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`

```python
def greet():
    return 'Good evening, master'
```
Define functions using `def`

```python
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)
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*

```python
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*

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

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

val = 10
```

Python Functions

```
stack
```

```
value
```

10

val
Each function call creates a new *stack frame*

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

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

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

Python Functions
Each function call creates a new stack frame

```python
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*

```python
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*

```python
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*

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

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

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

![Diagram showing stack frame and values](Image)
Each function call creates a new *stack frame*

```python
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
```

 valuation at each stack frame:

- **val** = 10
- result = double(val) returns a stack frame with **result** = 22
- The value was obtained by:
  - **add** receives **val** = 10 and returns **b** = 11
  - **double** receives **add(c)** = 11 and returns **d** = 22

stack:

<table>
<thead>
<tr>
<th>stack level</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
</tr>
</tbody>
</table>

Python functions diagram:

A diagram showing the stack and value at each function level.

- **Stack**:
  - Level 1: Call `double` with `val` = 10
  - Level 2: Call `add` with `a` = 10
- **Value**:
  - **result** = 22
  - **value** = 10
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

```python
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

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

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

'Hello, doctor'

Python Functions
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

```python
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

```python
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

```python
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

```python
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
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

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

```python
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

```python
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

```python
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

```python
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.
Functions and parameters don't have types

```python
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

```python
if type(arg) == int:
    ...
elif type(arg) == str:
    ...
```
Functions and parameters don't have types

```python
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:
    ...
```

Warning sign
Functions and parameters don't have types

```python
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:

```python
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

```python
def appender(a_string, a_list):
a_string += 'turing'
a_list.append('turing')
```
Values are copied into parameters
Which means lists are aliased

```python
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

```python
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

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

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

Strings and lists are aliased when parameters are used in functions in Python.
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*

```python
def adjust(value, amount=2.0):
    return value * amount
```
Can define *default parameter values*

```python
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

```python
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

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

print triplet('earth')  # OK so far...
venus earth mars
```
Parameters that have defaults must come after parameters that do not

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

print triplet('earth')  # OK so far...
venus earth mars

print triplet('pluto', 'earth')  # ?
```
Parameters that have defaults must come after parameters that do not

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

print triplet('earth')  # OK so far...
venus earth mars

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

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

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

print triplet('earth')  # OK so far...
venus earth mars

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 $7 \pm 2$ items
"When should I write a function?"

Human short term memory can hold $7 \pm 2$ 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 $7 \pm 2$ 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 $7 \pm 2$ 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\n               (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

```python
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

```python
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

```python
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

```python
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

```python
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

```python
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

Copyright © Software Carpentry 2010
This work is licensed under the Creative Commons Attribution License
See http://software-carpentry.org/license.html for more information.