pypey

Release 3.0.2

Jose Llarena

Jan 16, 2022

CONTENTS

Pypey is a library for concisely composing data transformation primitives. It support lazily evaluated collection pipelines with standard operations like map, reduce , filter and several others others. It is fashioned after libraries like Java Streams , Immutable.js and C++ Streams. It has been inspired by, and leans on, the excellent itertools and more-itertools

CHAPTER ONE

MOTIVATION

Many operations on data, like reading lines from a file, filtering or aggregating items, appear repeatedly across many domains and so they benefit from encapsulating to avoid duplication and enable code reuse. This encapsulation also has a second benefit: the abstracted operations now map 1-to-1 to the high-level description, ie, the intent of the coder. A third advantage is that it allows the different operations in a pipeline to be disentangled from each other.

Let's illustrate all this with an example: a frequent data processing routine where you need to build a word-to-id dictionary from a file containing words of text. The high-level recipe would be something like this:

```
1. read lines from file
2. split lines into words
3. keep only unique words across all lines
4. assign a unique number id to each word
5. put words and their ids in a dictionary
```

A typical implementation using only python built-ins would look like this:

```python
with open('text.txt') as file:          # 1. open file for reading
    idx = 0                             # 2. make id counter
    word_to_id = {}                     # 3. make empty dict

    for line in file:                   # 4. loop through lines
        for word in line.split():       # 5. split line and loop through words
            word = word.rstrip()        # 6. strip line terminator
            if word not in word_to_id:  # 7. check to see if it's in dictionary
                word_to_id[word] = idx  # 8. insert word with a new id if it's not
                idx += 1                # 9. update id counter
```

Notice how there are steps `2.` , `3.` , `6.` and `9.` do not correspond to anything in the high level recipe. Notice also, how operations `4.` to `9.` interleave with each other, happening once per loop iteration. Let's see how this could be implemented with `itertools`:

```python
from itertools import chain, count

with open('text.txt') as file:              # 1. open file for reading

    lines = file.readlines()                # 2. read lines from file
    stripped = map(str.rstrip, lines)       # 3. strip line terminator
    words = map(str.split, stripped)        # 4. split lines into words
    all_words = chain.from_iterable(words)  # 5. concatenate all lines
    unique = set(all_words)                 # 6. keep only unique words across all lines
    words_ids = zip(unique, count())        # 7. assign a unique id to each word
    word_to_id = dict(words_ids)            # 8. put in dictionary
```

Now the steps match the original intent better because they operate at higher level, ie, they work at the level of whole collections of items, and do not concern themselves with the data and syntactic structures needed when the algorithm is specified at the level of the individual items, as in the first implementation. The next implementation is more concise and leverages the ability to pipe the collections together:

```
from itertools import chain, count

with open('text.txt') as file:  # 1.
     # 2. + 3. + 4. + 5. + 6. + 7. + 8.
     word_to_id = dict(zip(set(chain.from_iterable(map(str.split, map(str.rstrip, file.
↪readlines())))), count()))
```

However, the sequence of steps is now laid out in reverse order or “inside-out”. With Pypey, the code is still concise and the steps always flow right:

```
from pypey import pype

word_to_id = (pype.file('text.txt') # 1. read lines from file, strip line terminator by␣
↪default
              .map(str.split)       # 2. split lines into words
              .flat()               # 3. concatenate all lines (by "flattening" them)
              .uniq()               # 4. keep only unique words across all lines
              .enum(swap=True)      # 5. assign a unique id to each word
              .to(dict))            # 6. put in a dictionary
```

This implementation matches the original intent best and removes the need for the coder to write boiler-plate that is not domain-specific. A more terse implementation helps when using the Python interpreter’s interactive mode (REPL):

```
>>> from pypey import pype
>>> # 1. + 2. + 3. + 4. + 5. + 6.
>>> word_to_id = pype.file('text.txt').map(str.split).flat().uniq().enum(swap=True).
↪to(dict)
```

1.1 Lazy and Deferred Evaluation

Both `itertools`-‘s and Pypey’s implementation would incur a performance penalty if each step created an intermediate collection. However by piping through lazy collections, ie, those that are evaluated incrementally only one item at a time as they are iterated through (based on generators), the performance is similar to a loop-based implementation. Furthermore, just as the loop-based approach, items are only read one at a time into memory, avoiding unnecessary allocation.

Not all operations can be implemented lazily, for instance, sorting is necessarily “eager” as it entails traversing the whole collection before being able to retrieve the first sorted item. Pypey still makes these eager operations deferred to allow delaying the consumption of the lazy collection until it’s actually needed:

```
>>> p = pype(['a', 'fun', 'day']).sort()
>>> p
<pypey.pype.Pype object at 0x7f58edaf4970>
>>> list(p)
['a', 'day', 'fun']
```

1.2 Argument Unpacking

PEP 3113 removed Python 2's ability to unpack function arguments from Python 3. This made using higher-order functions (functions taking or returning other functions) harder when applied to iterable items in a collection, especially so when lambdas are passed in, as it's impossible to use unpacking assignments in them. Pypey brings back a limited form of argument unpacking that works only at the top level of nesting. For instance:

```
>>> pype.dict({'a':1, 'fun':2, 'day':3}).map(lambda kv: (kv[0], kv[1] + 1)).to(list)
[('a', 2), ('fun', 3), ('day', 4)]
```

can also be written more clearly as:

```
>>> pype.dict({'a':1, 'fun':2, 'day':3}).map(lambda k, v: (k, v + 1)).to(list)
[('a', 2), ('fun', 3), ('day', 4)]
```

CHAPTER

TWO

GETTING STARTED

To get started, install the library with pip:

```
pip install pypey
```

Then use as:

```
>>> from pypey import pype
>>> pype(range(-2, 3)).map(abs).print()
2
1
0
1
2
<pypey.pype.Pype object at 0x7f56401e0f40>
```

To run tests install `pytest`:

```
pip install pytest
```

then run:

```
pytest
```

CHAPTER

THREE

RELATED LIBRARIES

Pypey is similar to itertools and more-itertools but takes an object-based approach instead, with method-chaining as the main pipe-building mechanism, instead of function composition. This allows pipes to always flow right (or down, if properly formatted) which is arguably a more intuitive ordering for coders used to Object Orientated Programming.

Pypey is perhaps most similar to python_lazy_streams as it too uses an object-based+method-chaining approach. Related is pipes, which implements function-chaining with operator overloading and decorators, allowing it to compose right like Pypey. stream.py also uses operator overloading and function chaining.

There's a number of libraries with extensive pipeline APIs, such as ReactiveX-'s function-chaining RxPy , but geared towards real-time event streams. Nvidia's Streamz is in the same space, but is object-based with method-chaining, and adds support for Pandas and cuDF. Apache Beam combines batch- and stream-processing and supports different backends like Spark, and uses function-chaining with operator overloading. Tensorflow's Dataset class is an method-chaining API focussed on loading tensors.

Riko is also an object-based+method-chaining type of API but specialises in structured text processing.

Mario is a CLI-based function-chaining API, similar to Unix Shell's pipes.

CHAPTER
FOUR

CONTENTS

4.1 API Reference

Pype's methods can be grouped according to whether they are lazy or eager vs whether they are deferred or immediate:

	Deferred	Immediate

Lazy	accum() broadcast() cat() chunk() clone() cycle() dist() do() drop() drop_while() enum() flat() flatmap() it() interleave() map() partition() pick() print() reject() select() slice() split() take() take_while() tee() to_file() uniq() window() zip() zip_with()	do()
		map()
		print()
		to_file()

Eager	divide() freqs() group_by() reverse() roundrobin() sample() shuffle() sorted() take() to_json() top() unzip()	eager()
		reduce()
		size()

Lazy methods would normally be deferred and eager methods immediate. However, if an eager method returns a pipe, Pypey defers its execution until it's iterated through, at which point, the backing `Iterable` will be consumed, if lazy.

`do()`, `map()`, `print()` and `to_file()` are intrinsically lazy but they can be eager in certain circumstances. The three side-effectful methods `do()`, `print()` and `to_file()` can be made eager by setting their `now` parameter to `True` (`to_file()`'s and `print()`'s are `True` by default) as they are often the last operations in a pipeline, when consumption of its backing `Iterable` is warranted. `do()` and `map()` consume their iterables when they are made parallel, if their `workers` parameter is set to a value larger than `0`.

`take()` is lazy when it's parameter is positive (head) and eager when it's negative (tail).

In order to stop a lazy backing `Iterable` from being consumed, `clone()` can be used to ensure that the consumption happens on the returned pipe and not the original one. `clone()` manages this by using `iteratools.tee` under the hood and it's the only method that mutates the internal state of a `Pype`

If reading the whole pipe into memory is not an issue, it's often simpler and more efficient to create a pipe with an eager backing `Iterable` (`list`, `tuple` and so on). For instance if reading from a file:

```
>>> from pypey import pype
>>> eager_pype = pype.file('text.txt').to(list, Pype)
```

or more conveniently:

```
>>> from pypey import pype
>>> eager_pype = pype.file('text.txt').eager()
```

Pype's methods can also be categorised according to what kind of pipe they return: *mappings* return a pipe of the same size as the one they are applied to; *partitions* return divisions of the original pipe; and *selections* return a subset of the pipe's items:

Mappings	accum() clone() enum() map() pick() reverse() shuffle() sort() zip_with()
Partitions	chunk() dist() divide() group_by() partition() split() unzip() window() zip()
Selections	drop() drop_while() reject() sample() select() slice() take() take_while() top() uniq()

If given an initial value, `accum()` will return a pipe with one more item than the original pipe's.

Some methods are just specialisations, or convenient versions of others:

General	Specific
`__iter__()`	`it()`
`do()`	`print()` `to_file()`
`map()`	`pick()` `zip_with()`
`sample()`	`shuffle()`
`select()`	`reject()`
`slice()`	`drop()` `take()`
`zip()`	`enum()`
`window()`	`chunk()`

Modules

4.1.1 pypes

Factory for creating pipes from different sources.

class Pyper

Bases: `object`

Factory for creating new pipes. Use *pype* instance.

```
>>> from pypey import pype
>>> list(pype([1,2,3]))
[1, 2, 3]
```

static dict(*dictionary: Mapping[pypey.func.K, pypey.func.V]*) → *pypey.pype.Pype*[Tuple[pypey.func.K, pypey.func.V]]

Returns a pipe where each item is a key-value pair in the given `Mapping`.

```
>>> from pypey import pype
list(pype.dict({'fun':1, 'day':2}))
[('fun', 1), ('day', 2)]
```

Parameters **dictionary** – the dictionary to pipe

Returns a pipe containing the dictionary's items

Raises `TypeError` if dictionary is not a `Mapping`

static file(*src: Union[AnyStr, os.PathLike, int]*, ***, *mode: str = 'r'*, *buffering: int = - 1*, *encoding: Optional[str] = 'utf8'*, *errors: Optional[str] = None*, *newline: Optional[str] = None*, *closefd: bool = True*, *opener: Optional[Callable[[...], int]] = None*, *strip: bool = True*) → *pypey.pype.Pype*[str]

Reads lines from given file into a pipe.

```
>>> from pypey import pype
>>> from os.path import join, dirname
>>> list(pype.file(join(dirname(__file__), 'unittests', 'test_file.txt')))
['line 1', 'line 2', 'line 3']
```

Parameters

- **src** – path to the file or file descriptor, as per built-in `open file` argument
- **mode** – mode as per built-in `open`, except no write modes are allowed
- **buffering** – buffering as per built-in `open`
- **encoding** – encoding as per built-in `open` except the default value is `utf8` instead of `None`
- **errors** – errors as per built-in `open`
- **newline** – newline as per built-in `open`
- **closefd** – closefd as per built-in `open`
- **opener** – opener as per built-in `open`
- **strip** – `True` if end of line should be removed from each line, `False` otherwise

Returns a pipe where each item is a line in the given file

Raises `ValueError` if `mode` has `w` (write) or `+` (append) in it

static json(*src: Union[AnyStr, os.PathLike, int]*, ***, *mode: str = 'r'*, *cls: Optional[Type] = None*, *object_hook: Optional[Callable] = None*, *parse_float: Optional[Callable] = None*, *parse_int: Optional[Callable] = None*, *parse_constant: Optional[Callable] = None*, *object_pairs_hook: Optional[Callable] = None*) → *pypey.pype.Pype*[Union[Tuple[str, Any], None, bool, float, int, str]]

Reads content in given json into a pipe.

```
>>> from pypey import pype
>>> from os.path import join, dirname
>>> dict(pype.json(join(dirname(__file__), 'unittests', 'object.json')))
{'a': 1.0, 'fun': 2.0, 'day': 3.0}
```

Parameters

- **src** – path to the file or file descriptor, as per built-in `open file` argument
- **mode** – mode as per built-in `open`, except no write modes are allowed
- **cls** – custom JSONDecoder class, as per `json.load`
- **object_hook** – object_hook `Callable`, as per `json.load`
- **parse_float** – parse_float `Callable`, as per `json.load`
- **parse_int** – parse_int `Callable`, as per `json.load`
- **parse_constant** – parse_constance `Callable`, as per `json.load`

- **object_pairs_hook** – object_pairs_hook `Callable`, as per `json.load`

Returns pype with single item if json contains a single value, several items if json contains a list and pairs of items if json contains object

pype: *pypey.pypes.Pyper* **= <pypey.pypes.Pyper object>**
Pype factory

4.1.2 pype

Main class for building streaming pipelines

class Pype(*it: Iterable[pypey.func.T]*)
Bases: `Generic[pypey.func.T]`

accum(*fn: Callable[[pypey.func.X, pypey.func.T], pypey.func.X], init: Optional[pypey.func.X] = None*) → *pypey.pype.Pype*[pypey.func.X]
Returns a pipe where each item is the result of combining a running total with the corresponding item in the original pipe:

```
>>> from pypey import pype
>>> list(pype([1, 2, 3]).accum(lambda total, n: total+n))
[1, 3, 6]
```

When an initial value is given, the resulting pipe will have one more item than the original one:

```
>>> from pypey import pype
>>> list(pype([1, 2, 3]).accum(lambda total, n: total+n, init=0))
[0, 1, 3, 6]
```

Similar to `itertools.accumulate`.

Parameters

- **init** – optional initial value to start the accumulation with
- **fn** – function where the first argument is the running total and the second the current item

Returns a pipe with accumulated items

Raises `TypeError` if `fn` is not a `Callable`

broadcast(*fn: Callable[[pypey.func.T], Iterable[pypey.func.X]]*) → *pypey.pype.Pype*[Tuple[pypey.func.T, pypey.func.X]]
Returns the flattened Cartesian product of this pipe's items and the items returned by `fn`. Conceptually similar to `numpy`-'s broadcasting.

```
>>> from pypey import pype
>>> list(pype(['a', 'fun', 'day']).broadcast(tuple).map(lambda word, char: f'
↪{word} -> {char}'))
['a -> a', 'fun -> f', 'fun -> u', 'fun -> n', 'day -> d', 'day -> a', 'day -> y
↪']
```

Parameters **fn** – function to create `Iterable` from each of this pipe's items

Returns a pipe where each item is a pair with the first element being the nth instance of this pipe's items and the second an element of `fn`-s returned `Iterable`

Raises `TypeError` if `fn` is not a `Callable`

cat(*other: Iterable[pypey.func.X]*) → *pypey.pype.Pype*[Union[pypey.func.T, pypey.func.X]]

Concatenates this pipe with the given `Iterable`.

```
>>> list(pype([1, 2, 3]).cat([4, 5, 6]))
[1, 2, 3, 4, 5, 6]
```

Parameters **other** – `Iterable` to append to this one

Returns a concatenated pipe

Raises `TypeError` if `other` is not an `Iterable`

chunk(*size: Union[int, Iterable[Optional[int]]]*) → *pypey.pype.Pype*[*pypey.pype.Pype*[pypey.func.T]]

Breaks pipe into sub-pipes with up to `size` items each:

```
>>> from pypey import pype
>>> [list(chunk) for chunk in pype([1, 2, 3, 4]).chunk(2)]
[[1, 2], [3, 4]]
```

If this pipe's size is not a multiple of `size`, the last chunk will have fewer items than `size`:

```
>>> from pypey import pype
>>> [list(chunk) for chunk in pype([1, 2, 3]).chunk(2)]
[[1, 2], [3]]
```

If `size` is larger than this pipe's size, only one chunk will be returned:

```
>>> from pypey import pype
>>> [list(chunk) for chunk in pype([1, 2, 3, 4]).chunk(5)]
[[1, 2, 3, 4]]
```

If `size` is an iterable of ints, chunks will have corresponding sizes:

```
>>> [list(chunk) for chunk in pype([1, 2, 3, 4]).chunk([1, 3])]
[[1], [2, 3, 4]]
```

If the sum of sizes is smaller than this pipe's length, the remaining items will not be returned:

```
>>> from pypey import pype
>>> list(chunk) for chunk in pype([1, 2, 3, 4]).chunk([1, 2])]
[[1], [2, 3]]
```

If the sum of sizes is larger than this pipe's length, fewer items will be returned in the chunk that overruns the pipe and further chunks will be empty:

```
>>> from pypey import pype
>>> [list(chunk) for chunk in pype([1, 2, 3, 4]).chunk([1, 2, 3, 4])]
[[1], [2, 3], [4], []]
```

The last size can be `None`, in wich case, the last chunk's will be remaining items after the one-but-last size.

```
>>> from pypey import pype
>>> [list(chunk) for chunk in pype([1, 2, 3, 4]).chunk([1, None])]
[[1], [2, 3, 4]]
```

This method tees the backing `Iterable`.

Similar to `more_itertools.ichunked` and `more_itertools.split_into`.

Parameters **`size`** – chunk size or sizes

Returns a pipe of pipes with up to *size* items each or with sizes specified by iterable of sizes

Raises `TypeError` if `size` is not an `int` or an `Iterable` of `int`-s

Raises `ValueError` if `size` is not positive or if any of the iterable of sizes is not positive

`clone``()` → *pypey.pype.Pype*[pypey.func.T]

Lazily clones this pipe. This method tees the backing `Iterable` and replaces it with a new copy.

```
>>> from pypey import pype
>>> list(pype([1, 2, 3]).clone())
[1, 2, 3]
```

Similar to `itertools.tee`.

Returns a copy of this pipe

`cycle``(`*n: Optional[int] = None*`)` → *pypey.pype.Pype*[pypey.func.T]

Returns items in pipe `n` times if `n` is not `None`:

```
>>> from pypey import pype
>>> list(pype([1, 2, 3]).cycle(2))
[1, 2, 3, 1, 2, 3]
```

else it returns infinite copies:

```
>>> from pypey import pype
>>> list(pype([1, 2, 3]).cycle().take(6))
[1, 2, 3, 1, 2, 3]
```

Similar to `itertools.cycle` with `n` = `None` and to `more_itertools.ncycles` with integer `n`.

Parameters **`n`** – number of concatenated copies or `None` for infinite copies

Returns a pipe that cycles either `n` or infinite times over the items of this one

Raises `TypeError` if `n` is neither an `int` nor `None`

Raises `ValueError` if `n` is not negative

`dist``(`*n: int*`)` → *pypey.pype.Pype*[*pypey.pype.Pype*[pypey.func.T]]

Returns a pipe with `n` items, each being smaller pipes containing this pipe's elements distributed equally amongst them:

```
>>> from pypey import pype
>>> [list(segment) for segment in pype([1, 2, 3, 4, 5, 6]).dist(2)]
[[1, 3, 5], [2, 4, 6]]
```

If this pipe's size is not evenly divisible by `n`, then the size of the returned `Iterable` items will not be identical:

```
>>> from pypey import pype
>>> [list(segment) for segment in pype([1, 2, 3, 4, 5]).dist(2)]
[[1, 3, 5], [2, 4]]
```

If this pipe's size is smaller than `n`, the last pipes in the returned pipe will be empty:

```
>>> from pypey import pype
>>> [list(segment) for segment in pype([1, 2, 3, 4, 5]).dist(7)]
[[1], [2], [3], [4], [5], [], []]
```

This method tees the backing `Iterable`.

Similar to `more_itertools.distribute`.

Parameters `n` – the number of pipes with distributed elements

Returns a pipe with this pipe's items distributed amongst the contained pipes

Raises `TypeError` if `n` is not an `int`

Raises `ValueError` if `n` is not positive

`divide`(*n: int*) → *pypey.pype.Pype*[*pypey.pype.Pype*[pypey.func.T]]

Breaks pipe into `n` sub-pipes:

```
>>> from pypey import pype
>>> [list(div) for div in pype([1, 2, 3, 4, 5, 6]).divide(2)]
[[1, 2, 3], [4, 5, 6]]
```

If this pipe's size is not a multiple of `n`, the sub-pipes' sizes will be equal except the last one, which will contain all excess items:

```
>>> from pypey import pype
>>> [list(div) for div in pype([1, 2, 3, 4, 5, 6, 7]).divide(3)]
[[1, 2], [3, 4], [5, 6, 7]]
```

If this pipe's size is smaller than `n`, the resulting pipe will contain as many single-item pipes as there are in it, followed by `n` minus this pipe's size empty pipes.

```
>>> from pypey import pype
>>> [list(div) for div in pype([1, 2, 3]).divide(4)]
[[1], [2], [3], []]
```

This method requires calculating the size of this pipe, and thus will eagerly consume the backing `Iterable` if it's lazy.

Similar to `more_itertools.divide`.

Parameters `n` – number of segments

Returns a pipe of `n` pipes

Raises `TypeError` if `n` is not an `int`

Raises `ValueError` if `n` is not positive

`do`(*fn: Callable[[pypey.func.T], Any]*, ***, *now: bool = False*, *workers: int = 0*, *chunk_size: int = 100*) → *pypey.pype.Pype*[pypey.func.T]

Produces a side effect for each item, with the given function's return value ignored. It is typically used to execute an operation that is not functionally pure such as printing to console, updating a GUI, writing to disk or sending data over a network.

```
>>> from pypey import pype
>>> p = pype(iter([1, 2, 3])).do(lambda n: print(f'{n}'))
```

(continues on next page)

(continued from previous page)

```
>>> list(p)
1
2
3
[1, 2, 3]
```

If now is set to True the side effect will take place immediately and the backing Iterable will be consumed if lazy.

```
>>> from pypey import pype
>>> p = pype(iter([1, 2, 3])).do(lambda n: print(f'{n}'), now=True)
1
2
3
>>> list(p)
[]
```

If workers is greater than 0 the side effect will be parallelised using multiprocessing if possible, or pathos if not. pathos multiprocessing implementation is slower and limited vs the built-in multiprocessing but it does allow using lambdas and local functions. When using workers, the backing Iterable is teed to avoid consumption. Using a large chunk_size can greately speed up parallelisation; it is ignored if workers is 0.

Also known as for_each, tap and sink.

Similar to more_itertools.side_effect.

Parameters

- **fn** – a function taking a possibly unpacked item
- **now** – False to defer the side effect until iteration, True to write immediately
- **workers** – number of extra processes to parallelise this method's side effect function
- **chunk_size** – size of subsequence of Iterable to be processed by workers

Returns this pipe

Raises TypeError if fn is not a Callable or workers or chunk_size are not int

Raises ValueError if workers is negative or chunk_size is non-positive

drop(*n: int*) → *pypey.pype.Pype*[pypey.func.T]

Returns this pipe but with the first or last n items missing:

```
>>> from pypey import pype
>>> list(pype([1, 2, 3, 4]).drop(2))
[3, 4]

>>> from pypey import pype
>>> list(pype([1, 2, 3, 4]).drop(-2))
[1, 2]
```

Parameters **n** – number of items to skip, positive if at the beginning of the pipe, negative at the end

Returns pipe with n dropped items

Raises `TypeError` if `n` is not an `int`

`drop_while`(*pred: Callable[[...], bool]*) → *pypey.pype.Pype*[pypey.func.T]

Drops items as long as the given predicate is `True`; afterwards, it returns every item:

```
>>> from pypey import pype
>>> list(pype([1, 2, 3, 4]).drop_while(lambda n: n < 3))
[3, 4]
```

Similar to `itertools.dropwhile`.

Parameters **`pred`** – A function taking a possibly unpacked item and returning a boolean

Returns a pipe that is a subset of this one

Raises `TypeError` if `pred` is not a `Callable`

`eager`() → *pypey.pype.Pype*[pypey.func.T]

Returns a pype with the same contents as this one but with an eager backing collection. This will trigger reading the whole backing `Iterable` into memory.

```
>>> from pypey import pype
>>> p = pype(range(-5, 5)).map(abs)
>>> p.size()
10
>>> p.size()
0
>>> p = pype(range(-5, 5)).map(abs).eager()
>>> p.size()
10
>>> p.size()
10
```

Returns this pipe, but eager

`enum`(*start: int = 0*, **, *swap: bool = False*) → *pypey.pype.Pype*[Union[Tuple[int, pypey.func.T], Tuple[pypey.func.T, int]]]

Pairs each item with an increasing integer index:

```
>>> from pypey import pype
>>> list(pype(['a', 'fun', 'day']).enum())
[(0, 'a'), (1, 'fun'), (2, 'day')]
```

`swap` = `True` will swap the index and item around:

```
>>> from pypey import pype
>>> list(pype(['a', 'fun', 'day']).enum(swap=True))
[('a', 0), ('fun', 1), ('day', 2)]
```

Similar to built-in `enumerate`.

Parameters

- **`start`** – start of the index sequence
- **`swap`** – if `True` index will be returned second, else it will be returned first

Returns a pipe the same size as this one but with each item being `tuple` of index and original item

Raises `TypeError` if `start` is not an `int`

flat() → *pypey.pype.Pype*[pypey.func.T]

Flattens iterable items into a collection of their elements:

```
>>> from pypey import pype
>>> list(pype(['a', 'fun', 'day']).flat())
['a', 'f', 'u', 'n', 'd', 'a', 'y']
```

Similar to `itertools.chain.from_iterable`

Returns a pipe with the elements of its `Iterable` items as items

Raises `TypeError` if items are not `Iterable`

flatmap(*fn: Callable[[...], pypey.func.I]*) → *pypey.pype.Pype*[pypey.func.X]

Maps `Iterable` items and then flattens the result into their elements.

Equivalent to *`Pype.map()`* followed by *`Pype.flat()`*

Parameters **fn** – function taking a possibly unpacked item and returning a value

Returns a pipe with mapped flattened items

Raises `TypeError` if items are not `Iterable` or `fn` is not a `Callable`

freqs(*total: bool = True*) → *pypey.pype.Pype*[Tuple[Union[pypey.func.T, object], int, float]]

Computes this pipe's items' absolute and relative frequencies and optionally the total:

```
>>> from pypey import pype
>>> tuple(pype('AAB').freqs())
(('A', 2, 0.6666666666666666), ('B', 1, 0.3333333333333333), (_TOTAL_, 3, 1.0))
```

If *total* is *False*, the total is left out:

```
>>> from pypey import pype
>>> tuple(pype('AAB').freqs(total=False))
(('A', 2, 0.6666666666666666), ('B', 1, 0.3333333333333333))
```

Returns a pype containing tuples with this pipe's uniques items, plus the total as the `pype.TOTAL` item, with their absolute and relative frequencies

group_by(*key: Callable[[...], pypey.func.Y]*) → *pypey.pype.Pype*[Tuple[pypey.func.Y, List[pypey.func.T]]]

Groups items according to the key returned by the `key` function:

```
>>> from pypey import pype
>>> list(pype(['a', 'fun', 'day']).group_by(len))
[(1, ['a']), (3, ['fun', 'day'])]
```

This method is eager and will consume the backing `Iterable` if it's lazy.

Similar to `itertools.groupby` except elements don't need to be sorted

Parameters **key** – function taking a possibly unpacked item and returning a grouping key

Returns a pipe made up of pairs of keys and lists of items

Raises `TypeError` if `fn` is not a `Callable`

interleave(*other: Iterable[pypey.func.X], n: int = 1, trunc: bool = True*) → *pypey.pype.Pype*[Union[pypey.func.T, pypey.func.X, Any]]

Returns a pipe where items in this pipe are interleaved with items in the given `Iterable`, in order. If either this pipe or the other `Iterable` are exhausted the interleaving stops:

```
>>> from pypey import pype
>>> list(pype(['a', 'fun', 'fun', 'day']).interleave([1, 2, 3]))
['a', 1, 'fun', 2, 'fun', 3]
```

Setting `trunc` to `True` will keep adding the left over items in the `Iterable` that hasn't been exhausted after the other one is:

```
>>> from pypey import pype
>>> list(pype(['a', 'fun', 'fun', 'day']).interleave([1, 2, 3], trunc=False))
['a', 1, 'fun', 2, 'fun', 3, 'day']
```

The number of items in this pipe's to leave between the items in the `Iterable` can be varied:

```
>>> from pypey import pype
>>> list(pype(['a', 'fun', 'fun', 'day']).interleave([1, 2, 3], n=2))
['a', 'fun', 1, 'fun', 'day', 2]
```

This operation is lazy.

A cross between `more_itertools.interleave_longest`, `more_itertools.interleave` and `more_itertools.intersperse`.

Parameters

- **other** – the `Iterable` whose items will be interleaved with this pipe's
- **n** – the number of this pipe's items to leave between each of the `Iterable`-'s ones
- **trunc** – `True` if the unexhausted `Iterable` should be truncated once the other one is

Returns A pipe with this pipe's elements and the given `Iterable`-'s in order

Raises `TypeError` if `other` is not an `Iterable` or `n` is not an ``int`

Raises `ValueError` if `n` is less than one

it() → Iterator[pypey.func.T]

Returns an `Iterator` for this pipe's items. It's a more concise version of, and functionally identical to, `Pype.__iter__()`

```
>>> from pypey import pype
>>> list(iter(pype([1, 2, 3]))) == list(pype([1, 2, 3]).it())
True
```

Returns an `Iterator` for this pipe's items

map(*fn: Callable[[...], pypey.func.Y], *other_fns: Callable[[...], pypey.func.X], workers: int = 0, chunk_size: int = 100*) → *pypey.pype.Pype*[Union[pypey.func.X, pypey.func.Y]]

Transforms this pipe's items according to the given function(s):

```
>>> from math import sqrt
>>> from pypey import pype
>>> list(pype([1, 2, 3]).map(sqrt))
[1.0, 1.4142135623730951, 1.7320508075688772]
```

If more than one function is provided, they will be chained into a single one before being applied to each item:

```
>>> from pypey import pype
>>> list(pype(['a', 'fun', 'day']).map(str.upper, reversed, list))
[['A'], ['N', 'U', 'F'], ['Y', 'A', 'D']]
```

If `workers` is greater than `0` the mapping will be parallelised using `multiprocessing` if possible or `pathos` if not. `pathos` multiprocessing implementation is slower and has different limitations than the built-in multiprocessing but it does allow using lambdas. When using workers, the backing `Iterable` is teed to avoid consumption. Using a large `chunk_size` can greately speed up parallelisation; it is ignored if `workers` is `0`.

Similar to built-in `map`.

Parameters

- **`fn`** – a function taking a possibly unpacked item and returning a value
- **`other_fns`** – other functions to be chained with `fn`, taking a possibly unpacked item and returning a value
- **`workers`** – number of extra processes to parallelise this method's mapping function(s)
- **`chunk_size`** – size of subsequence of `Iterable` to be processed by workers

Returns a pipe with this pipe's items mapped to values

Raises `TypeError` if `fn` is not a `Callable` or `other_fns` is not a `tuple` of `Callable` or if `workers` or `chunk_size` are not `int`

Raises `ValueError` if `workers` is negative or `chunk_size` is non-positive

`partition`(*pred: Callable[[...], bool]*) → Tuple[*pypey.pype.Pype*[pypey.func.T], *pypey.pype.Pype*[pypey.func.T]]

Splits this pipe's items into two pipes, according to whether the given predicate returns `True` or `False`:

```
>>> from pypey import pype
>>> [list(p) for p in pype([1, 2, 3, 4]).partition(lambda n: n%2)]
[[2, 4], [1, 3]]
```

This method tees the backing `Iterable`.

Parameters **`pred`** – A function taking a possibly unpacked item and returning a boolean

Returns a 2-`tuple` with the first item being a pipe with items for which `pred` is `False` and the second a pipe with items for which it is `True`

Raises `TypeError` if `pred` is not a `Callable`

`pick`(*key: Any*) → *pypey.pype.Pype*[Any]

Maps each item to the given `key`. Allowed keys are any supported by `object.__item__` :

```
>>> from pypey import pype
>>> list(pype(['a', 'fun', 'day']).pick(0))
['a', 'f', 'd']
```

as well as `@property`-defined object properties and `namedtuple` attributes:

```
>>> from collections import namedtuple
>>> from pypey import pype
>>> Person = namedtuple('Person', ['age'])
>>> list(pype([Person(42), Person(24)]).pick(Person.age))
[42, 24]
```

Equivalent to `Pype.map(lambda item: item.key)()` and `Pype.map(lambda item: item[key])()`.

Parameters **key** – key to pick from each item

Returns a pipe where each item in this pipe has been replaced with the given key

print(*fn: typing.Callable[[...], str] = <class 'str'>, *, sep: str = ' ', end: str = '\n', file: typing.IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>, flush: bool = False, now: bool = True*) → *pypey.pype.Pype*[pypey.func.T]

Prints string returned by given function using built-in `print`:

```
>>> from pypey import pype
>>> p = pype(iter([1, 2, 3])).print()
>>> list(p)
1
2
3
[1, 2, 3]
```

If `now` is set to `True`, the printing takes place immediately and the backing `Iterable` is consumed:

```
>>> from pypey import pype
>>> p = pype(iter([1, 2, 3])).print(now=True)
1
2
3
>>> list(p)
[]
```

The keyword-only parameters are the same as the built-in `print` (minus the `now` flag).

Parameters

- **fn** – A function taking a possibly unpacked item and returning a string
- **sep** – separator as per built-in `print`
- **end** – terminator, as per built-in `print`
- **file** – text stream, as per built-in `print`
- **flush** – flush, as per built-in `print`
- **now** – `False` if the printing should be deferred, `True` otherwise

Returns this pipe, with a possibly consumed backing `Iterable` if `now` is set to `True`

Raises `TypeError` if `fn` is not a `Callable`

reduce(*fn: Callable[[pypey.func.H, pypey.func.T], pypey.func.H], init: Optional[pypey.func.X] = None*) → pypey.func.H

Reduces this pipe to a single value through the application of the given aggregating function to each item:

```
>>> from operator import add
>>> from pypey import pype
>>> pype([1, 2, 3]).reduce(add)
6
```

If `init` is not `None`, it will be placed as the start of the returned pipe and serve as a default value in case the pipe is empty:

```
>>> from pypey import pype
>>> pype([1, 2, 3]).reduce(add, init=-1)
5
```

This function is eager and immediate.

Similar to `functools.reduce`.

Parameters

- **fn** – a function taking an aggregate of the previous items as its first argument and the current item as its second, and returning a new aggregate
- **init** – a value to be placed before all other items if it's not `None`

Returns a value of the same type as the return of the given function, or `init` if the pipe is empty

Raises `TypeError` if `fn` is not a `Callable`

Raises `ValueError` if this pipe is empty and `init` is `None`

reject(*pred: Callable[[...], bool]*) → *pypey.pype.Pype*[pypey.func.T]

Returns a pipe with only the items for each the given predicate returns `False`:

```
>>> from pypey import pype
>>> list(pype(['a', 'FUN', 'day']).reject(str.isupper))
['a', 'day']
```

Opposite of *`Pype.select()`*.

Similar to built-in `filterfalse`.

Parameters **pred** – a function taking a possibly unpacked item and returning a boolean

Returns a pipe with the subset of this pipe's items for which `pred` returns `False`

Raises `TypeError` if `pred` is not a `Callable`

reverse() → *pypey.pype.Pype*[pypey.func.T]

Returns a pipe where this pipe's items appear in reversed order:

```
>>> from pypey import pype
>>> list(pype([1, 2, 3]).reverse())
[3, 2, 1]
```

This operation is eager but deferred.

Similar to built-in `reversed`.

Returns a pipe with items in reversed order

roundrobin() → *pypey.pype.Pype*[pypey.func.T]

Returns a pipe where each item is taken from each of this pipe's elements' in turn:

```
>>> from pypey import pype
>>> list(pype(['a', 'fun', 'day']).roundrobin())
['a', 'f', 'd', 'u', 'a', 'n', 'y']
```

This operation is eager but deferred.

Similar to `more_itertools.interleave_longest`.

Returns A pipe with items taken from this pipe's `Iterable` items

Raises `TypeError` if any of this pipe's items is not an `Iterable`

sample(*k: int*, *seed_: Optional[Any] = None*) → *pypey.pype.Pype*[pypey.func.T]

Returns a pipe with **k** items sampled without replacement from this pipe:

```
>>> from pypey import pype
>>> list(pype([1, 2, 3, 4, 5]).sample(2))
[1, 3]
```

This operation is eager but deferred.

Similar to `random.sample`.

Parameters

- **k** – a non negative `int` specifying how many items to sample
- **seed** – an value to seed the random number generator

Returns a pipe with sampled items from this pipe

Raises `TypeError` if k is not an `int`

Raises `ValueError` if **k** is negative

select(*pred: Callable[[...], bool]*) → *pypey.pype.Pype*[pypey.func.T]

Returns a pipe with only the items for each `pred` returns `True`, opposite of *Pype.reject()*:

```
>>> from pypey import pype
>>> list(pype(['a', 'FUN', 'day']).select(str.isupper))
['FUN']
```

Also known as `filter`.

Similar to built-in `filter`.

Parameters **pred** – a function taking a possibly unpacked item and returning a boolean

Returns a pipe with the subset of this pipe's items for which the given `pred` returns `True`

Raises `TypeError` if `pred` is not a `Callable`

shuffle(*seed_: Optional[Any] = None*) → *pypey.pype.Pype*[pypey.func.T]

Returns a shuffled version of this pipe:

```
>>> from pypey import pype
>>> list(pype([1, 2, 3, 4, 5]).shuffle())
[3, 2, 1, 5, 4]
```

This method is eager but deferred.

Similar to `random.shuffle`

Parameters **seed** – a value to seed the random generator

Returns This pipe, but with its items shuffled

size() → int

Returns number of items in this pipe:

```
>>> from pypey import pype
>>> pype([1, 2, 3]).size()
3
```

This operation is eager and immediate.

Returns an `int` correpsonding to the cardinality of this pipe

slice(*start: int*, *end: int*) → *pypey.pype.Pype*[pypey.func.T]

Returns a slice of this pipe between items at positions `start` and `end`, exclusive:

```
>>> from pypey import pype
>>> list(pype([1, 2, 3, 4]).slice(1, 3))
[2, 3]
```

Similar to `itertools.islice`.

Parameters

- **start** – index of first element to be returned
- **end** – index of element after the last element to be returned

Returns pipe with a slice of the items of this pipe

Raises `TypeError` if `start` or `end` are not `int`-s

Raises `ValueError` if `start` or `end` are negative or if `end` is smaller than `start`

sort(*key: Optional[Callable[[...], pypey.func.Y]] = None*, ***, *rev: bool = False*) → *pypey.pype.Pype*[pypey.func.T]

Sorts this pipe's items, using the return value of `key` if not `None`:

```
>>> from pypey import pype
>>> list(pype(['a', 'funny', 'day']).sort(len))
['a', 'day', 'funny']
```

This method is eager but deferred.

Similar to builtin `sorted`.

Parameters

- **key** – a function possibly taking a unpacked item and returning a value to sort by, or `None`
- **rev** – `True` if the sort order should be reversed, `False` otherwise.

Returns a sorted pipe

Raises `TypeError` if `key` is not a `Callable`

split(*when: Callable[[...], bool]*, *mode: str = 'after'*) → *pypey.pype.Pype*[*pypey.pype.Pype*[pypey.func.T]]

Returns a pipe containing sub-pipes split off this pipe where the given `when` predicate is `True`:

```
>>> from pypey import pype
>>> [list(split) for split in pype(list('afunday')).split(lambda char: char ==
↪'a')
[['a'], ['f', 'u', 'n', 'd', 'a'], ['y']]
```

The default mode is to split after every item for which the predicate is `True`. When `mode` is set to `before`, the split is done before:

```
>>> from pypey import pype
>>> [list(split) for split in pype(list('afunday')).split(lambda char: char ==
↪'a', 'before')]
[['a', 'f', 'u', 'n', 'd'], ['a', 'y']]
```

And when `mode` is set to `at`, the pipe will be split both before and after, leaving the splitting item out:

```
>>> from pypey import pype
>>> [list(split) for split in pype(list('afunday')).split(lambda char: char ==
↪'a', 'at')]
[[], ['f', 'u', 'n', 'd'], ['y']]
```

Similar to `more_itertools.split_before`, `more_itertools.split_after` and `more_itertools.split_at`.

Parameters

- **`when`** – A function possibly taking a unpacked item and returning `True` if this pipe should be split before this item
- **`mode`** – which side of the splitting item the pipe is split, one of `after`, `at` or `before`

Returns a pipe of pipes split off this pipe at items where `when` returns `True`

Raises `TypeError` if `when` is not a `Callable` or `mode`` is not a ``str`

Raises `ValueError` if `mode` is a `str` but not one the supported ones

`take`(*n: int*) → *pypey.pype.Pype*[pypey.func.T]

Returns a pipe containing the first or last `n` items of this pipe, depending on the sign of `n`:

```
>>> from pypey import pype
>>> list(pype([1, 2, 3, 4]).take(-2))
[3, 4]

>>> from pypey import pype
>>>list(pype([1, 2, 3, 4]).take(2))
[1, 2]
```

This operation is eager but deferred when `n` is negative else it's lazy.

Also know as *head* and *tail*.

Parameters **`n`** – a negative `int` specifying the number of items of this pipe's tail or a positive `int` for the first `n` elements

Returns a pipe with this pipe's first or last `n` items

Raises `TypeError` if `n` is not an `int`

`take_while`(*pred: Callable[[...], bool]*) → *pypey.pype.Pype*[pypey.func.T]

Returns a pipe containing this pipe's items until `pred` returns `False` :

```
>>> from pypey import pype
>>> list(pype([1, 2, 3, 4]).take_while(lambda n: n < 4))
[1, 2, 3]
```

Similar to `itertools.takewhile`.

Parameters **pred** – a function taking a possibly unpacked item and returning a boolean

Returns a pipe that is a subset of this one minus the items after `pred` returns `True`

Raises `TypeError` if `pred` is not a `Callable`

`tee`(*n: int*) → *pypey.pype.Pype*[*pypey.pype.Pype*[pypey.func.T]]

Returns `n` lazy copies of this pipe:

```
>>> from pypey import pype
>>> [list(copy) for copy in pype([1, 2, 3]).tee(2)]
[[1, 2, 3], [1, 2, 3]]
```

This method tees the backing `Iterable` but does not replace it (unlike *`Pype.clone()`*).

Similar to `itertools.tee`.

Returns a pipe containing `n` copies of this pipe

Raises `TypeError` if `n` is not an `int`

Raises `ValueError` if `n` is non-positive

`to`(*fn: Callable[[Iterable[pypey.func.T]], pypey.func.Y], *other_fns: Callable[[...], pypey.func.X]*) → Union[pypey.func.Y, pypey.func.X]

Applies given function to this pipe:

```
>>> from pypey import pype
>>> pype(['a', 'fun', 'day']).to(list)
['a', 'fun', 'day']
```

This method is eager if the given function is eager and lazy if it's lazy:

```
>>> from pypey import pype
>>> p = pype(['a', 'fun', 'day']).to(enumerate)
>>> p
<enumerate object at 0x7fdb743003c0>
>>> list(p)
[(0, 'a'), (1, 'fun'), (2, 'day')]
```

If provided with more than one function, it will pipe them together:

```
>>> from pypey import pype
>>> pype(['a', 'fun', 'day']).to(list, len)
3
```

Equivalent to `fn_n(...fn2(fn1(pipe)))`.

Parameters

- **fn** – function to apply to this pipe
- **other_fns** – other functions to be chained with `fn`

Returns the return value of the given function(s)

Raises `TypeError` if any of the provided functions is not a `Callable`

to_file(*target: Union[AnyStr, os.PathLike, int]*, ***, *mode: str = 'w'*, *buffering: int = - 1*, *encoding: Optional[str] = 'utf8'*, *errors: Optional[str] = None*, *newline: Optional[str] = None*, *closefd: bool = True*, *opener: Optional[Callable[[...], int]] = None*, *eol: bool = True*, *now: bool = True*) → *pypey.pype.Pype*[pypey.func.T]

Writes items to file:

```
>>> from tempfile import gettempdir
>>> from os.path import join
>>> from pypey import pype
>>> p = pype(['a', 'fun', 'day']).to_file(join(gettempdir(), 'afunday.txt'),␣
↪eol=False)
>>>list(p)
['a', 'fun', 'day']
>>> list(pype.file(join(gettempdir(), 'afunday.txt')))
['afunday']
```

The first eight parameters are identical to built-in `open`. If `eol` is set to `True`, each item will be converted to string and a line terminator will be appended to it:

```
>>> from pypey import pype
>>> p = pype([1, 2, 3]).to_file(join(gettempdir(), '123.txt', eol=True))
>>> list(p)
[1, 2, 3]
>>> list(pype.file(join(gettempdir(), '123.txt')))
['1', '2', '3']
```

This method is intrinsically lazy but it's set to immediate/eager by default. As such, if `now` is set to `True` and the backing `Iterable` is lazy, it will be consumed and this method will return an empty pipe:

```
>>> from pypey import pype
>>> p = pype(iter([1, 2, 3])).to_file(join(gettempdir(), '123.txt'), now=True)
>>> list(p)
[]
>>> list(pype.file(join(gettempdir(), '123.txt')))
['1', '2', '3']
```

Parameters

- **target** – target to write this pipe's items to
- **mode** – mode as per built-in `open`, except no read modes are allowed
- **buffering** – buffering as per built-in `open`
- **encoding** – encoding as per built-in `open` except the default value is `utf8` instead of None
- **errors** – errors as per built-in `open`
- **newline** – newline as per built-in `open`
- **closefd** – closefd as per built-in `open`
- **opener** – opener as per built-in `open`
- **eol** – `True` if a line separator should be added to each item, `False` otherwise
- **now** – `False` to defer writing until pipe is iterated through, `True` to write immediately

Returns this pipe, possibly after writing its items to file

Raises `ValueError` if `mode` has *r* or *+*

`to_json`(*target: Union[AnyStr, os.PathLike, int]*, **, *mode: str = 'w'*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, *as_dict: bool = True*)

Writes items to a file as a json value:

```
>>> from tempfile import gettempdir
>>> from os.path import join
>>> from pypey import pype
>>> p = pype(['a', 'fun', 'day']).to_json(join(gettempdir(), 'afunday.json'))
<pypey.pype.Pype object at 0x7f7c1971a8d0>
>>> list(pype.json(join(gettempdir(), 'afunday.json')))
['a', 'fun', 'day']
```

The first parameter is the same to built-in `open`, and the rest are identical to the ones in `json.dump` excep the last one which specifies if pairs should be written as dict or as a list. This method will never write single primitives if the pipe contains a single value.

This method is eager and immediate

Parameters

- **`target`** – target to write this pipe's items to
- **`mode`** – mode as per built-in `open`, except no read modes are allowed
- **`skipkeys`** – skipkeys as per built-in `json.dump`
- **`ensure_ascii`** – ensure_ascii as per built-in `json.dump`
- **`check_circular`** – check_circular as per built-in `json.dump`
- **`allow_nan`** – allow_nan as per built-in `json.dump`
- **`cls`** – cls as per built-in `json.dump`
- **`indent`** – indent as per built-in `json.dump`
- **`separators`** – separators as per built-in `json.dump`
- **`default`** – default as per built-in `json.dump`
- **`sort_keys`** – sort_keys as per built-in `json.dump`
- **`as_dict`** – True if item pairs should be written as key-value pairs in an object, False if as a list

Returns this pipe, after writing its items to a file as a json value

Raises `ValueError` if `mode` has *r* or *+*

Raises `TypeError` if `as_dict` is `True` and items are not pairs

`top`(*n: int*, *key: typing.Callable[[pypey.func.T], typing.Any] = <function ident>*) → *pypey.pype.Pype*[pypey.func.T]

Returns a pipe with the `n` items having the highest value, as defined by the `key` function.

```
>>> from pypey import pype
>>> list(pype(['a', 'fun', 'day']).top(2, len))
['fun', 'day']
```

This method is eager but deferred.

Parameters

- **n** – the number of items to return
- **key** – the function defining the value to find the top elements for

Returns a pipe with the top n elements

Raises TypeError` if ``n is not an int or key is not a Callable

Raises ValueError if n is non-positive

uniq() → *pypey.pype.Pype*[pypey.func.T]

Returns unique number of items:

```
>>> from pypey import pype
>>> list(pype(['a', 'b', 'b', 'a']).uniq())
['a', 'b']
```

This method tees the backing Iterable.

Similar to more_itertools.unique_everseen.

Returns A pipe with the unique items in this pipe

unzip() → *pypey.pype.Pype*[*pypey.pype.Pype*[Any]]

Returns a pipe of pipes each with the items of this pipe's Iterable items:

```
>>> from pypey import pype
>>> [list(p) for p in pype(['any', 'fun', 'day']).unzip()]
[['a', 'f', 'd'], ['n', 'u', 'a'], ['y', 'n', 'y']]
```

This method is eager but deferred.

Similar to more_itertools.unzip

Returns a pipe of pipes with the unzipped items in this pipe's Iterable items

Raises TyperError if any of this pipe's items is not an Iterable

window(*size: int*, **, *shift: int = 1*, *pad: Optional[Any] = None*) → *pypey.pype.Pype*[Tuple[Optional[pypey.func.T], ...]]

Returns a pipe containing pipes, each being a sliding window over this pipe's items:

```
>>> from pypey import pype
>>> list(pype(iter([1, 2, 3])).window(size=2))
[(1, 2), (2, 3)]
```

If size is larger than this pipe, pad is used fill in the missing values:

```
>>> from pypey import pype
>>> list(pype(iter([1, 2, 3])).window(size=4, pad=-1))
[(1, 2, 3, -1)]
```

Similar to more_itertools.windowed.

Parameters

- **size** – the size of the window
- **shift** – the shift between successive windows

- **pad** – the value to use to fill missing values

Returns a pipe of pipes, each being a sliding window over this pipe

Raises `TypeError` if either ``size` or `shift` is not an `int`

Raises `ValueError` if `size` is negative or `shift` is non-positive

zip(**others: Iterable[Any]*, *trunc: bool = True*, *pad: Optional[Any] = None*) → *pypey.pype.Pype*[Tuple[pypey.func.T, ...]]

Zips items in this pipe with each other or with items in each of the given `Iterable`-s. If no `Iterable`-s are provided, the items in this pipe will be zipped with each other:

```
>>> from pypey import pype
>>> list(pype(['a', 'fun', 'day']).zip(trunc=False, pad='?'))
[('a', 'f', 'd'), ('?', 'u', 'a'), ('?', 'n', 'y')]
```

Self-zipping will consume the backing `Iterable` if it's lazy. If other `Iterable`-s are provided, the items in this pipe will be zipped with the items in those:

```
>>> from pypey import pype
>>> list(pype(['a', 'fun', 'day']).zip([1, 2, 3, 4]))
[('a', 1), ('fun', 2), ('day', 3)]
```

Similar to built-in `zip` and `itertools.zip_longest`.

Parameters

- **others** – `Iterables` to be zipped with this with this pipe
- **trunc** – `True` to truncate all `Iterable`-s to the size of the shortest one, `False` to pad all to the size of the longest one
- **pad** – value to pad shorter `Iterable`-s with if `trunc` is `False`; if it's `True` it's ignored

Returns a pipe with the zipped items of this pipe with each other or with the given `Iterable`-s' ones

Raises `TypeError` any of `others` is not an `Iterable`

zip_with(*fn: Callable[[...], pypey.func.Y]*) → *pypey.pype.Pype*[Tuple[pypey.func.T, pypey.func.Y]]

Returns a pipe where each item is a 2-`tuple` with this pipe's item as the first and the output of `fn` as the second. This is useful for adding an extra piece of data to the current pipeline:

```
>>> from pypey import pype
>>> list(pype(['a','fun', 'day']).zip_with(len))
[('a', 1), ('fun', 3), ('day', 3)]
```

and it's a more concise version of:

```
>>> from pypey import pype
>>> list(pype(['a','fun', 'day']).map(lambda w: (w, len(w))))
[('a', 1), ('fun', 3), ('day', 3)]
```

Parameters **fn** – a function taking a possibly unpacked item and returning a value to be zipped with this pipe's item

Returns a new pipe with zipped items

TOTAL = _TOTAL_

Constant indicating the aggregated counts in *Pype.freqs()*

class Total

Bases: `str`

4.1.3 func

Functions and constants for a concise use of higher-level functions

ident(*item: pypey.func.T*) → pypey.func.T

Identity function, returns the argument passed to it.

Parameters **item** – any argument

Returns the argument passed in

pipe(**functions: Callable*) → Callable

Chains given functions.

```
>>> from pypey import pipe
>>> from math import sqrt
>>> [pipe(len, sqrt)(w) for w in ('a', 'fun','day')]
[1.0, 1.7320508075688772, 1.7320508075688772]
```

For functions taking multiple arguments, the return of the previous function in the chain will be unpacked only if it's a `tuple`:

```
>>> from pypey import pipe
>>> pipe(divmod, lambda quotient, remainder: quotient + remainder)(10, 3)
4
```

If a function returns an `Iterable` that it's not a tuple but unpacking in the next function is still needed, built-in `tuple` can be inserted in between to achieve the desired effect:

```
>>> from pypey import pipe
>>> pipe(range, tuple,  lambda _1, _2_, _3: sum([_1, _3]))(3)
2
```

Conversely, if a function returns a `tuple` but unpacking is not required in the next function, built-in `list` can be used to achieve the desired effect:

```
>>> from pypey import pipe
>>> pipe(divmod, list, sum)(10, 3)
4
```

Note that `list` is the only exception to the rule that `tuple` returns will be unpacked.

Parameters **functions** – a variable number of functions

Returns a combined function

px

Concise alias of `functools.partial`

require(*cond: bool*, *message: str*, *exception: typing.Type[Exception] = <class 'TypeError'>*)

Guard clause, useful for implementing exception-raising checks concisely, especially useful in lambdas.

```
>>> from pypey import require, pype
>>> pype([1,2,'3']).do(lambda n: require(isinstance(n, int), 'not an int'),␣
↪now=True)
Traceback (most recent call last):
   ...
TypeError: not an int
```

Parameters

- **cond** – if `False` the given exception will be thrown, otherwise this function is a no-op
- **message** – exception message
- **exception** – exception to throw if `cond` is `False`, defaults to `TypeError`

Returns nothing

require_val(*cond: bool*, *message: str*)

Throws `ValueError` exception if `cond` is `False`, equivalent to *require()* with `exception=ValueError`.

```
>>> from pypey import require_val, pype
>>> pype([1,2,-3]).do(lambda n: require_val(n>0, 'not a positive number'), now=True)
Traceback (most recent call last):
    ...
ValueError: not a positive number
```

Parameters

- **cond** – if `False` the a ValueError will be thrown, otherwise this function is a no-op
- **message** – the exception message

Returns nothing

throw(*exception: Type[Exception]*, *message: str*)

Throws given exception with given message, equivalent to built-in `raise`. This function is useful for raising exceptions inside lambdas as `raise` is syntactically invalid in them.

```
>>> from pypey import throw, pype
>>> pype([1,2,3]).do(lambda n: throw(ValueError, 'test'), now=True)
Traceback (most recent call last):
  ...
ValueError: test
```

Parameters

- **exception** – the exception to throw
- **message** – the exception message

Returns nothing

Fn

Callable type; Callable[[int], str] is a function of (int) -> str.

The subscription syntax must always be used with exactly two values: the argument list and the return type. The argument list must be a list of types or ellipsis; the return type must be a single type.

There is no syntax to indicate optional or keyword arguments, such function types are rarely used as callback types.

alias of `Callable`

4.1.4 dycts

Factory methods matching those of `Pype` and `dict`

`countdyct`
CountDyct factory

`defdyct`
DefDyct factory

`dyct`
Dyct factory

4.1.5 dyct

Pipeable versions of `dict`, `collections.defaultdict` and `collections.Counter`

`class CountDyct`(*counts: Iterable[pypey.dyct.K] = ()*)
Bases: `Generic`[`pypey.dyct.K`], `collections.Counter`

A pipeable version of `collections.Counter` containing a superset of its methods

`inc`(*item: pypey.dyct.K*, *count: int = 1*) → *pypey.dyct.CountDyct*[pypey.dyct.K]
Increments counts of `item` by `count`.

Parameters

- **`item`** – item to be incremented
- **`count`** – increment

Returns this `CountDyct`

`pype`() → *pypey.pype.Pype*[Tuple[pypey.dyct.K, int]]
Return this `CountDyct-`'s items as a `Pype`

Returns `Pype` containing pairs of key, count

`class DefDyct`(*default_factory: Optional[Callable[[...], pypey.func.Y]]*, *fn: Optional[Callable[[pypey.dyct.V, pypey.dyct.V], pypey.func.Y]] = None*, **args*, ***kwargs*)
Bases: `Generic`[`pypey.dyct.K`, `pypey.dyct.V`], `collections.defaultdict`

A pipeable version of `collections.defaultdict` containing a superset of its methods

`add`(*key: pypey.dyct.K*, *value: pypey.dyct.V*) → *pypey.dyct.DefDyct*[pypey.dyct.K, pypey.dyct.V]
Updates value associated with given key with given value, using `fn` passed in in constructor If `fn` is None, then inplace addition will be used.

Parameters

- **`key`** – key to add
- **`value`** – value to add

Returns this `DefDyct`

`pype`() → *pypey.pype.Pype*[Tuple[pypey.dyct.K, pypey.dyct.V]]
Return this `DefDyct-`'s items as a `Pype`

Returns `Pype` containing pairs of key, values

class Dyct(**args*, ***kwargs*)

Bases: `Generic`[`pypey.dyct.K`, `pypey.dyct.V`], `collections.UserDict`

A pipeable version of `dict` containing a superset of its methods

pype() → *pypey.pype.Pype*[Tuple[pypey.dyct.K, pypey.dyct.V]]

Return this `Dyct`-'s items as a `Pype`

Returns `Pype` containing pairs of key, values

reverse(*overwrite: bool = True*) → *pypey.dyct.Dyct*[pypey.dyct.V, Union[pypey.dyct.K, Set[pypey.dyct.K]]]

Reverses keys and values in this `Dyct`. To prevent keys mapping to equal values from being lost, set `overwrite` to `False` and they will be kept in a `Set`.

Parameters **overwrite** – `True` if keys should be overwritten `False` if they should be preserved in a `Set`.

Returns a new `Dyct` with keys for values and values for keys

set(*key: pypey.dyct.K*, *value: pypey.dyct.V*) → *pypey.dyct.Dyct*[pypey.dyct.K, pypey.dyct.V]

sets given key to given value

Parameters

- **key** – key to add/overwrite
- **value** – value to (re-)set

Returns this `Dyct`

4.2 License

pypey is under the MIT License. See the LICENSE file.

See also:

The MIT License

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX