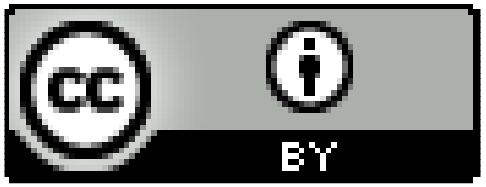
Python

First-Class Functions



An integer is 32 bits of data...

An integer is 32 bits of data...

...that variables can refer to

An integer is 32 bits of data...

...that variables can refer to

A string is a sequence of bytes representing characters...

An integer is 32 bits of data...

...that variables can refer to

A string is a sequence of bytes representing characters...

...that variables can refer to

An integer is 32 bits of data...

...that variables can refer to

A string is a sequence of bytes representing characters...

...that variables can refer to

A function is a sequence of bytes representing instructions...

An integer is 32 bits of data...

...that variables can refer to

A string is a sequence of bytes representing characters...

...that variables can refer to

A function is a sequence of bytes representing instructions...

...and yes, variables can refer to them to

An integer is 32 bits of data...

...that variables can refer to

A string is a sequence of bytes representing characters...

...that variables can refer to

A function is a sequence of bytes representing instructions...

...and yes, variables can refer to them to

This turns out to be very useful, and very powerful

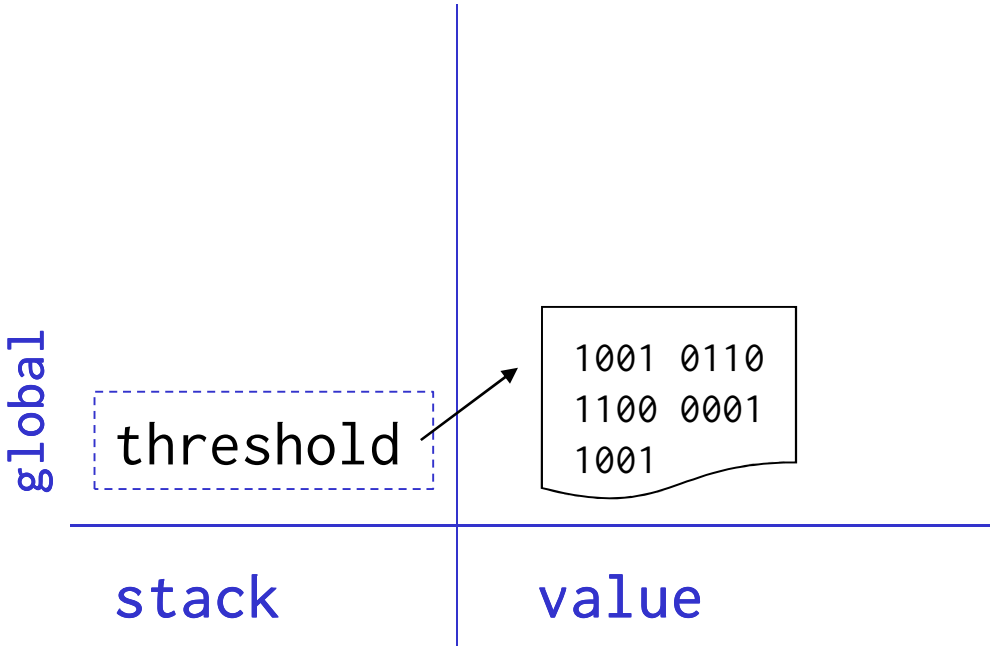
What happens when a function is defined

What happens when a function is defined

```
def threshold(signal):
  return 1.0 / sum(signal)
```

What happens when a function is defined

```
def threshold(signal):
  return 1.0 / sum(signal)
```

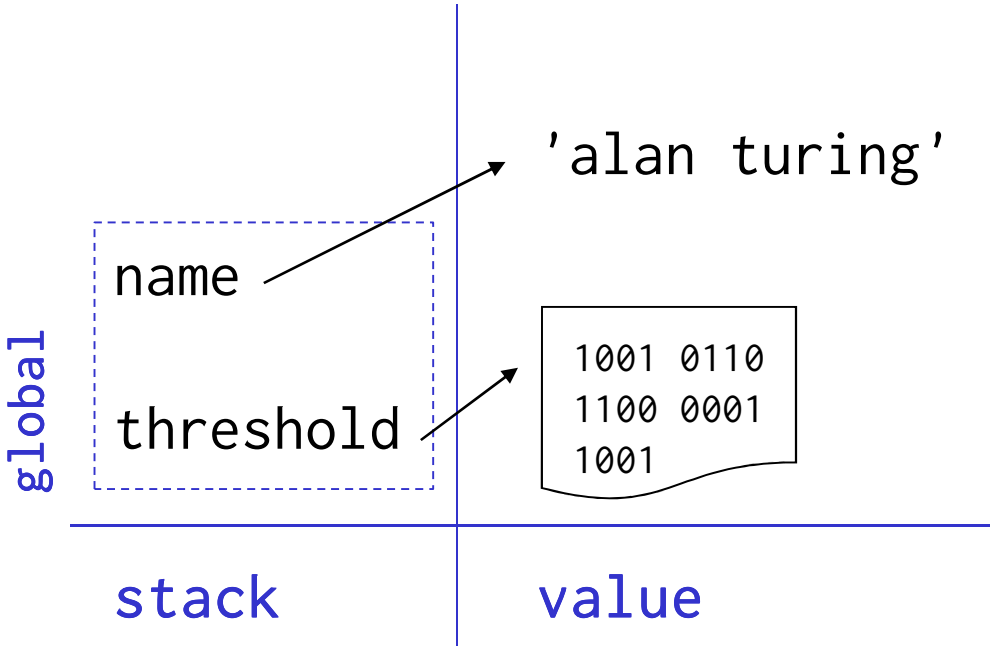


What happens when a function is defined

```
def threshold(signal):
  return 1.0 / sum(signal)
```

Not really very different from:

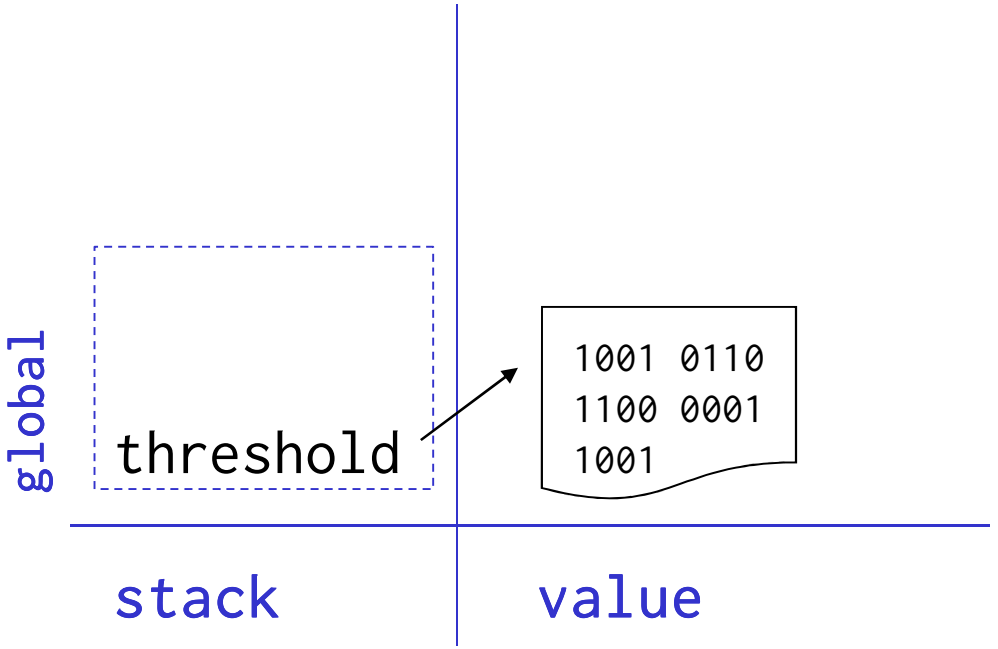
```
name = 'Alan Turing'
```



Can assign that value to another variable

Can assign that value to another variable

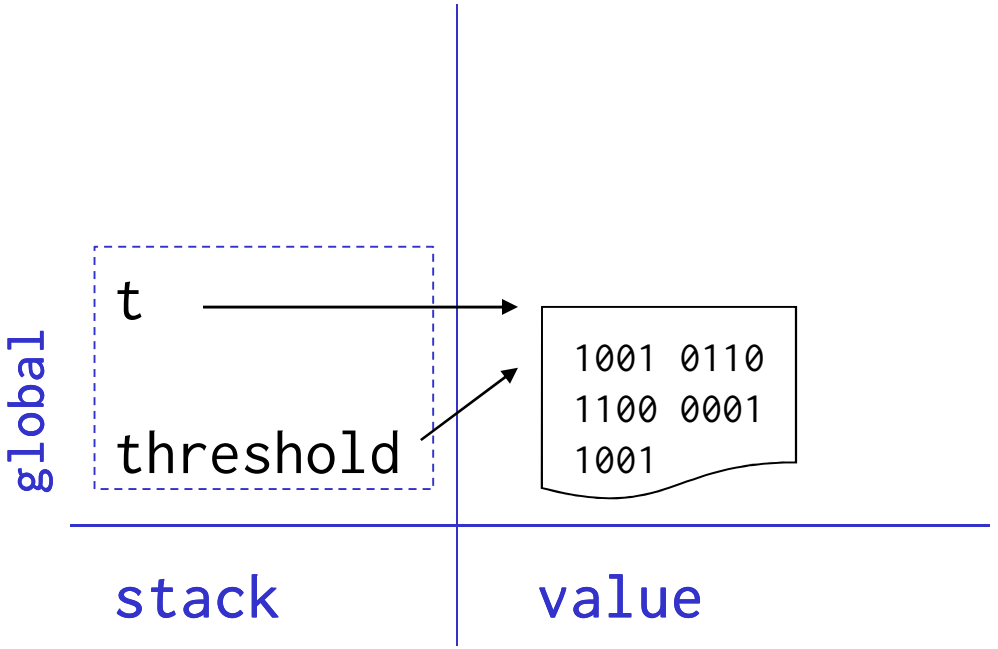
```
def threshold(signal):
  return 1.0 / sum(signal)
```



Can assign that value to another variable

```
def threshold(signal):
  return 1.0 / sum(signal)

t = threshold
```

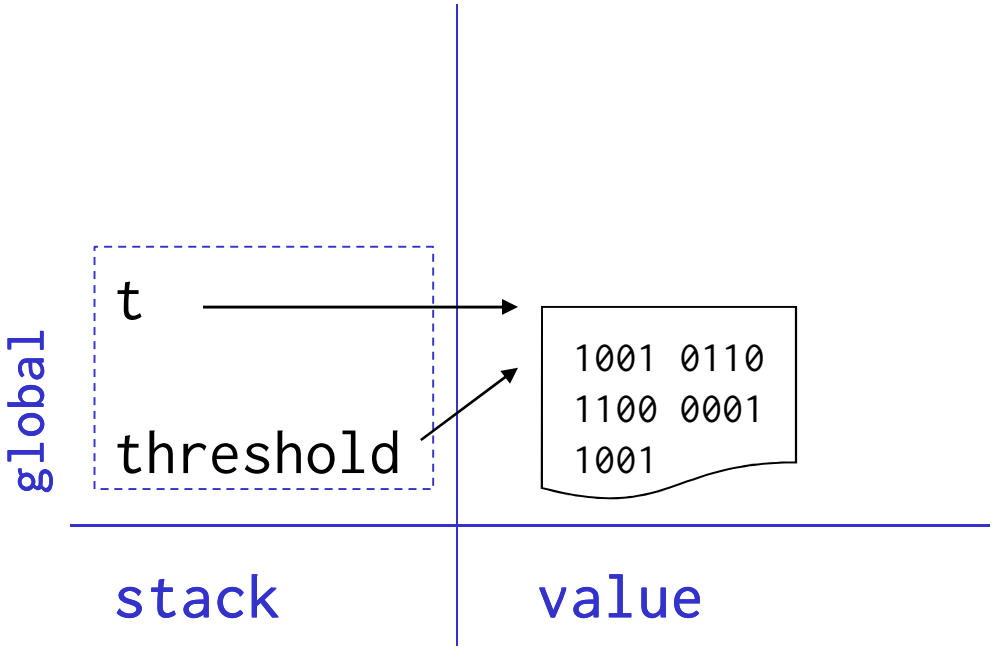


Can assign that value to another variable

```
def threshold(signal):
  return 1.0 / sum(signal)

t = threshold
print t([0.1, 0.4, 0.2])
```

1.42857

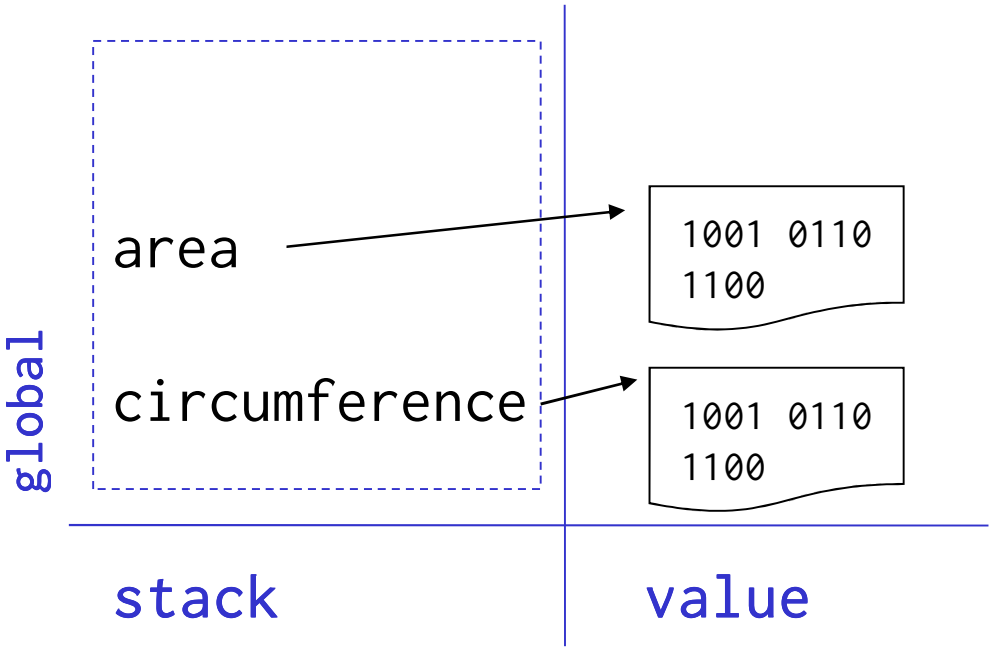


Can put (a reference to) the function in a list

Can put (a reference to) the function in a list

```
def area(r):
  return PI * r * r

def circumference(r):
  return 2 * PI * r
```

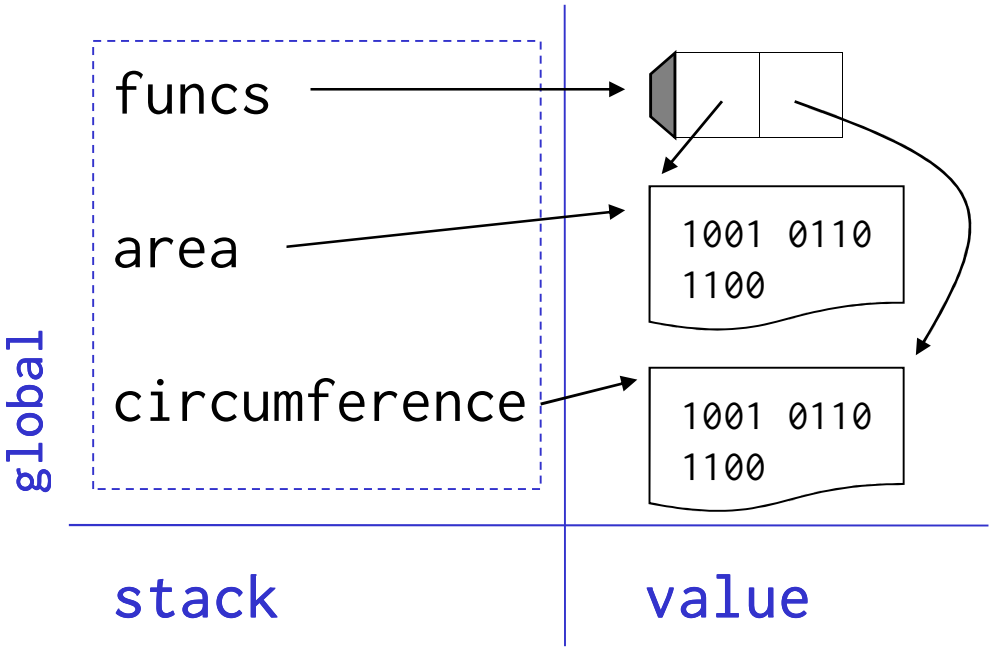


Can put (a reference to) the function in a list

```
def area(r):
  return PI * r * r

def circumference(r):
  return 2 * PI * r

funcs = [area, circumference]
```



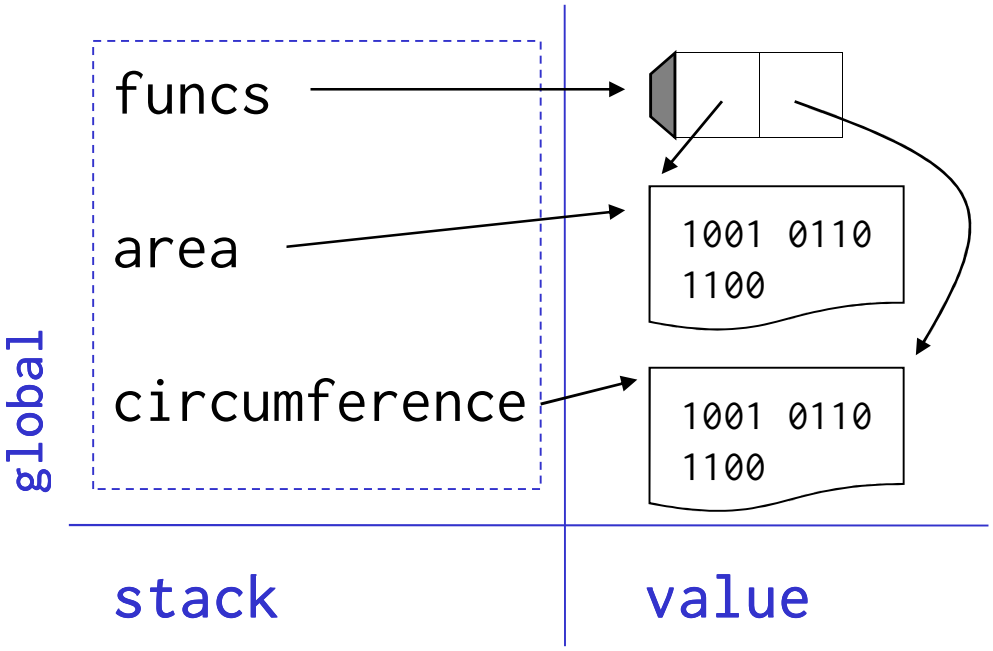
Can put (a reference to) the function in a list

```
def area(r):
  return PI * r * r

def circumference(r):
  return 2 * PI * r

funcs = [area, circumference]

for f in funcs:
  print f(1.0)
```



Can put (a reference to) the function in a list

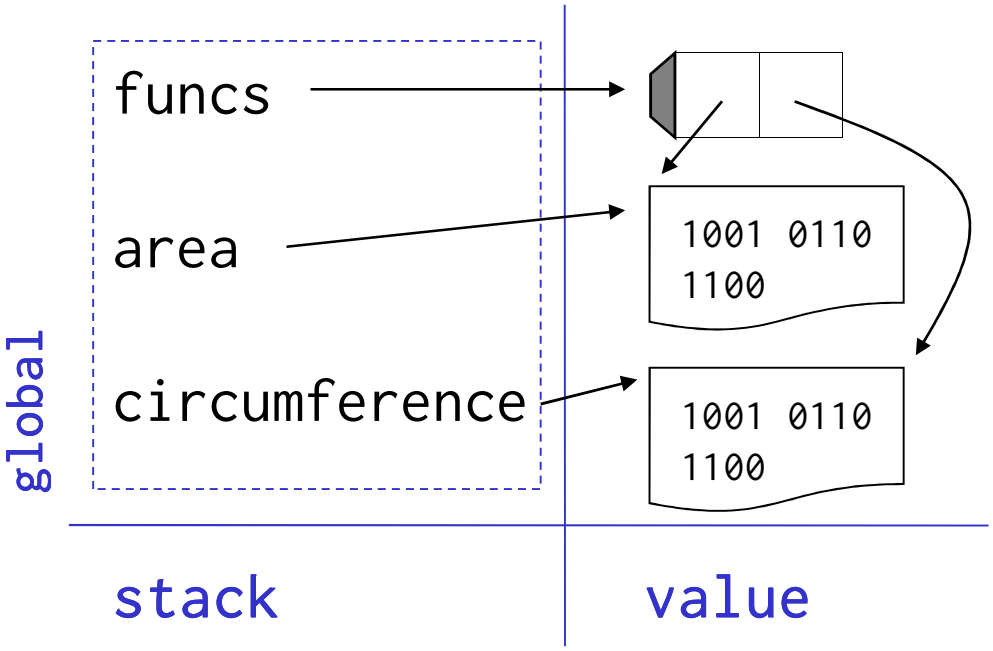
```
def area(r):
  return PI * r * r

def circumference(r):
  return 2 * PI * r

funcs = [area, circumference]

for f in funcs:
  print f(1.0)
```

```
3.14159
6.28318
```



Can pass (a reference to) the function into a function

Can pass (a reference to) the function into a function

```
def call_it(func, value):
  return func(value)
```

Can pass (a reference to) the function into a function

```
def call_it(func, value):
  return func(value)

print call_it(area, 1.0)
```
3.14159

Can pass (a reference to) the function into a function

```
def call_it(func, value):
  return func(value)

print call_it(area, 1.0)
```
3.14159

```
print call_it(circumference, 1.0)
```
6.28318

Can now write functions of functions

Can now write functions of functions

```
def do_all(func, values):
  result = []
  for v in values:
    temp = func(v)
    result.append(temp)
  return result
```

Can now write functions of functions

```
def do_all(func, values):
  result = []
  for v in values:
    temp = func(v)
    result.append(temp)
  return result
```

```
print do_all(area, [1.0, 2.0, 3.0])
```

Can now write functions of functions

```
def do_all(func, values):
  result = []
  for v in values:
    temp = func(v)
    result.append(temp)
  return result
```

```
print do_all(area, [1.0, 2.0, 3.0])
[3.14159, 12.56636, 28.27431]
```

Can now write functions of functions

```
def do_all(func, values):
  result = []
  for v in values:
    temp = func(v)
    result.append(temp)
  return result
```

```
print do_all(area, [1.0, 2.0, 3.0])
[3.14159, 12.56636, 28.27431]
```

```
def slim(text):
  return text[1:-1]
```

Can now write functions of functions

```
def do_all(func, values):
  result = []
  for v in values:
    temp = func(v)
    result.append(temp)
  return result
```

```
print do_all(area, [1.0, 2.0, 3.0])
```

```
[3.14159, 12.56636, 28.27431]
```

```
def slim(text):
  return text[1:-1]

print do_all(slim, ['abc', 'defgh'])
```

```
b efg
```

Higher-order functions allow re-use of control flow

Higher-order functions allow re-use of control flow

```
def combine_values(func, values):
  current = values[0]
  for i in range(1, len(values)):
    current = func(current, v)
  return current
```

Higher-order functions allow re-use of control flow

```
def combine_values(func, values):
  current = values[0]
  for i in range(1, len(values)):
    current = func(current, v)
  return current

def add(x, y): return x + y
def mul(x, y): return x * y
```

Higher-order functions allow re-use of control flow

```
def combine_values(func, values):
  current = values[0]
  for i in range(1, len(values)):
    current = func(current, v)
  return current

def add(x, y): return x + y
def mul(x, y): return x * y

print combine_values(add, [1, 3, 5])
9
```

Higher-order functions allow re-use of control flow

```
def combine_values(func, values):
  current = values[0]
  for i in range(1, len(values)):
    current = func(current, v)
  return current

def add(x, y): return x + y
def mul(x, y): return x * y

print combine_values(add, [1, 3, 5])
9
print combine_values(mul, [1, 3, 5])
15
```

Without higher order functions

Without higher order functions

	op_1	op_2	op_3
data_structure_A	do_1A	do_2A	do_3A
data_structure_B	do_1B	do_2B	do_3B
data_structure_C	do_1C	do_2C	do_3C

Without higher order functions

	op_1	op_2	op_3
data_structure_A	do_1A	do_2A	do_3A
data_structure_B	do_1B	do_2B	do_3B
data_structure_C	do_1C	do_2C	do_3C

total: 9

Without higher order functions

	op_1	op_2	op_3
data_structure_A	do_1A	do_2A	do_3A
data_structure_B	do_1B	do_2B	do_3B
data_structure_C	do_1C	do_2C	do_3C

total: 9

With higher order functions

Without higher order functions

	op_1	op_2	op_3
data_structure_A	do_1A	do_2A	do_3A
data_structure_B	do_1B	do_2B	do_3B
data_structure_C	do_1C	do_2C	do_3C

total: 9

With higher order functions

	op_1	op_2	op_3
operate_on_A			
operate_on_B			
operate_on_C			

Without higher order functions

	op_1	op_2	op_3
data_structure_A	do_1A	do_2A	do_3A
data_structure_B	do_1B	do_2B	do_3B
data_structure_C	do_1C	do_2C	do_3C

total: 9

With higher order functions

	op_1	op_2	op_3
operate_on_A			
operate_on_B			
operate_on_C			

total: 6

Must need to know *something* about the function in order to call it

Must need to know *something* about the function

in order to call it

Like number of arguments

Must need to know *something* about the function

in order to call it

Like ~~number of arguments~~

Must need to know *something* about the function

in order to call it

Like ~~number of arguments~~

```
def add_all(*args):
  total = 0
  for a in args:
    total += a
  return total
```

Must need to know *something* about the function

in order to call it

Like ~~number of arguments~~

```
def add_all(*args):
  total = 0
  for a in args:
    total += a
  return total
```

Must need to know *something* about the function

in order to call it

Like ~~number of arguments~~

```
def add_all(*args):
  total = 0
  for a in args:
    total += a
  return total

print add_all()
```

```
0
```

Must need to know *something* about the function

in order to call it

Like ~~number of arguments~~

```
def add_all(*args):
  total = 0
  for a in args:
    total += a
  return total

print add_all()
0
print add_all(1, 2, 3)
6
```

Combine with "regular" parameters

Combine with "regular" parameters

```
def combine_values(func, *values):
  current = values[0]
  for i in range(1, len(values)):
    current = func(current, v)
  return current
```

Combine with "regular" parameters

```
def combine_values(func, *values):
  current = values[0]
  for i in range(1, len(values)):
    current = func(current, v)
  return current
```

Combine with "regular" parameters

```
def combine_values(func, *values):
  current = values[0]
  for i in range(1, len(values)):
    current = func(current, v)
  return current

print combine_values(add, 1, 3, 5)
9
```

Combine with "regular" parameters

```
def combine_values(func, *values):
  current = values[0]
  for i in range(1, len(values)):
    current = func(current, v)
  return current

print combine_values(add, 1, 3, 5)
9
```

What does `combine_values(add)` do?

Combine with "regular" parameters

```
def combine_values(func, *values):
  current = values[0]
  for i in range(1, len(values)):
    current = func(current, v)
  return current

print combine_values(add, 1, 3, 5)
9
```

What does `combine_values(add)` do?

What *should* it do?

`filter(F, S)`	select elements of `S` for which `F` is `True`

`filter(F, S)`	select elements of `S` for which `F` is `True`
`map(F, S)`	apply `F` to each element of `S`

`filter(F, S)`	select elements of `S` for which `F` is `True`
`map(F, S)`	apply `F` to each element of `S`
`reduce(F, S)`	use `F` to combine all elements of `S`

`filter(F, S)`	select elements of `S` for which `F` is `True`
`map(F, S)`	apply `F` to each element of `S`
`reduce(F, S)`	use `F` to combine all elements of `S`

```
def positive(x): return x >= 0
print filter(positive, [-3, -2, 0, 1, 2])
```

```
[0, 1, 2]
```

`filter(F, S)`	select elements of `S` for which `F` is `True`
`map(F, S)`	apply `F` to each element of `S`
`reduce(F, S)`	use `F` to combine all elements of `S`

```
def positive(x): return x >= 0
print filter(positive, [-3, -2, 0, 1, 2])
```

```
[0, 1, 2]
```

```
def negate(x): return -x
print map(negate, [-3, -2, 0, 1, 2])
```

```
[3, 2, 0, -1, -2]
```

`filter(F, S)`	select elements of `S` for which `F` is `True`
`map(F, S)`	apply `F` to each element of `S`
`reduce(F, S)`	use `F` to combine all elements of `S`

```
def positive(x): return x >= 0
print filter(positive, [-3, -2, 0, 1, 2])
```
```
[0, 1, 2]
```

```
def negate(x): return -x
print map(negate, [-3, -2, 0, 1, 2])
```
```
[3, 2, 0, -1, -2]
```

```
def add(x, y): return x+y
print reduce(add, [-3, -2, 0, 1, 2])
```
```
-2
```

What is programming?

What is programming?

Novice: writing instructions for the computer

What is programming?

Novice: writing instructions for the computer

Expert: creating and combining abstractions

What is programming?

Novice: writing instructions for the computer

Expert: creating and combining abstractions

figure out what the pattern is

What is programming?

Novice: writing instructions for the computer

Expert: creating and combining abstractions

figure out what the pattern is

write it down as clearly as possible

What is programming?

Novice: writing instructions for the computer

Expert: creating and combining abstractions

figure out what the pattern is

write it down as clearly as possible

build more patterns on top of it

What is programming?

Novice: writing instructions for the computer

Expert: creating and combining abstractions

figure out what the pattern is

write it down as clearly as possible

build more patterns on top of it

But limits on short-term memory still apply

What is programming?

Novice: writing instructions for the computer

Expert: creating and combining abstractions

figure out what the pattern is

write it down as clearly as possible

build more patterns on top of it

But limits on short-term memory still apply

Hard to understand what meta-meta-functions actually *do*

created by

Greg Wilson

October 2010