Functions **wrap-up**

heh.

```python
# program.py

statement1
statement2
statement3

#…
```

```python
# program.py

def foo():
    statement1
    statement2
    statement3
    return

var = foo()

#…
```

```
# program.py

# 1st run
statement1
statement2
statement3

# 2nd run
statement1
statement2
statement3

#…
```

```
# program.py

def foo():
    statement1
    statement2
    statement3
    return

var1 = foo()
var2 = foo()

#…
```

mandatory args optional args

```
def foo(x, y, z=3):
    return x + y + z
```

```
foo(1, 2, 4)
7
foo(1, 2)
6
foo(x=1, y=2)
6
```

```
def foo(*, x, y, z=3):
    return x + y + z

# …
foo(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes 0 positional
arguments but 2 were given
```

```
def foo(*, x, y, z=3):
    return x + y + z

# …
foo(x=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() missing 1 required
keyword-only argument: 'y'
```

```
from arbitrary_module import search_tweets

# text = …

search_tweets(text, 8, 5)
search_tweets(text, 12, 10, True)
```

```
from arbitrary_module import search_tweets

# text = …

search_tweets(text, max_old=8, count=5)
search_tweets(text, retweets=True)
```

```
def search_tweets(txt, *, max_old=24, count=10, retweets=False):
    # …
```

↓

```
# text = …
search_tweets(text, 12, 10, True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: search_tweets() takes 1
positional arguments but 4 were given
```

5 stages of grief

- Denial
- Anger
- Bargaining
- Depression
- Acceptance

1.Denial

No, I don’t need to create a new type!

```
configured_func = configure(**config)
configured_func(*args)
```

```
def configure(**cfg):
    return ?
```

```
warning = logger(logtype='WARNING')
error = logger(logtype='ERROR')

>>> warning('careful')
WARNING: careful

>>> error('oops')
ERROR: oops
```

```
def logger(logtype):
    return ?
```

λ

```
var = lambda x: statement(x) # return function
var(x) # returns statement(x)
```

```
def logger(logtype):
    return ?
```

λ

```
>>> def func(x):
        return print(x)

>>> var = func
>>> var(‘ayo’)
ayo

>>> var = lambda x: print(x)
>>> var(‘ayo’)
ayo
```

```
warning = logger(logtype='WARNING')
error = logger(logtype='ERROR')

>>> warning('careful')
WARNING: careful

>>> error('oops')
ERROR: oops
```

```
def logger(logtype):
    return ?
```

2.Anger

```
def log_me(file, msg):
    print(msg)
    f = open(…

logger = lambda logtype: lambda msg: …
warning = logger(‘WARNING’)
system_log = lambda log: logme(‘syslog’, log)

system_log(warning(‘careful’))
# too much lambda!!!
```

2.Anger

```
def log_me(file, msg):
    print(msg)
    f = open(…
```

a proper function definition anyway

```
logger = lambda logtype: lambda msg: …
warning = logger('WARNING')
system_log = lambda log: logme('syslog', log)

system_log(warning('careful'))
# too much lambda!!!
```

Closures

```
from datetime import datetime as dt
t = dt.now

def logger(logtype):
    def log_me(msg):
        log_msg = '[{}] {}: {}'.format(t(), logtype, msg)
        # write log to file
        print(log_msg)
    return log_me
```

Closures

```
def closure_func(val):
    value = val
    def inner_function():
        # change the value here
        return value
    return inner_function
```

Possible???

```
def counter(val):
    value = val
    def count():
        nonlocal value
        value += 1
        return value
    return count

>>> c = counter(0)
>>> c()
1
>>> c()
2
...
```

get reference to

3.Bargaining

In search for help from higher power...

```
def print_my_name(func):
    def new_func(*args, **kwargs):
        print(func.__name__) # do stuff with func
        return func(*args, **kwargs)
    return new_func
```

This time returning the function as intended

Decorators

```
def print_my_name(func):
    def new_func(*args, **kwargs):
        print(func.__name__)
        return func(*args, **kwargs)
    return new_func

>>> prints_name = print_my_name(function)
>>> function() == prints_name()
function
True
```

```
def print_my_name(func):
    def new_func(*args, **kwargs):
        print(func.__name__)
        return func(*args, **kwargs)
    return new_func

@print_my_name
def foo(x):
    return x

>>> foo(1)
foo
1
>>>
```

equals to:

```
foo = print_my_name(foo)
```

4.Depression

(kill me now)

```
>>> c = counter()
>>> for result in c:
...     # do stuff with result
...
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'function' object is not iterable
```

Generators

```
def generator(x):
    yield x
    yield x
    yield x

>>> g = generator(1)
>>> next(g)
1
>>> next(g)
1
```

Generators

```
...
>>> next(g)
1
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
def counter(val):
    value = val
    def count():
        nonlocal value
        value += 1
        return value
    return inner_function

>>> c = counter(0)
>>> c()
1
>>> c()
2
...
```

This is a sequence!!!

5.Acceptance

well.. ok then

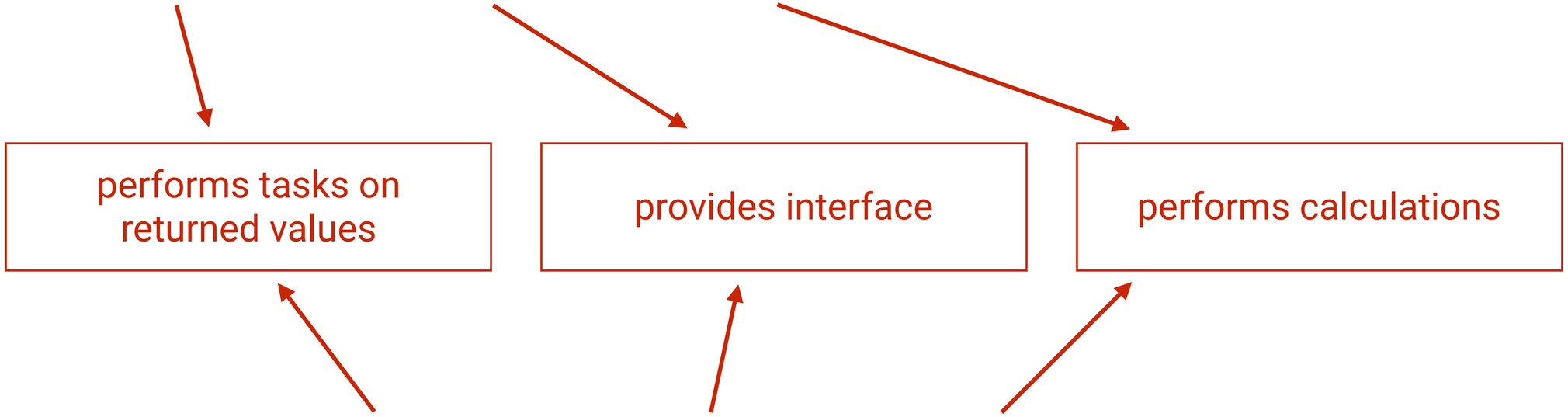
performs tasks on returned values

provides interface

performs calculations

```
results = manage(controller(device(eval_func)))
for result in results:
    # do stuff
```

```
...
system_log = lambda log: logme('syslog', log)
system_log(warning('careful'))
```




```
results = manage(controller(device(eval_func)))
for result in results:
    # do stuff
```