions <https://isocpp.org/wiki/faq/const-correctness#const-member-fns>‘_) in C++) because operations are assumed to be stateless. For example, `b = a + 1` is allowed whereas `a += 1` is not. However, we do not believe this imposes significant restrictions.

2.1 Some background

Computations are encoded as a DAG where operations are represented by nodes and dependencies between operations are represented by edges. Operations are usually stateless, i.e. they provide the same output given the same input (see *Writing new operations using pf.Operation* for an exception). All state information is provided by the `context`, which is a mapping from nodes to values.

When Pythonflow evaluates an operation, it will check whether the current `context` provides a value for the operation and return it immediately if possible. If the current `context` does not provide a value for the operation, Pythonflow will evaluate the dependencies of the operation, evaluate the operation of interest, store the computed value in the context, and return the value.

2.2 Referring to operations

Operations can be referred to using their python instances or their unique `name` attribute. By default, `name` is set to a random but unique identifier. The name of an operation can be specified when they are created, e.g.

```
with pf.Graph() as graph:
    a = pf.constant(4)
    b = pf.constant(38)
    x = pf.add(a, b, name='my_addition')
```

```
>>> x.name
'my_addition'
```

Or the name of an operation can be changed after it has been created (as long as the name is unique within each graph), e.g.

```
with pf.Graph() as graph:
    a = pf.constant(4)
    b = pf.constant(38)
    x = a + b
    x.name = 'my_addition'
```

Finally, you can change an operation's name using its `set_name` method, e.g.

```
with pf.Graph() as graph:
    a = pf.constant(4)
    b = pf.constant(38)
    x = (a + b).set_name('my_addition')
```

Once a name has been set, an operation can be evaluated like so

```
>>> graph('my_addition')
42
```

Pythonflow will enforce that names are indeed unique.

```
with pf.Graph() as graph:
    a = pf.constant(4, name='constant1')
    b = pf.constant(38, name='constant1')
```

```
Traceback (most recent call last):
ValueError: duplicate name 'constant1'
```

2.3 Providing inputs

Inputs for the dataflow graph can be provided using placeholders.

```
with pf.Graph() as graph:
    a = pf.placeholder(name='first_input')
    b = pf.constant(4)
    x = a + b
```

```
>>> graph(x, {a: 5})
9
```

```
>>> graph(x, {'first_input': 8})
12
```

```
>>> graph(x, first_input=7)
11
```

The latter two options are only available if the operation has been given a sensible name. Pythonflow will make sure that you do not provide inconsistent inputs:

```
>>> graph(x, {a: 5}, first_input=7)
Traceback (most recent call last):
ValueError: duplicate value for operation '<pf.placeholder 'first_input'>'
```

And that all necessary placeholders have been specified:

```
>>> graph(x)
Traceback (most recent call last):
ValueError: missing value for placeholder 'first_input'
```

2.4 Handling sequences

Unfortunately, Pythonflow does not support list comprehensions but the same results can be achieved using `pf.map_`, `pf.list_`, `pf.tuple_`, `pf.zip_`, `pf.sum_`, `pf.filter_` and other operations. Suppose we want to find the surnames of all artists whose first name begins with an `A`.

```
with pf.Graph() as graph:
    artists = pf.placeholder(name='artists')
    filtered = pf.filter_(lambda artist: artist['first'].startswith('A'), artists)
    surnames = pf.map_(lambda artist: artist['last'], filtered)
    # Convert to a list to evaluate the map call
    surnames = pf.list_(surnames)
```

```
>>> graph(surnames, artists=[
...     {
...         'first': 'Ariana',
...         'last': 'Grande'
...     },
...     {
...         'first': 'Justin',
...         'last': 'Bieber'
```

(continues on next page)

(continued from previous page)

```
...     }
... ])
['Grande']
```

2.5 Adding your own operations

Sometimes the operations that come out of the box aren't enough for your needs and you want to build something more sophisticated. There are three different options for adding new operations and we will cover each in turn.

2.5.1 Turning functions into operations

You can use the `pf.func_op` class to create an operation from a *callable*. The syntax is identical to `partial` functions except that the arguments are operations rather than values.

```
import random
random.seed(1)

with pf.Graph() as graph:
    uniform = pf.func_op(random.uniform, 0, 1)
    scaled_uniform = 10 * uniform
```

```
>>> graph([uniform, scaled_uniform])
(0.13436424411240122, 1.3436424411240122)
```

The example above not only shows how to use existing functions as operations but also illustrates that each operation is evaluated at most once when you call `graph`. Consequently, any computationally expensive operations are automatically cached.

2.5.2 Writing new operations using decorators

If you only ever intend to use a callable as an operation, you can implement the operation using a decorator:

```
@pf.opmethod(length=2)
def split_in_two(x):
    num = len(x)
    return x[:num // 2], x[num // 2:]


with pf.Graph() as graph:
    x = pf.placeholder()
    y, z = split_in_two(x)
```

```
>>> graph([y, z], {x: 'Hello World!'})
('Hello ', 'World!')
```

You may use the `opmethod` decorator with or without parameters. Specifying the `length` parameter enables unpacking of operations as illustrated above. However, this means that your operations must not have a parameter called *length* that you set using a keyword argument (positional arguments are fine). If you are wrapping an existing method that takes a `length` argument in a `func_op`, use a `lambda` function to rename the parameter like so.

```
def existing_function_you_cannot_change(length):
    return 'a' * length

with pf.Graph() as graph:
    length = pf.placeholder()
    # Rename the keyword argument using a lambda function
    y = pf.func_op(lambda length_: existing_function_you_cannot_change(length_),␣
↪length_=length)
    # Positional arguments don't cause any trouble
    z = pf.func_op(existing_function_you_cannot_change, length)
```

```
>>> graph([y, z], {length: 3})
('aaa', 'aaa')
```

2.5.3 Writing new operations using `pf.Operation`

If you want to create stateful operations, you need to dig a bit deeper into pythonflow. For example, stateful operations may be useful when you need to access a database but don't want to open a new connection every time you send a request. Stateful operations are implemented by inheriting from *pf.Operation* and implementing the `_evaluate` method like so.

```
import sqlite3


class SqliteOperation(pf.Operation):
    def __init__(self, database, query):
        # Pass on the query as an operation
        super(SqliteOperation, self).__init__(query)
        # Open a new database connection
        self.database = database
        self.connection = sqlite3.connect(self.database)

    def _evaluate(self, query):
        # The `_evaluate` method takes the same arguments as the `__init__` method
        # of the super class. Whereas the `__init__` method of the superclass receives
        # operations as arguments, the `__call__` method receives the evaluated
        # operations
        return self.connection.execute(query)


with pf.Graph() as graph:
    query = pf.placeholder(name='query')
    response = SqliteOperation(':memory:', query)
```

```
>>> graph(response, query='CREATE TABLE Companies (name VARCHAR)')
<sqlite3.Cursor object at ...>
>>> graph(response, query="INSERT INTO Companies (name) VALUES ('Spotify')")
<sqlite3.Cursor object at ...>
>>> graph(response, query="SELECT * FROM Companies").fetchall()
[('Spotify',)]
```

2.6 Conditional operations

Sometimes you may want to evaluate different parts of the DAG depending on a condition. For example, you may want to apply the same operations to data but switch between training and validation data like so.

```
with pf.Graph() as graph:
    training_data = pf.placeholder("training")
    validation_data = pf.placeholder("validation")
    condition = pf.placeholder("condition")
    data = pf.conditional(condition, training_data, validation_data)
```

```
>>> graph(data, condition=True, training=4)
4
```

Note that the `pf.conditional` operation only evaluates the part of the DAG it requires to return a value. If it evaluated the entire graph, the evaluation above would have raised a `ValueError` because we did not provide a value for the placeholder `validation_data`.

2.7 Try-except-finally operations

The `try_` operation allows you to evaluate an operation and fall back to alternatives if the operation fails. For example, you may want to handle divisions by zero like so.

```
with pf.Graph() as graph:
    a = pf.placeholder('a')
    b = pf.placeholder('b')
    c = pf.try_(a / b, [(ZeroDivisionError, "check your inputs")])
```

```
>>> graph(c, a=3, b=2)
1.5
>>> graph(c, a=3, b=0)
'check your inputs'
```

You can also use the `finally_` argument to ensure an operation is evaluated irrespective of whether another operation succeeds or fails.

2.8 Explicitly controlling dependencies

Pythonflow automatically determines the operations it needs to evaluate to return the desired output. But sometimes it is desirable to explicitly specify operations that should be evaluated. For example, you may want to print a value for debugging purposes like so.

```
with pf.Graph() as graph:
    x = pf.placeholder('x')
    y = pf.mul(2, x, dependencies=[pf.print_(pf.str_format("placeholder value: {}",␣
↪x))])
```

```
>>> graph(y, x=4)
placeholder value: 4
8
```

You may also use the context manager `control_dependencies` to specifiy explicit dependencies like so.

```
with pf.Graph() as graph:
    x = pf.placeholder('x')
    with pf.control_dependencies([pf.print_(pf.str_format("placeholder value: {}",␣
↪x))]):
        y = 2 * x
```

```
>>> graph(y, x=9)
placeholder value: 9
18
```

2.9 Assertions

When you're developing your graphs, you probably want to make sure that everything is behaving as you expect. You can check that values conform to your expectations like so.

```
with pf.Graph() as graph:
    mass = pf.placeholder('mass')
    height = pf.placeholder('height')
    assertions = [
        pf.assert_(mass > 0, "mass must be positive but got %f", mass),
        pf.assert_(height > 0, "height must be positive but got %f", height)
    ]
    with pf.control_dependencies(assertions):
        bmi = mass / height ** 2
```

```
>>> graph(bmi, mass=72, height=-1.8)
Traceback (most recent call last):
AssertionError: height must be positive but got -1.800000
```

To make the definition of graphs less verbose, you can also specify the return value of an assertion should it succeed using the *value* keyword argument like so.

```
with pf.Graph() as graph:
    mass = pf.placeholder('mass')
    height = pf.placeholder('height')
    mass = pf.assert_(mass > 0, "mass must be positive but got %f", mass, value=mass)
    height = pf.assert_(height > 0, "height must be positive but got %f", height,␣
↪value=height)
    bmi = mass / height ** 2
```

```
>>> graph(bmi, mass=72, height=-1.8)
Traceback (most recent call last):
AssertionError: height must be positive but got -1.800000
```

2.10 Profiling and callbacks

If your graph doesn't perform as well as you would like, you can gain some insight into where it's spending its time by attaching a profiler. For example, the following graph loads an image and applies a few transformations.

```
import pythonflow as pf


with pf.Graph() as graph:
    # Only load the libraries when necessary
    imageio = pf.import_('imageio')
    ndimage = pf.import_('scipy.ndimage')
    np = pf.import_('numpy')

    filename = pf.placeholder('filename')
    image = (imageio.imread(filename).set_name('imread')[..., :3] / 255.0).set_name(
↪'image')
    noise_scale = pf.constant(.25, name='noise_scale')
    noise = (1 - np.random.uniform(0, noise_scale, image.shape)).set_name('noise')
    noisy_image = (image * noise).set_name('noisy_image')
    angle = np.random.uniform(-45, 45)
    rotated_image = ndimage.rotate(noisy_image, angle, reshape=False).set_name(
↪'rotated_image')
```

```
profiler = pf.Profiler()
graph('rotated_image', filename='docs/spotify.png', callback=profiler)
```

Printing the profiler shows the ten most expensive operations. More detailed information can be retrieved by accessing the `times` attribute or calling `get_slow_operations`.

```
>>> print(profiler)
<pf.func_op 'rotated_image' target=<function call at 0x7fa6fff3abf8> args=<3 items>␣
↪kwargs=<1 items>>: 0.014486551284790039
<pf.func_op 'imread' target=<function call at 0x7fa6fff3abf8> args=<2 items> kwargs=
↪<0 items>>: 0.003396749496459961
<pf.func_op '214e8b1e1db94dfa9840d9cc4e510c25' target=<function call at␣
↪0x7fa6fff3abf8> args=<4 items> kwargs=<0 items>>: 0.0013699531555175781
<pf.func_op 'image' target=<built-in function truediv> args=<2 items> kwargs=<0 items>
↪>: 0.0005080699920654297
<pf.func_op 'noisy_image' target=<built-in function mul> args=<2 items> kwargs=<0␣
↪items>>: 0.00014448165893554688
<pf.func_op 'noise' target=<built-in function sub> args=<2 items> kwargs=<0 items>>:␣
↪0.000125885009765625
<pf.func_op '2acc029ccdb34007a826a3af3230a483' target=<function import_module at␣
↪0x7fa71a86f488> args=<1 items> kwargs=<0 items>>: 3.838539123535156e-05
<pf.func_op '5b342c50ea0c48d0b136cb6d93fdc579' target=<function import_module at␣
↪0x7fa71a86f488> args=<1 items> kwargs=<0 items>>: 2.8133392333984375e-05
<pf.func_op '583f053d792245c4bad181e2430eb8d2' target=<built-in function getitem>␣
↪args=<2 items> kwargs=<0 items>>: 2.0265579223632812e-05
<pf.func_op '9f39f2b5e0d74fe695099cd78d8ecd11' target=<function import_module at␣
↪0x7fa71a86f488> args=<1 items> kwargs=<0 items>>: 1.9788742065429688e-05
```

Rotating and reading the image from disk are the two most expensive operations.

The profiler is implemented as a callback that is passed to the graph when fetches are evaluated. Callbacks are context managers whose context is entered before an operation is evaluated and exited after the operation has been evaluated. Callbacks must accept exactly two arguments: the operation under consideration and the context. Here is an example that prints some information about the evaluation of the fetches.

```
import contextlib

```

(continues on next page)

(continued from previous page)

```
@contextlib.contextmanager
def print_summary(operation, context):
    print("About to evaluate '%s', %d values in the context." % (operation.name,␣
↪len(context)))
    yield
    print("Evaluated '%s', %d values in the context." % (operation.name,␣
↪len(context)))
```

```
>>> graph('rotated_image', filename='docs/spotify.png', callback=print_summary)
About to evaluate '2acc029ccdb34007a826a3af3230a483', 1 values in the context.
Evaluated '2acc029ccdb34007a826a3af3230a483', 2 values in the context.
About to evaluate '60f266f8dd69441eb25dea412a92dbb0', 2 values in the context.
Evaluated '60f266f8dd69441eb25dea412a92dbb0', 3 values in the context.
About to evaluate '9f39f2b5e0d74fe695099cd78d8ecd11', 3 values in the context.
Evaluated '9f39f2b5e0d74fe695099cd78d8ecd11', 4 values in the context.
About to evaluate '7e6579c9d3784500a1dd555e3ea81bde', 4 values in the context.
Evaluated '7e6579c9d3784500a1dd555e3ea81bde', 5 values in the context.
About to evaluate 'imread', 5 values in the context.
Evaluated 'imread', 6 values in the context.
About to evaluate '583f053d792245c4bad181e2430eb8d2', 6 values in the context.
Evaluated '583f053d792245c4bad181e2430eb8d2', 7 values in the context.
About to evaluate 'image', 7 values in the context.
Evaluated 'image', 8 values in the context.
About to evaluate '5b342c50ea0c48d0b136cb6d93fdc579', 8 values in the context.
Evaluated '5b342c50ea0c48d0b136cb6d93fdc579', 9 values in the context.
About to evaluate '748ad66b9338417594015a71486b324f', 9 values in the context.
Evaluated '748ad66b9338417594015a71486b324f', 10 values in the context.
About to evaluate '240dcda966e34cfb957d4b91e4e31a4f', 10 values in the context.
Evaluated '240dcda966e34cfb957d4b91e4e31a4f', 11 values in the context.
About to evaluate 'noise_scale', 11 values in the context.
Evaluated 'noise_scale', 12 values in the context.
About to evaluate 'c44468bd3aac44ddb8b3e8bd2b9eb832', 12 values in the context.
Evaluated 'c44468bd3aac44ddb8b3e8bd2b9eb832', 13 values in the context.
About to evaluate '214e8b1e1db94dfa9840d9cc4e510c25', 13 values in the context.
Evaluated '214e8b1e1db94dfa9840d9cc4e510c25', 14 values in the context.
About to evaluate 'noise', 14 values in the context.
Evaluated 'noise', 15 values in the context.
About to evaluate 'noisy_image', 15 values in the context.
Evaluated 'noisy_image', 16 values in the context.
About to evaluate '08bd356d6d464837a6a367c34591c961', 16 values in the context.
Evaluated '08bd356d6d464837a6a367c34591c961', 17 values in the context.
About to evaluate '8fa2493c78174b2f9d9849afa4e7e1f5', 17 values in the context.
Evaluated '8fa2493c78174b2f9d9849afa4e7e1f5', 18 values in the context.
About to evaluate '6c1d0576cadc418b8f037455fec5c749', 18 values in the context.
Evaluated '6c1d0576cadc418b8f037455fec5c749', 19 values in the context.
About to evaluate 'rotated_image', 19 values in the context.
Evaluated 'rotated_image', 20 values in the context.
```

CHAPTER 3

Distributed data preparation

Training data for machine learning models is often transformed before the learning process starts. But sometimes it is more convenient to generate the training data online rather than precomputing it: Maybe the data we want to present to the model depends on the current state of the model, or maybe we want to experiment with different data preparation techniques without having to rerun pipelines. This document is concerned with helping you speed up the data preparation by distributing it across multiple cores or machines.

Data preprocessing can generally be expressed as `sink = map(transformation, source)`, where `source` is a generator of input data, `transformation` is the transformation to be applied, and `sink` is an iterable over the transformed data. We use ZeroMQ to distribute the data processing using a load-balancing message broker illustrated below. Each task (also known as a client) sends one or more messages to the broker from a `source`. The broker distributes the messages amongst a set of workers that apply the desired `transformation`. Finally, the broker collects the results and forwards them to the relevant task which act as a `sink`.

3.1 Building an image transformation graph

For example, you may want to speed up the process of applying a transformation to an image using the following graph.

```python
import pythonflow as pf


with pf.Graph() as graph:
    # Only load the libraries when necessary
    imageio = pf.import_('imageio')
    ndimage = pf.import_('scipy.ndimage')
    np = pf.import_('numpy')

    filename = pf.placeholder('filename')
    image = (imageio.imread(filename).set_name('imread')[..., :3] / 255.0).set_name(
↪'image')
    noise_scale = pf.constant(.25, name='noise_scale')
    noise = (1 - np.random.uniform(0, noise_scale, image.shape)).set_name('noise')
```

(continues on next page)

(continued from previous page)

```
    noisy_image = (image * noise).set_name('noisy_image')
    angle = np.random.uniform(-45, 45)
    rotated_image = ndimage.rotate(noisy_image, angle, reshape=False).set_name(
↪'rotated_image')
```

Let's run the default pipeline using the Spotify logo.

```
context = {'filename': 'docs/spotify.png'}
graph('rotated_image', context)
```

Because the context keeps track of all computation steps, we have access to the original image, the noisy image, and the rotated image:

3.2 Distributing the workload

Pythonflow provides a straightforward interface to turn your graph into a processor for distributed data preparation:

```
from pythonflow import pfmq

backend_address = 'tcp://address-that-workers-should-connect-to'
worker = pfmq.Worker.from_graph(graph, backend_address)
worker.run()
```

Running the processors is up to you and you can use your favourite framework such as ipyparallel or Foreman.

3.3 Consuming the data

Once you have started one or more processors, you can create a message broker to facilitate communication between tasks and workers.

```
broker = pfmq.MessageBroker(backend_address)
broker.run_async()

request = {
    'fetches': 'rotated_image',
    'context': {'filename': 'docs/spotify.png'}
}
rotated_image = broker.apply(request)
```

The call to `run_async` starts the message broker in a background thread. To avoid serialization overhead, only the explicitly requested `fetches` are sent over the wire and the `context` is not updated as in the example above.

Calling the consumer directly is useful for debugging, but in most applications you probably want to process more than one example as illustrated below.

```
iterable = broker.imap(requests)
```

Using `imap` rather than using the built-in `map` applied to `broker.apply` has significant performance benefits: the message broker will dispatch as many messages as there are connected workers. Using the built-in `map` will only use one worker at a given time.

Note: By default, the consumer and processors will use `pickle` to (de)serialize all messages. You may want to consider your own serialization format or use msgpack.

Note: Pythonflow does not resend lost messages and your program will not recover if a message is lost.

CHAPTER 4

pythonflow package

exception pythonflow.core.**EvaluationError**

Bases: RuntimeError

Failed to evaluate an operation.

class pythonflow.core.**Graph**

Bases: object

Data flow graph constituting a directed acyclic graph of operations.

apply(*fetches*, *context=None*, ***, *callback=None*, ***kwargs*)

Evaluate one or more operations given a context.

Note: This function modifies the context in place. Use `context=context.copy()` to avoid the context being modified.

Parameters

- **fetches** (*list[str or Operation] or str or Operation*) – One or more *Operation* instances or names to evaluate.
- **context** (*dict or None*) – Context in which to evaluate the operations.
- **callback** (*callable or None*) – Callback to be evaluated when an operation is evaluated.
- **kwargs** (*dict*) – Additional context information keyed by variable name.

Returns **values** – Output of the operations given the context.

Return type tuple[object]

Raises

- ValueError – If *fetches* is not an *Operation* instance, operation name, or a sequence thereof.

- `ValueError` – If *context* is not a mapping.

static get_active_graph(*graph=None*)

Obtain the currently active graph instance by returning the explicitly given graph or using the default graph.

Parameters **graph** (`Graph or None`) – Graph to return or *None* to use the default graph.

Raises `ValueError` – If no *Graph* instance can be obtained.

normalize_context(*context*, ***kwargs*)

Normalize a context by replacing all operation names with operation instances.

Note: This function modifies the context in place. Use `context=context.copy()` to avoid the context being modified.

Parameters

- **context** (`dict[Operation or str, object]`) – Context whose keys are operation instances or names.
- **kwargs** (`dict[str, object]`) – Additional context information keyed by variable name.

Returns **normalized_context** – Normalized context whose keys are operation instances.

Return type dict[*Operation*, object]

Raises

- `ValueError` – If the context specifies more than one value for any operation.
- `ValueError` – If *context* is not a mapping.

normalize_operation(*operation*)

Normalize an operation by resolving its name if necessary.

Parameters **operation** (`Operation or str`) – Operation instance or name of an operation.

Returns **normalized_operation** – Operation instance.

Return type *Operation*

Raises

- `ValueError` – If *operation* is not an *Operation* instance or an operation name.
- `RuntimeError` – If *operation* is an *Operation* instance but does not belong to this graph.
- `KeyError` – If *operation* is an operation name that does not match any operation of this graph.

class pythonflow.core.**Operation**(**args*, *length=None*, *graph=None*, *name=None*, *dependencies=None*, ***kwargs*)

Bases: `object`

Base class for operations.

Parameters

- **args** (`tuple`) – Positional arguments passed to the *_evaluate* method.
- **kwargs** (`dict`) – Keyword arguments passed to the *_evaluate* method.

- **length** (*int or None*) – Optional number of values returned by the operation. The length only needs to be specified if the operation should support iterable [unpacking](https://www.python.org/dev/peps/pep-3132/).
- **graph** (Graph *or None*) – Data flow graph for this operation or *None* to use the default graph.
- **name** (*str or None*) – Name of the operation or *None* to use a random, unique identifier.
- **dependencies** (*list*) – Explicit sequence of operations to evaluate before evaluating this operation.

evaluate (*context*, *callback=None*)
Evaluate the operation given a context.

Parameters

- **context** (*dict*) – Normalised context in which to evaluate the operation.
- **callback** (*callable or None*) – Callback to be evaluated when an operation is evaluated.

Returns **value** – Output of the operation given the context.

Return type object

evaluate_dependencies (*context*, *callback=None*)
Evaluate the dependencies of this operation and discard the values.

Parameters

- **context** (*dict*) – Normalised context in which to evaluate the operation.
- **callback** (*callable or None*) – Callback to be evaluated when an operation is evaluated.

classmethod evaluate_operation (*operation*, *context*, ***kwargs*)
Evaluate an operation or constant given a context.

name
Unique name of the operation

Type str

set_name (*name*)
Set the name of the operation and update the graph.

Parameters **value** (*str*) – Unique name of the operation.

Returns **self** – This operation.

Return type *Operation*

Raises

- ValueError – If an operation with *value* already exists in the associated graph.
- KeyError – If the current name of the operation cannot be found in the associated graph.

pythonflow.core.**call** (*func*, **args*, ***kwargs*)
Call *func* with positional arguments *args* and keyword arguments *kwargs*.

Parameters

- **func** (*callable*) – Function to call when the operation is executed.
- **args** (*list*) – Sequence of positional arguments passed to *func*.

- **kwargs** (*dict*) – Mapping of keyword arguments passed to *func*.

`pythonflow.core.`**`control_dependencies`**(*dependencies*, *graph=None*)
Ensure that all *dependencies* are executed before any operations in this scope.

Parameters **dependencies** (*list*) – Sequence of operations to be evaluted before evaluating any operations defined in this scope.

class `pythonflow.core.`**`func_op`**(*target*, **args*, ***kwargs*)
Bases: `pythonflow.core.Operation`

Operation wrapper for stateless functions.

Parameters

- **target** (*callable*) – function to evaluate the operation
- **args** (*tuple*) – positional arguments passed to the target
- **kwargs** (*dict*) – keyword arguments passed to the target

`pythonflow.core.`**`opmethod`**(*target=None*, ***kwargs*)
Decorator for creating operations from functions.

`pythonflow.operations.`**`assert_`**(*condition*, *message=None*, **args*, *value=None*)
Return *value* if the *condition* is satisfied and raise an *AssertionError* with the specified *message* and *args* if not.

`pythonflow.operations.`**`cache`**(*operation*, *get*, *put*, *key=None*)
Cache the values of *operation*.

Parameters

- **operation** (`Operation`) – Operation to cache.
- **get** (*callable(object)*) – Callable to retrieve an item from the cache. Should throw *KeyError* or *FileNotFoundError* if the item is not in the cache.
- **put** (*callable(object, object)*) – Callable that adds an item to the cache. The first argument is the key, the seconde the value.
- **key** (`Operation`) – Key for looking up an item in the cache. Defaults to a simple *hash* of the arguments of *operation*.

Returns **cached_operation** – Cached operation.

Return type *Operation*

`pythonflow.operations.`**`cache_file`**(*operation*, *filename_template*, *load=None*, *dump=None*, *key=None*)
Cache the values of *operation* in a file.

Parameters

- **operation** (`Operation`) – Operation to cache.
- **filename_template** (*str*) – Template for the filename taking a single *key* parameter.
- **load** (*callable(str)*) – Callable to retrieve an item from a given file. Should throw *FileNotFoundError* if the file does not exist.
- **dump** (*callable(object, str)*) – Callable to save the item to a file. The order of arguments differs from the *put* argument of *cache* to be compatible with *pickle.dump*, *numpy.save*, etc.
- **key** (`Operation`) – Key for looking up an item in the cache. Defaults to a simple *hash* of the arguments of *operation*.

Returns **cached_operation** – Cached operation.

Return type *Operation*

class `pythonflow.operations.`**`conditional`**(*predicate*, *x*, *y=None*, ***, *length=None*, *name=None*, *dependencies=None*)

Bases: `pythonflow.core.Operation`

Return *x* if *predicate* is *True* and *y* otherwise.

Note: The conditional operation will only execute one branch of the computation graph depending on *predicate*.

`evaluate`(*context*, *callback=None*)

Evaluate the operation given a context.

Parameters

- **`context`** (`dict`) – Normalised context in which to evaluate the operation.
- **`callback`** (`callable or None`) – Callback to be evaluated when an operation is evaluated.

Returns **value** – Output of the operation given the context.

Return type object

`pythonflow.operations.`**`constant`**(*value*)

Operation returning the input value.

`pythonflow.operations.`**`identity`**(*value*)

Operation returning the input value.

class `pythonflow.operations.`**`lazy_constant`**(*target*, ***kwargs*)

Bases: `pythonflow.core.Operation`

Operation that returns the output of *target* lazily.

Parameters

- **`target`** (`callable`) – Function to evaluate when the operation is evaluated.
- **`kwargs`** (`dict`) – Keyword arguments passed to the constructor of *Operation*.

class `pythonflow.operations.`**`placeholder`**(*name=None*, ***kwargs*)

Bases: `pythonflow.core.Operation`

Placeholder that needs to be given in the context to be evaluated.

`pythonflow.operations.`**`str_format`**(*format_string*, **args*, ***kwargs*)

Use python's advanced string formatting to convert the format string and arguments.

References

https://www.python.org/dev/peps/pep-3101/

class `pythonflow.operations.`**`try_`**(*operation*, *except_=None*, *finally_=None*, ***kwargs*)

Bases: `pythonflow.core.Operation`

Try to evaluate *operation*, fall back to alternative operations in *except_*, and ensure that *finally_* is evaluated.

Note: The alternative operations will only be executed if the target operation fails.

Parameters

- **`operation`** (`Operation`) – Operation to evaluate.
- **`except`** (*`list[(type, Operation)]`*) – List of exception types and corresponding operation to evaluate if it occurs.
- **`finally`** (`Operation`) – Operation to evaluate irrespective of whether *operation* fails.

`evaluate`(*context*, *callback=None*)

Evaluate the operation given a context.

Parameters

- **`context`** (*`dict`*) – Normalised context in which to evaluate the operation.
- **`callback`** (*`callable or None`*) – Callback to be evaluated when an operation is evaluated.

Returns **value** – Output of the operation given the context.

Return type object

class `pythonflow.util.`**`Profiler`**

Bases: `object`

Callback for profiling computational graphs.

`times`

Mapping from operations to execution times.

Type dict[*Operation*, float]

`get_slow_operations`(*num_operations=None*)

Get the slowest operations.

Parameters **`num_operations`** (*`int or None`*) – Maximum number of operations to return or *None*

Returns **times** – Mapping of execution times keyed by operations.

Return type collections.OrderedDict

class `pythonflow.util.`**`batch_iterable`**(*iterable*, *batch_size*, *transpose=False*)

Bases: `object`

Split an iterable into batches of a specified size.

Parameters

- **`iterable`** (*`iterable`*) – Iterable to split into batches.
- **`batch_size`** (*`int`*) – Size of each batch.
- **`transpose`** (*`bool`*) – Whether to transpose each batch.

`pythonflow.util.`**`deprecated`**(*func*)

Mark a callable as deprecated.

class `pythonflow.util.`**`lazy_import`**(*module*)

Bases: `object`

Lazily import the given module.

Parameters **`module`** (*`str`*) – Name of the module to import

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index