Python

Functions

A programming language should *not* include everything anyone might ever want

A programming language should *not* include everything anyone might ever want

Instead, it should make it easy for people to create what they need to solve specific problems

A programming language should *not* include everything anyone might ever want

Instead, it should make it easy for people to create what they need to solve specific problems

Define functions to create higher-level operations

A programming language should *not* include everything anyone might ever want

Instead, it should make it easy for people to create what they need to solve specific problems

Define functions to create higher-level operations

"Create a language in which the solution to your original problem is trivial."

Define functions using `def`

Define functions using `def`

```
def greet():
  return 'Good evening, master'
```

Define functions using def

```
def greet():
  return 'Good evening, master'

temp = greet()
print temp
Good evening, master
```

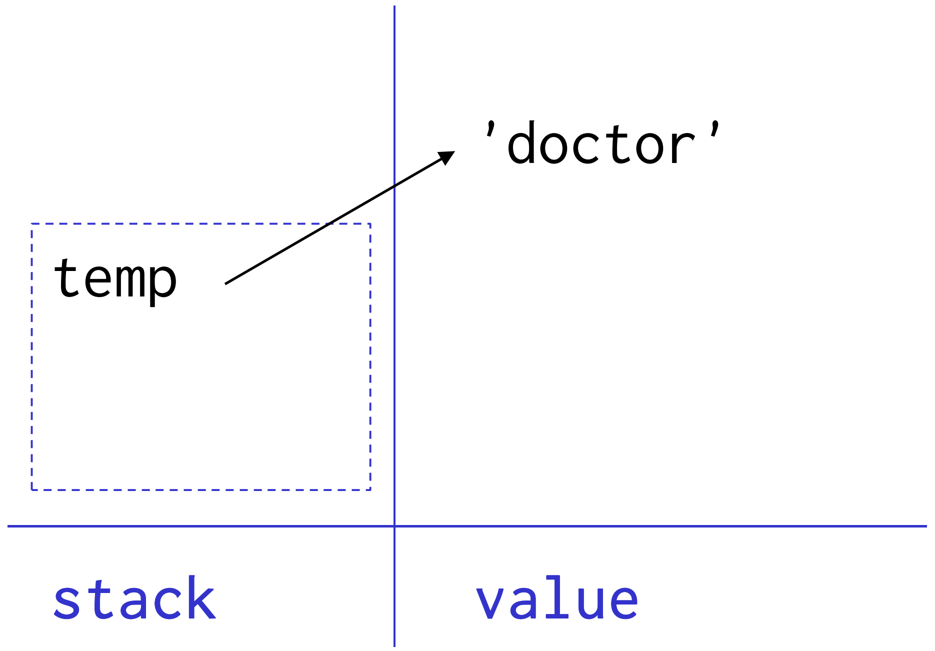
Give them parameters

Give them parameters

```
def greet(name):
  answer = 'Hello, ' + name
  return answer
```

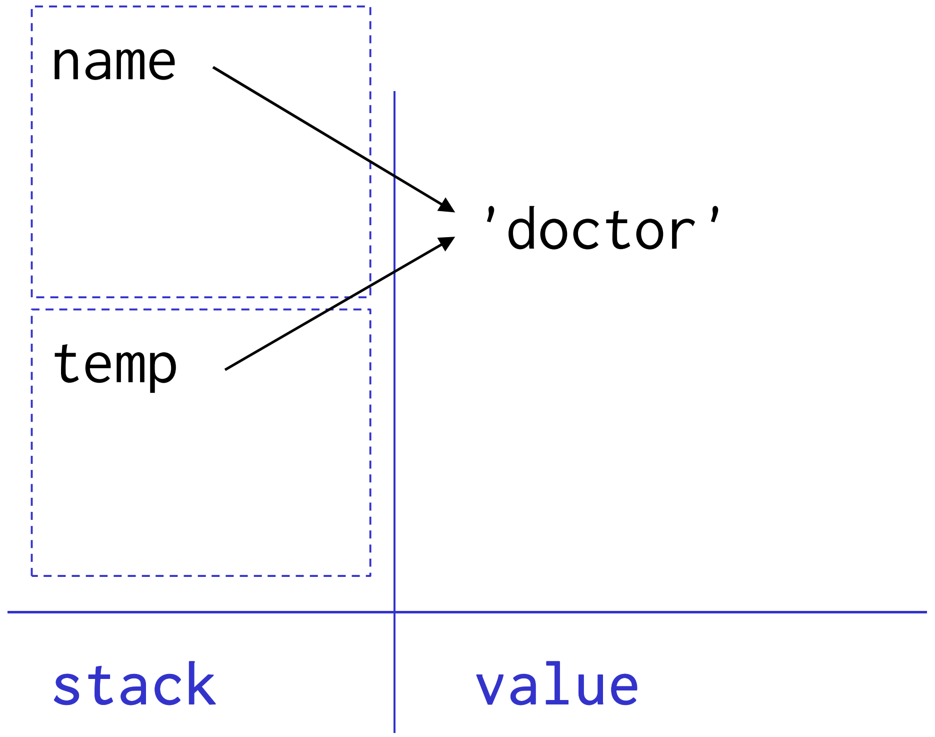
Give them parameters

```
def greet(name):
  answer = 'Hello, ' + name
  return answer

temp = 'doctor'
```



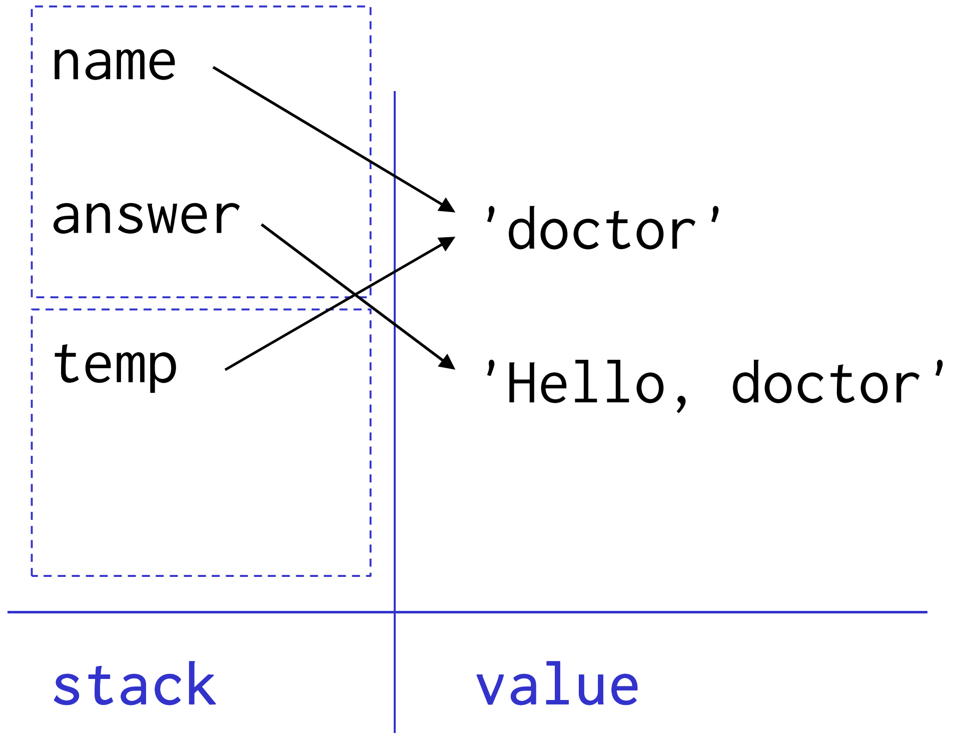
Give them parameters

```
def greet(name):
  answer = 'Hello, ' + name
  return answer

temp = 'doctor'
result = greet(temp)
```



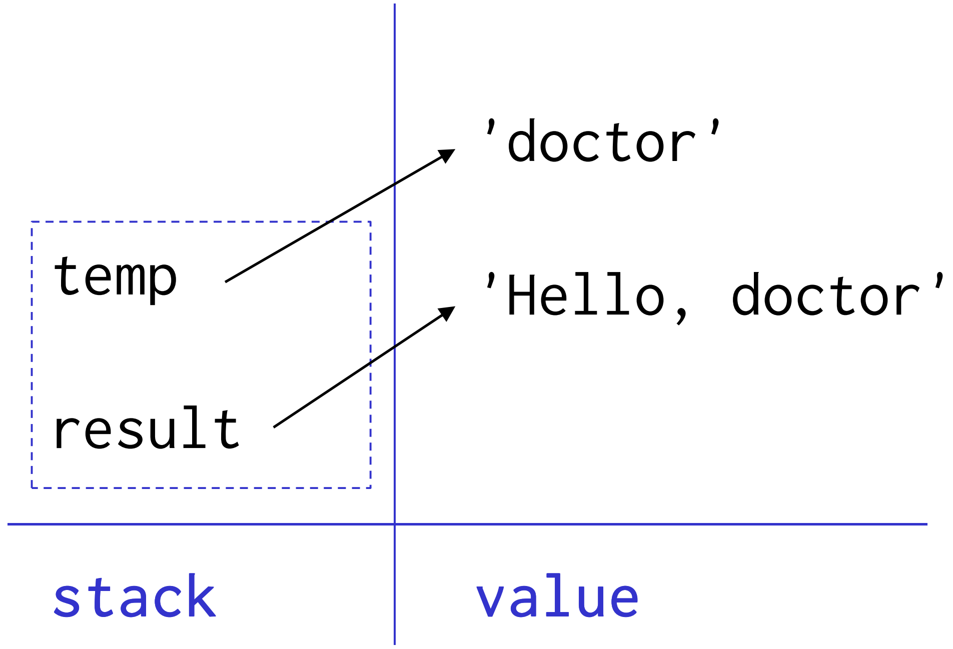
Give them parameters

```
def greet(name):
  answer = 'Hello, ' + name
  return answer

temp = 'doctor'
result = greet(temp)
```



Give them parameters

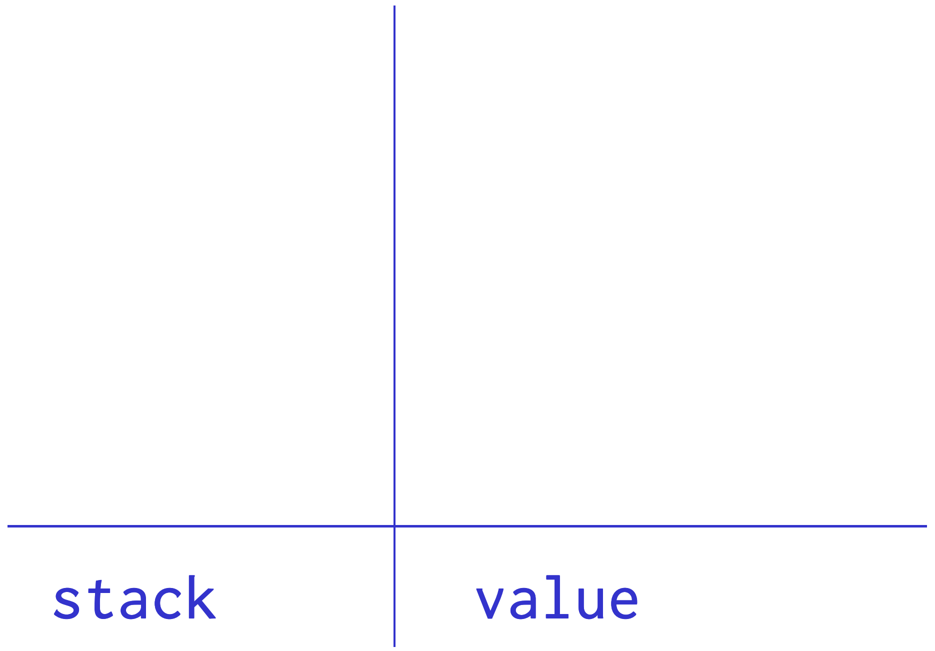
```
def greet(name):
  answer = 'Hello, ' + name
  return answer

temp = 'doctor'
result = greet(temp)
```



Each function call creates a new *stack frame*

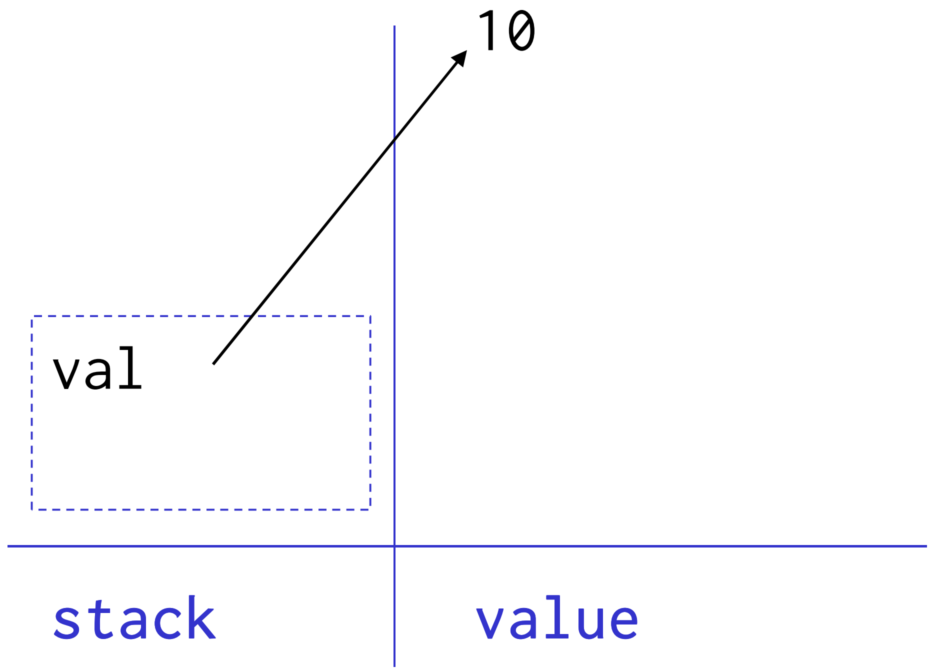
Each function call creates a new *stack frame*

```
def add(a):
  b = a + 1
  return b

def double(c):
  d = 2 * add(c)
  return d
```



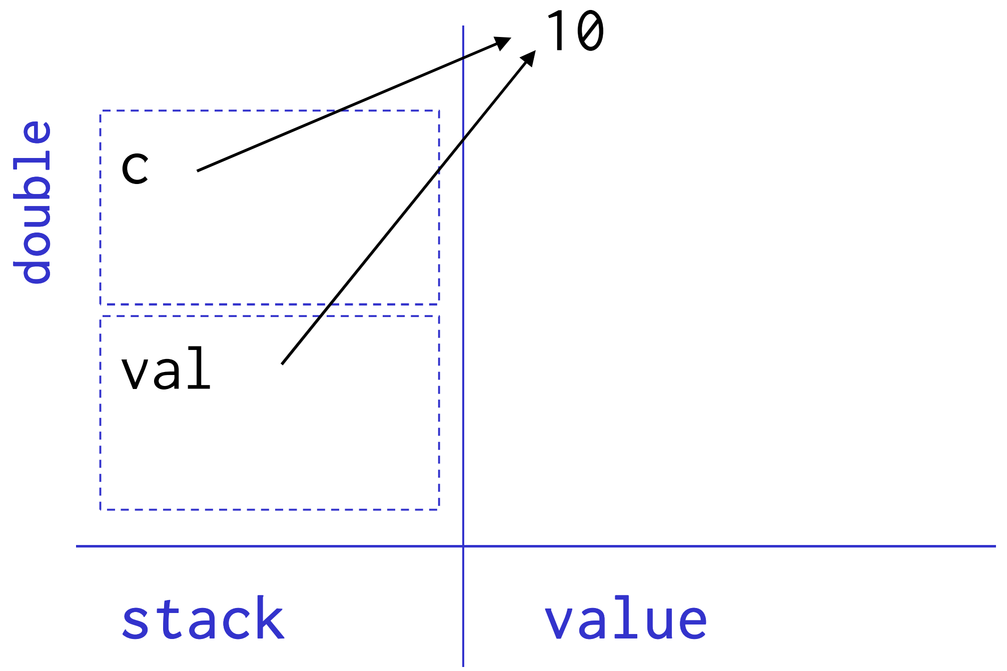
Each function call creates a new *stack frame*

```
def add(a):
  b = a + 1
  return b

def double(c):
  d = 2 * add(c)
  return d

val = 10
```



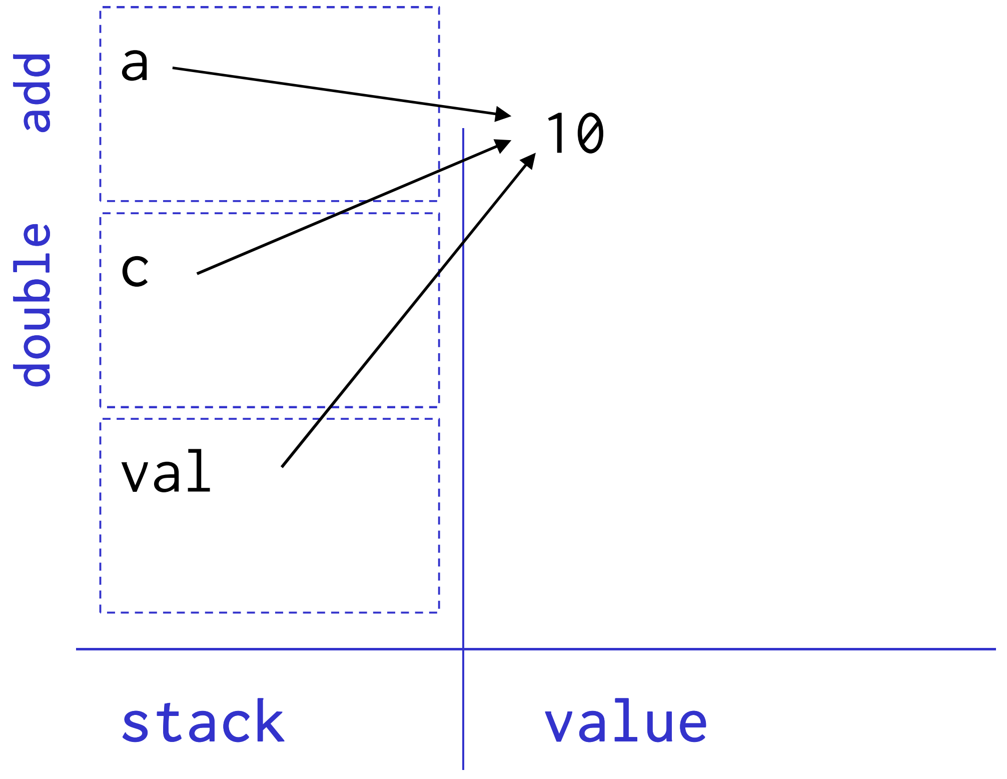
Each function call creates a new *stack frame*

```
def add(a):
  b = a + 1
  return b

def double(c):
  d = 2 * add(c)
  return d

val = 10
result = double(val)
```



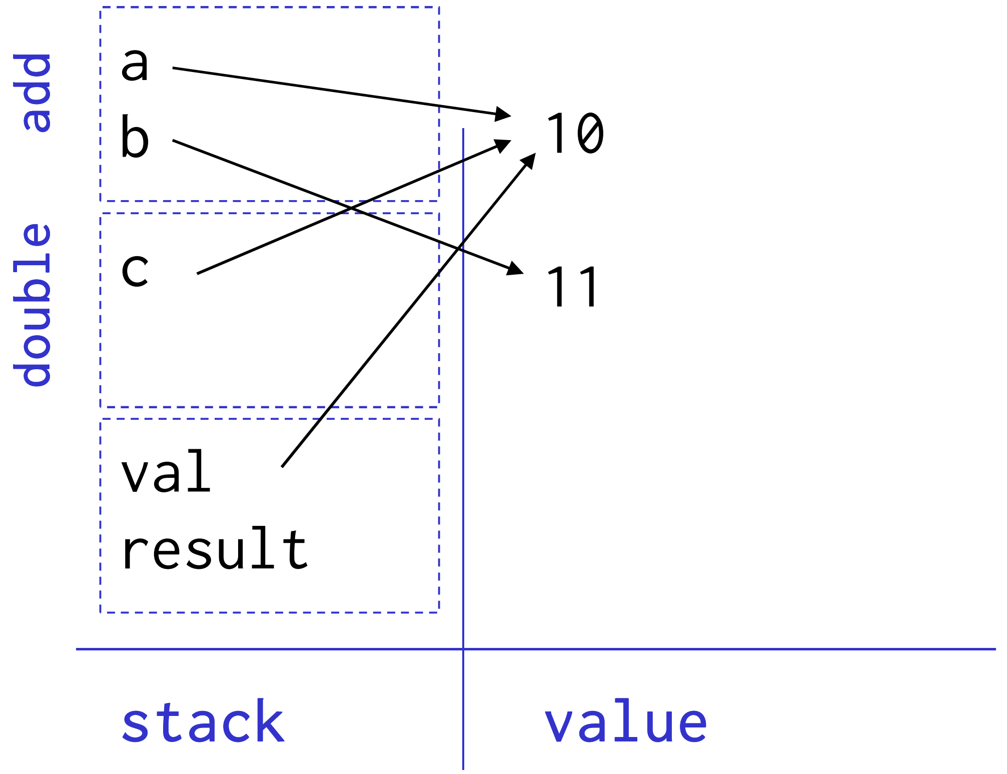
Each function call creates a new *stack frame*

```
def add(a):
  b = a + 1
  return b

def double(c):
  d = 2 * add(c)
  return d

val = 10
result = double(val)
```



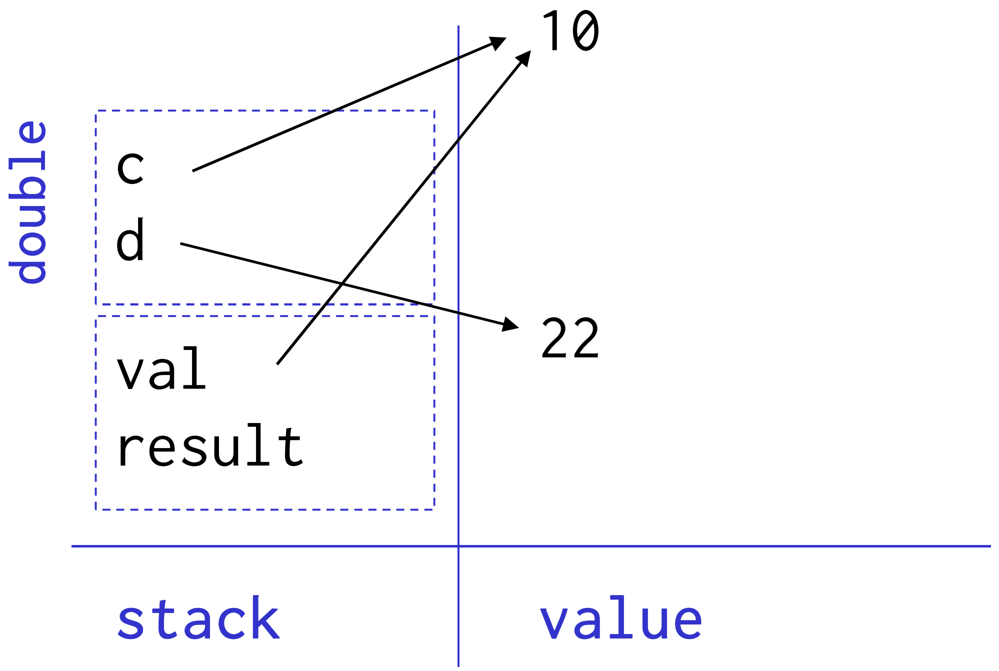
Each function call creates a new *stack frame*

```
def add(a):
  b = a + 1
  return b

def double(c):
  d = 2 * add(c)
  return d

val = 10
result = double(val)
print result
```



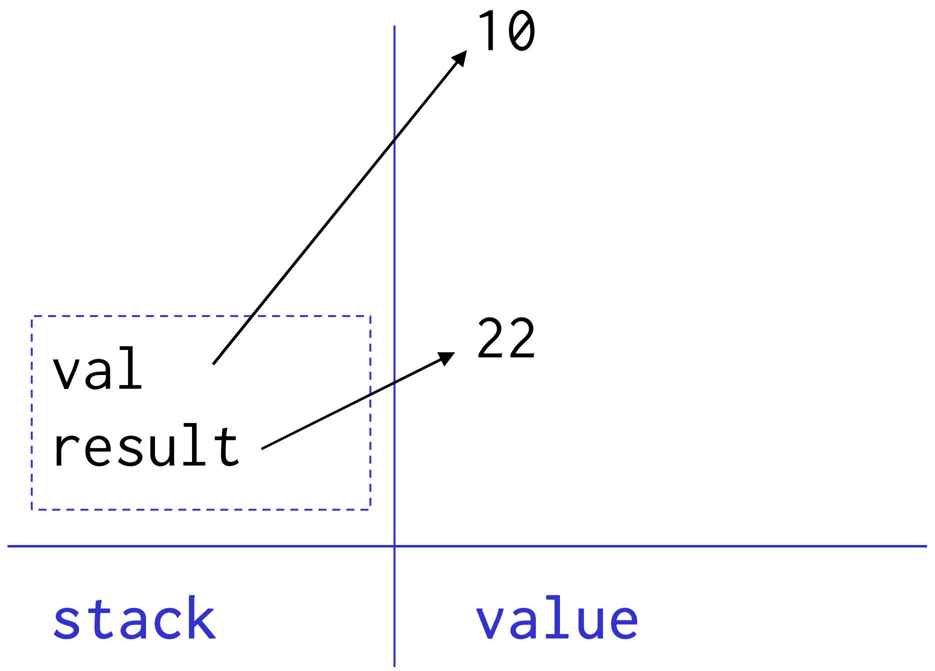
Each function call creates a new *stack frame*

```
def add(a):
  b = a + 1
  return b

def double(c):
  d = 2 * add(c)
  return d

val = 10
result = double(val)
print result
```



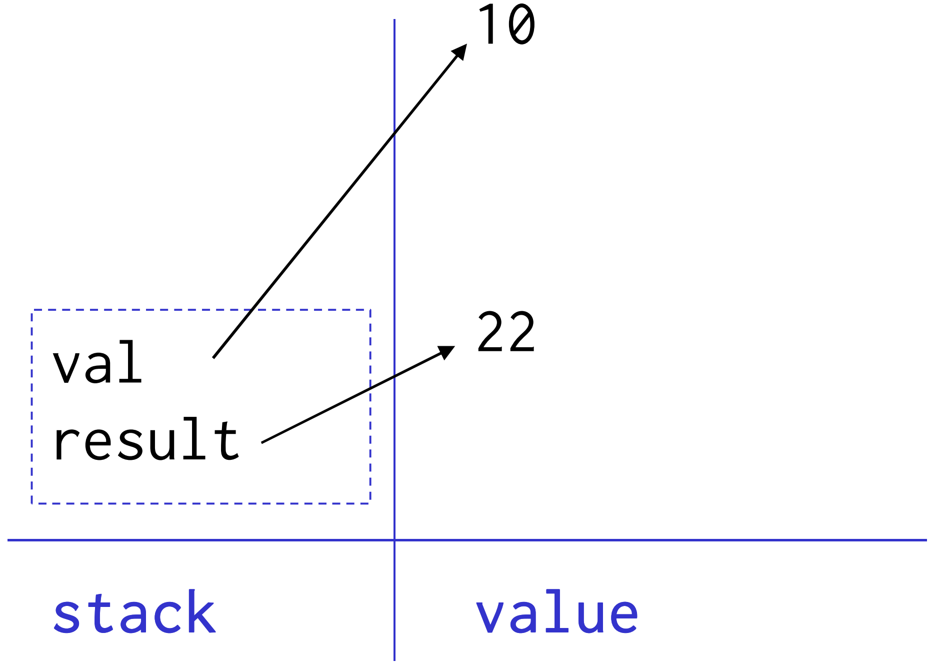
Each function call creates a new *stack frame*

```
def add(a):
  b = a + 1
  return b

def double(c):
  d = 2 * add(c)
  return d

val = 10
result = double(val)
print result
```



Each function call creates a new *stack frame*

```
def add(a):
  b = a + 1
  return b

def double(c):
  d = 2 * add(c)
  return d

val = 10
result = double(val)
print result
22
```



Only see variables in the *current* and *global* frames

Only see variables in the *current* and *global* frames

Current beats global

Only see variables in the *current* and *global* frames

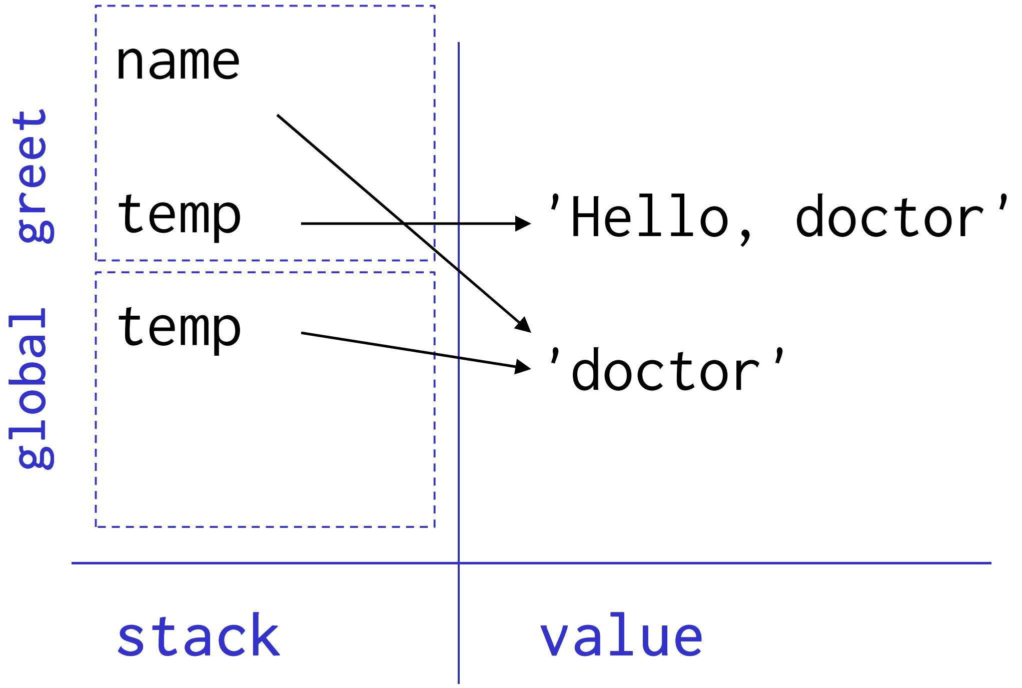
Current beats global

```
def greet(name):
  temp = 'Hello, ' + name
  return temp

temp = 'doctor'
result = greet(temp)
```

Only see variables in the *current* and *global* frames

Current beats global

```
def greet(name):
  temp = 'Hello, ' + name
  return temp

temp = 'doctor'
result = greet(temp)
```

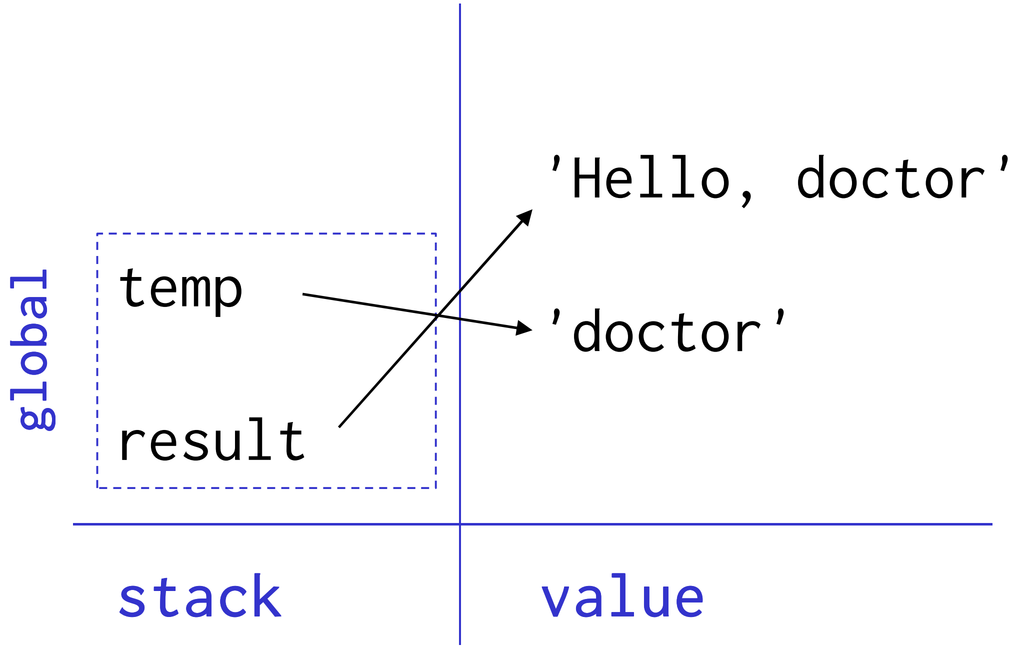


Only see variables in the *current* and *global* frames

Current beats global

```
def greet(name):
  temp = 'Hello, ' + name
  return temp

temp = 'doctor'
result = greet(temp)
print result
```

```
Hello, doctor
```



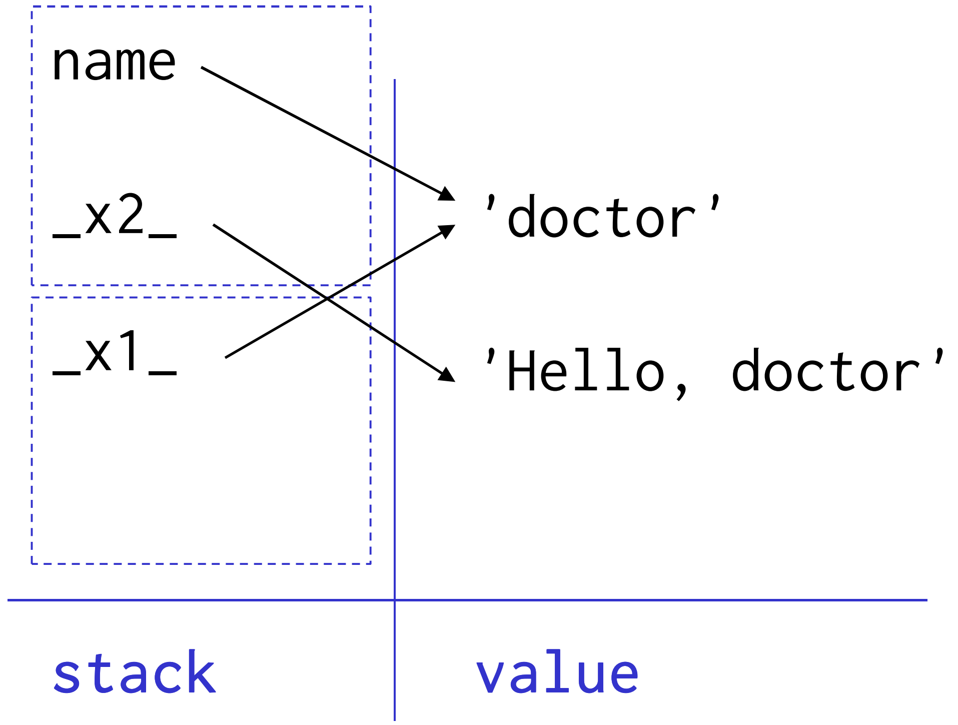
Can pass values in and accept results directly

Can pass values in and accept results directly

```
def greet(name):
  return 'Hello, ' + name

print greet('doctor')
```

Can pass values in and accept results directly

```
def greet(name):
  return 'Hello, ' + name

print greet('doctor')
```



Can return at any time

Can return at any time

```
def sign(num):
  if num > 0:
    return 1
  elif num == 0:
    return 0
  else:
    return -1
```

Can return at any time

```
def sign(num):
  if num > 0:
    return 1
  elif num == 0:
    return 0
  else:
    return -1

print sign(3)
1
```

Can return at any time

```
def sign(num):
  if num > 0:
    return 1
  elif num == 0:
    return 0
  else:
    return -1

print sign(3)
1
print sign(-9)
-1
```

Can return at any time

```
def sign(num):
  if num > 0:
    return 1
  elif num == 0:
    return 0
  else:
    return -1

print sign(3)
1
print sign(-9)
-1
```

Over-use makes functions hard to understand

Can return at any time

```
def sign(num):
  if num > 0:
    return 1
  elif num == 0:
    return 0
  else:
    return -1

print sign(3)
1
print sign(-9)
-1
```

Over-use makes functions

hard to understand

No prescription possible, but:

Can return at any time

```
def sign(num):
  if num > 0:
    return 1
  elif num == 0:
    return 0
  else:
    return -1

print sign(3)
1
print sign(-9)
-1
```

Over-use makes functions hard to understand

No prescription possible, but:

– a few at the beginning to handle special cases

Can return at any time

```
def sign(num):
  if num > 0:
    return 1
  elif num == 0:
    return 0
  else:
    return -1

print sign(3)
1
print sign(-9)
-1
```

Over-use makes functions hard to understand

No prescription possible, but:

– a few at the beginning to handle special cases

– one at the end for the "general" result

Every function returns something

Every function returns something

```
def sign(num):
  if num > 0:
    return 1
  elif num == 0:
    return 0
# else:
#   return -1
```

Every function returns something

```
def sign(num):
  if num > 0:
    return 1
  elif num == 0:
    return 0
# else:
#   return -1

print sign(3)
1
```

Every function returns something

```
def sign(num):
  if num > 0:
    return 1
  elif num == 0:
    return 0
# else:
#   return -1

print sign(3)
1
print sign(-9)
None
```

Every function returns something

```
def sign(num):
  if num > 0:
    return 1
  elif num == 0:
    return 0
# else:
#   return -1

print sign(3)
1
print sign(-9)
None
```

If the function doesn't return a value, Python returns `None`

Every function returns something

```
def sign(num):
  if num > 0:
    return 1
  elif num == 0:
    return 0
# else:
#   return -1

print sign(3)
1
print sign(-9)
None
```

If the function doesn't return a value, Python returns None

Yet another reason why commenting out blocks of code is a bad idea...

Functions and parameters don't have types

Functions and parameters don't have types

```
def double(x):
  return 2 * x
```

Functions and parameters don't have types

```
def double(x):
  return 2 * x

print double(2)
4
```

Functions and parameters don't have types

```
def double(x):
  return 2 * x

print double(2)
4
print double('two')
twotwo
```

Functions and parameters don't have types

```
def double(x):
  return 2 * x

print double(2)
4
print double('two')
twotwo
```

Only use this when the function's behavior depends *only* on properties that all possible arguments share software carpentry

Functions and parameters don't have types

```
def double(x):
  return 2 * x

print double(2)
4
print double('two')
twotwo
```

Only use this when the function's behavior depends *only* on properties that all possible arguments share

```
if type(arg) == int:
  ...
elif type(arg) == str:
  ...
...
```

Functions and parameters don't have types

```
def double(x):
  return 2 * x

print double(2)
4
print double('two')
twotwo
```

Only use this when the function's behavior depends *only* on properties that all possible arguments share

Warning sign

```
if type(arg) == int:
  ...
elif type(arg) == str:
  ...
...
```

Functions and parameters don't have types

```
def double(x):
  return 2 * x

print double(2)
4
print double('two')
twotwo
```

Only use this when the function's behavior depends *only* on properties that all possible arguments share

Warning sign

There's a better way to do this

```
if type(arg) == int:
  ...
elif type(arg) == str:
  ...
...
```

Values are copied into parameters

Values are copied into parameters

Which means lists are aliased

Values are copied into parameters

Which means lists are aliased

```
def appender(a_string, a_list):
    a_string += 'turing'
    a_list.append('turing')
```

Values are copied into parameters

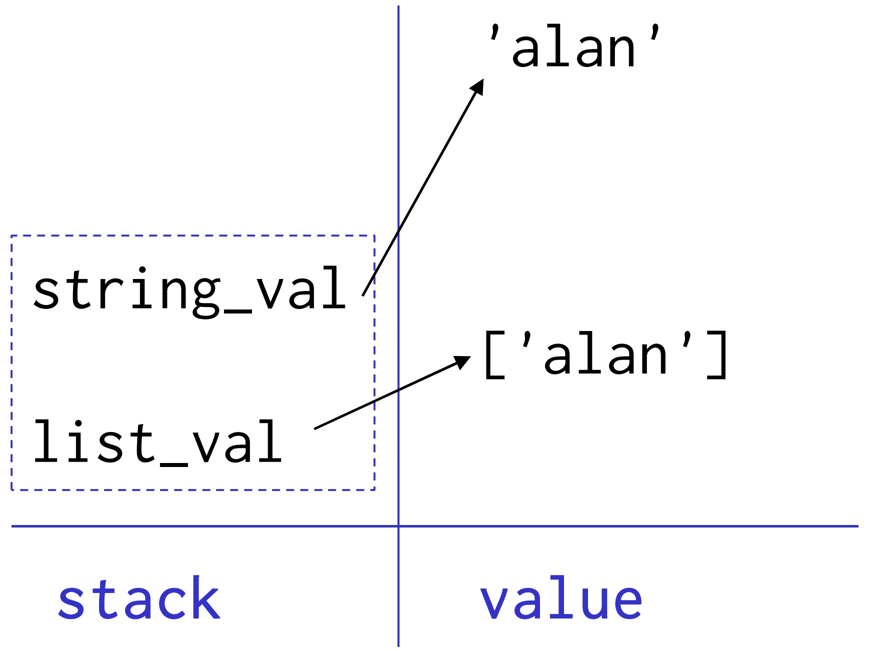
Which means lists are aliased

```
def appender(a_string, a_list):
    a_string += 'turing'
    a_list.append('turing')

string_val = 'alan'
list_val = ['alan']
appender(string_val, list_val)
```

Values are copied into parameters

Which means lists are aliased

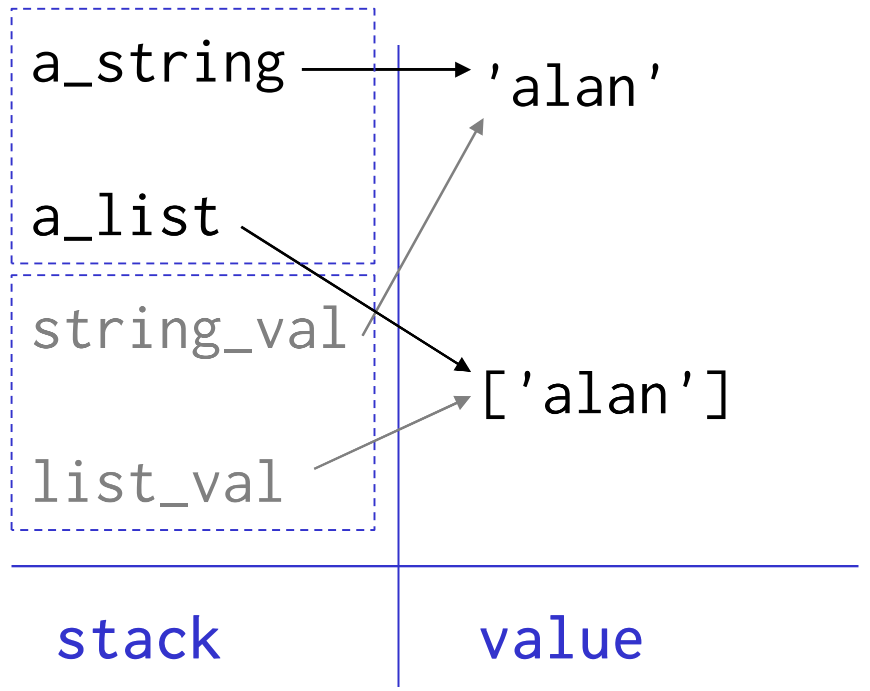
```
def appender(a_string, a_list):
    a_string += 'turing'
    a_list.append('turing')

string_val = 'alan'
list_val = ['alan']
appender(string_val, list_val)
```



Values are copied into parameters

Which means lists are aliased

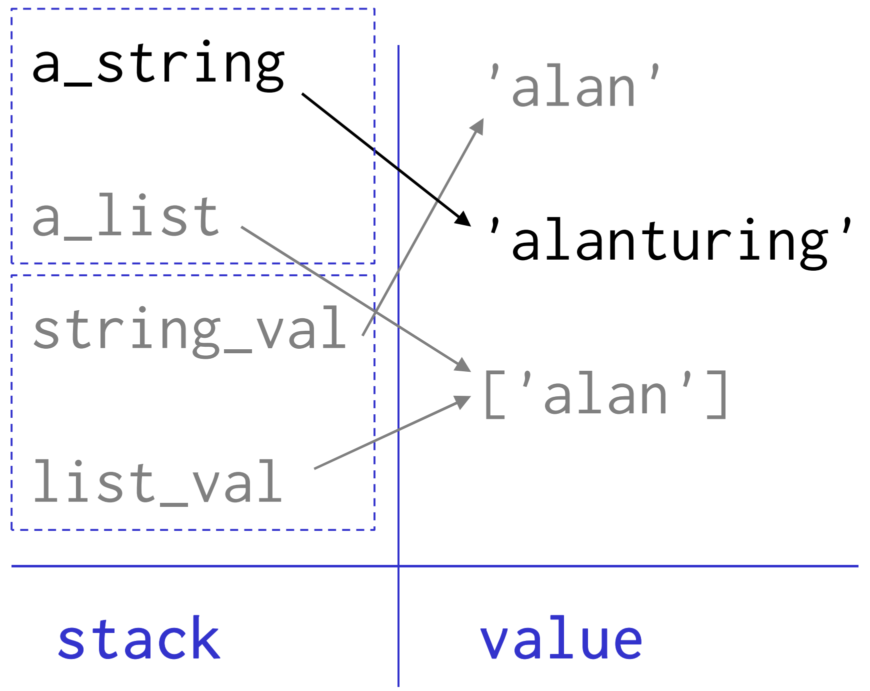
```
def appender(a_string, a_list):
    a_string += 'turing'
    a_list.append('turing')

string_val = 'alan'
list_val = ['alan']
appender(string_val, list_val)
```



Values are copied into parameters

Which means lists are aliased

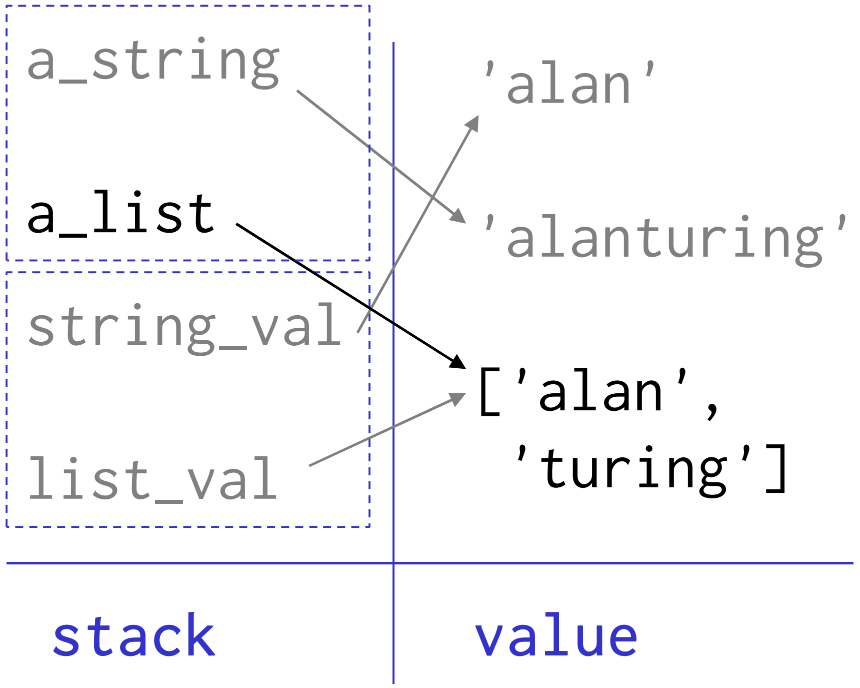
```
def appender(a_string, a_list):
    a_string += 'turing'
    a_list.append('turing')

string_val = 'alan'
list_val = ['alan']
appender(string_val, list_val)
```



Values are copied into parameters

Which means lists are aliased

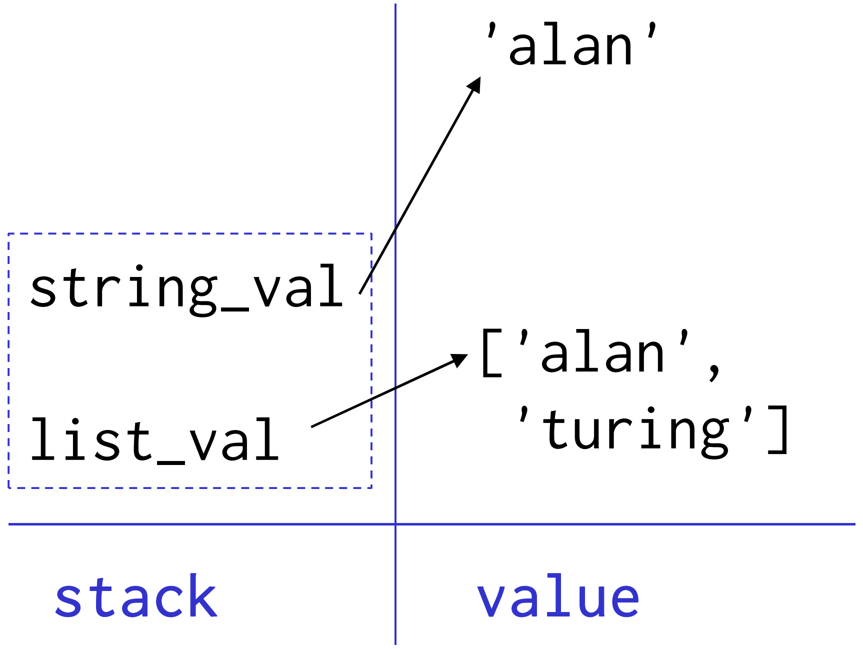
```
def appender(a_string, a_list):
    a_string += 'turing'
    a_list.append('turing')

string_val = 'alan'
list_val = ['alan']
appender(string_val, list_val)
```



Values are copied into parameters

Which means lists are aliased

```
def appender(a_string, a_list):
    a_string += 'turing'
    a_list.append('turing')

string_val = 'alan'
list_val = ['alan']
appender(string_val, list_val)
print string_val
alan
print list_val
['alan', 'turing']
```



Can define *default parameter values*

Can define *default parameter values*

```
def adjust(value, amount=2.0):
  return value * amount
```

Can define *default parameter values*

```
def adjust(value, amount=2.0):
  return value * amount

print adjust(5)
10
```

Can define *default parameter values*

```
def adjust(value, amount=2.0):
  return value * amount

print adjust(5)
10
print adjust(5, 1.001)
5.005
```

More readable than multiple functions

More readable than multiple functions

```
def adjust_general(value, amount):
  return value * amount

def adjust_default(value):
  return adjust_general(value, 2.0)
```

Parameters that have defaults must come *after* parameters that do not

Parameters that have defaults must come *after* parameters that do not

```
def triplet(left='venus', middle, right='mars'):
  return '%s %s %s' % (left, middle, right)
```

Parameters that have defaults must come *after* parameters that do not

```
def triplet(left='venus', middle, right='mars'):
  return '%s %s %s' % (left, middle, right)

print triplet('earth')
venus earth mars
```

OK so far...

Parameters that have defaults must come *after* parameters that do not

```
def triplet(left='venus', middle, right='mars'):
  return '%s %s %s' % (left, middle, right)

print triplet('earth')
venus earth mars

print triplet('pluto', 'earth')
```

OK so far...

?

Parameters that have defaults must come *after* parameters that do not

```
def triplet(left='venus', middle, right='mars'):
  return '%s %s %s' % (left, middle, right)

print triplet('earth')
venus earth mars
```

OK so far...

```
print triplet('pluto', 'earth')
```

?

$\searrow$

```
triplet('pluto', 'earth', 'mars')
```

Parameters that have defaults must come *after* parameters that do not

```
def triplet(left='venus', middle, right='mars'):
  return '%s %s %s' % (left, middle, right)

print triplet('earth')
venus earth mars
```

OK so far...

```
print triplet('pluto', 'earth')
```

?

```
                triplet('pluto', 'earth', 'mars')

          triplet('venus', 'pluto', 'earth')
```

"When should I write a function?"

"When should I write a function?"

Human short term memory can hold 7± 2 items

"When should I write a function?"

Human short term memory can hold 7± 2 items

If someone has to keep more than a dozen things

in their mind at once to understand a block of code,

it's too long

"When should I write a function?"

Human short term memory can hold 7± 2 items

If someone has to keep more than a dozen things

in their mind at once to understand a block of code,

it's too long

Break it into comprehensible pieces with functions

"When should I write a function?"

Human short term memory can hold 7± 2 items

If someone has to keep more than a dozen things

in their mind at once to understand a block of code,

it's too long

Break it into comprehensible pieces with functions

Even if each function is only called once

Example

```
for x in range(1, GRID_WIDTH-1):
  for y in range(1, GRID_HEIGHT-1):
    if (density[x-1][y] > density_threshold) or \
       (density[x+1][y] > density_threshold):
      if (flow[x][y-1] < flow_threshold) or\
         (flow[x][y+1] < flow_threshold):
        temp = (density[x-1][y] + density[x+1][y]) / 2
        if abs(temp - density[x][y]) > update_threshold:
          density[x][y] = temp
```

Refactoring #1: grid interior

```
for x in grid_interior(GRID_WIDTH):
  for y in grid_interior(GRID_HEIGHT):
    if (density[x-1][y] > density_threshold) or \
       (density[x+1][y] > density_threshold):
      if (flow[x][y-1] > flow_threshold) or\
         (flow[x][y+1] > flow_threshold):
        temp = (density[x-1][y] + density[x+1][y]) / 2
        if abs(temp - density[x][y]) > update_threshold:
          density[x][y] = temp
```

Refactoring #2: tests on X and Y axes

```
for x in grid_interior(GRID_WIDTH):
  for y in grid_interior(GRID_HEIGHT):
    if density_exceeds(density, x, y, density_threshold):
      if flow_exceeds(flow, x, y, flow_threshold):
        temp = (density[x-1][y] + density[x+1][y]) / 2
        if abs(temp - density[x][y]) > tolerance:
          density[x][y] = temp
```

Refactoring #3: update rule

```
for x in grid_interior(GRID_WIDTH):
  for y in grid_interior(GRID_HEIGHT):
    if density_exceeds(density, x, y, density_threshold):
      if flow_exceeds(flow, x, y, flow_threshold):
        update_on_tolerance(density, x, y, tolerance)
```

Refactoring #3: update rule

```
for x in grid_interior(GRID_WIDTH):
  for y in grid_interior(GRID_HEIGHT):
    if density_exceeds(density, x, y, density_threshold):
      if flow_exceeds(flow, x, y, flow_threshold):
        update_on_tolerance(density, x, y, tolerance)
```

Good programmers will write this first

Refactoring #3: update rule

```
for x in grid_interior(GRID_WIDTH):
  for y in grid_interior(GRID_HEIGHT):
    if density_exceeds(density, x, y, density_threshold):
      if flow_exceeds(flow, x, y, flow_threshold):
        update_on_tolerance(density, x, y, tolerance)
```

Good programmers will write this first

Then write the functions it implies

Refactoring #3: update rule

```
for x in grid_interior(GRID_WIDTH):
  for y in grid_interior(GRID_HEIGHT):
    if density_exceeds(density, x, y, density_threshold):
      if flow_exceeds(flow, x, y, flow_threshold):
        update_on_tolerance(density, x, y, tolerance)
```

Good programmers will write this first

Then write the functions it implies

Then refactor any overlap

created by

Greg Wilson

October 2010