

Design

v1.0.0

The package will be implemented in Go (1).

```
interface Stage:
```

```
* Handle(Item) Item: Handles the Item to be moved through the Pipe.
```

Stage corresponds to stages of computations (2). Item refers to any type (7).

```
StageFunc func(Item) Item
```

StageFunc is a wrapper that converts functions which would satisfy Stage into Stages.

```
class Pipe(...Stage):
```

```
* Receive(Item): Receive puts the Item in the Pipe to be processes.
```

```
* Deliver() Item: Deliver blocks until an Item is done being processed and  
    returns it.
```

Pipe connects Stages in the given order (2, 5). Receive and Deliver hide concurrency by immediately returning after putting an Item in the Pipe and blocking until an Item is done (6). Concurrent functions are started for every Stage whenever the Pipe has an Item that handles Items concurrently when available then places them in the next Stage (3, 4). These functions exit when the Pipe is empty. Pipe accepts varargs since the more common use-case is a hard-coded list of Stages.

```
func Process(Pipe, ...Item) []Item
```

Process is a utility to run many Items through a Pipe (8). It accepts varargs instead of a list so that arguments don't need to be converted to Items to be put into the list.

v1.1.0

```
interface Consumer:
```

```
* Consume(Item): Consumes the Item.
```

Consumer receives Items (1).

```
ConsumerFunc func(Item)
```

ConsumerFunc is a wrapper that converts functions which would satisfy Consumer into Consumers.

```
ProcessAndConsume(Pipe, Consumer, ...Item)
```

ProcessAndConsume is a utility that works like Process except it passes the Items to the Consumer as they come out of the Pipe (2).

v1.2.0

```
interface Producer:
* Produce() (Item, bool)
```

`Producer` creates `Items` (1). `false` is returned when the `Producer` is done and the `Item` returned with `false` is ignored.

```
ProducerFunc func() (Item, bool)
```

`ProducerFunc` is a wrapper that converts functions which would satisfy `Producer` into `Producers`.

```
func ProduceAndProcess(Pipe, Producer) []Item
```

`ProduceAndProcess` is a utility that works like `Process` except it produces `Items` from the `Producer` to pass into the `Pipe` (2).

```
func ProduceProcessAndConsume(Pipe, Producer, Consumer)
```

`ProduceProcessAndConsume` is a utility that works like `Process` except it produces `Items` from the `Producer` to pass into the `Pipe` and it passes the `Items` to the `Consumer` as they come out of the `Pipe` (3).