Weather forecast in Accra

Thursday		30°C
Friday		31°C
Saturday		29°C
Sunday		28°C

…

$$f = \frac{9}{5} c + 32$$

Temperature in Fahrenheit

Temperature in Celsius

Converting Celsius to Fahrenheit

$$f = \frac{9}{5} c + 32$$

```
tempC = 21
tempF = ((9.0 / 5.0) * tempC) + 32.0
print 'Saturday:', tempF, 'F'
```

```
Saturday: 69.80000000000001 F
```

But we want the whole forecast, not just one day

```
temp_sat_C = 21   # Saturday's forecast in C
temp_sun_C = 19   # Sunday's forecast in C
temp_mon_C = 23   # Monday's forecast in
Ctemp_tues_C = 26  # Tuesday's forecast in C...
```

Converting Celsius to Fahrenheit

Repetitive!

```
temp_sat_C = 21    # Saturday's forecast in C
temp_sun_C = 19    # Sunday's forecast in C
temp_mon_C = 23    # Monday's forecast in C
...
temp_sat_F = ((9.0 / 5.0) * temp_sat_C) + 32.0
print 'Saturday:', temp_sat_F, 'F'
temp_sun_F = ((9.0 / 5.0) * temp_sun_C ) + 32.0
print 'Sunday:', temp_sun_F, 'F'
temp_mon_F= ((9.9 / 5.0) * temp_mon_C ) + 33.0
print 'Monday:', temp_mon_F, 'F'
...
```

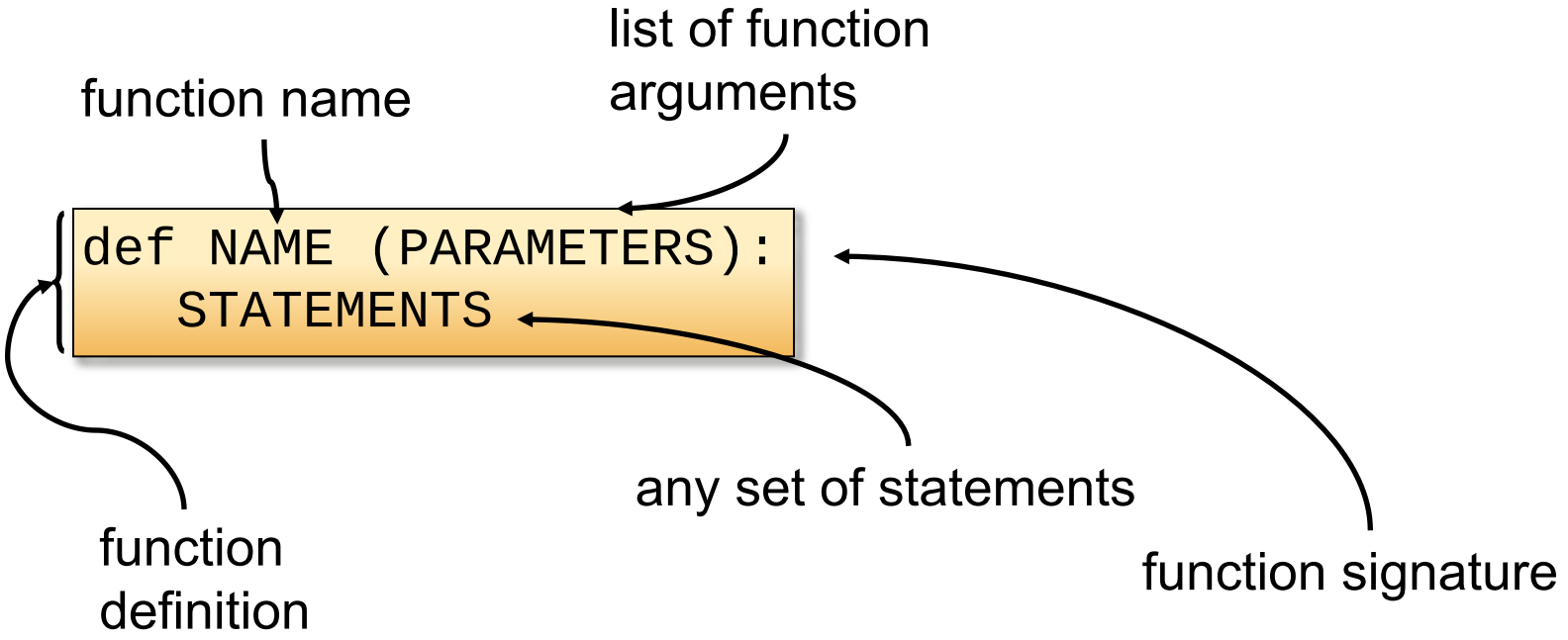
Easy to make mistakes

What if we want to spell out 'Fahrenheit' instead of 'F'?
Must change everywhere!

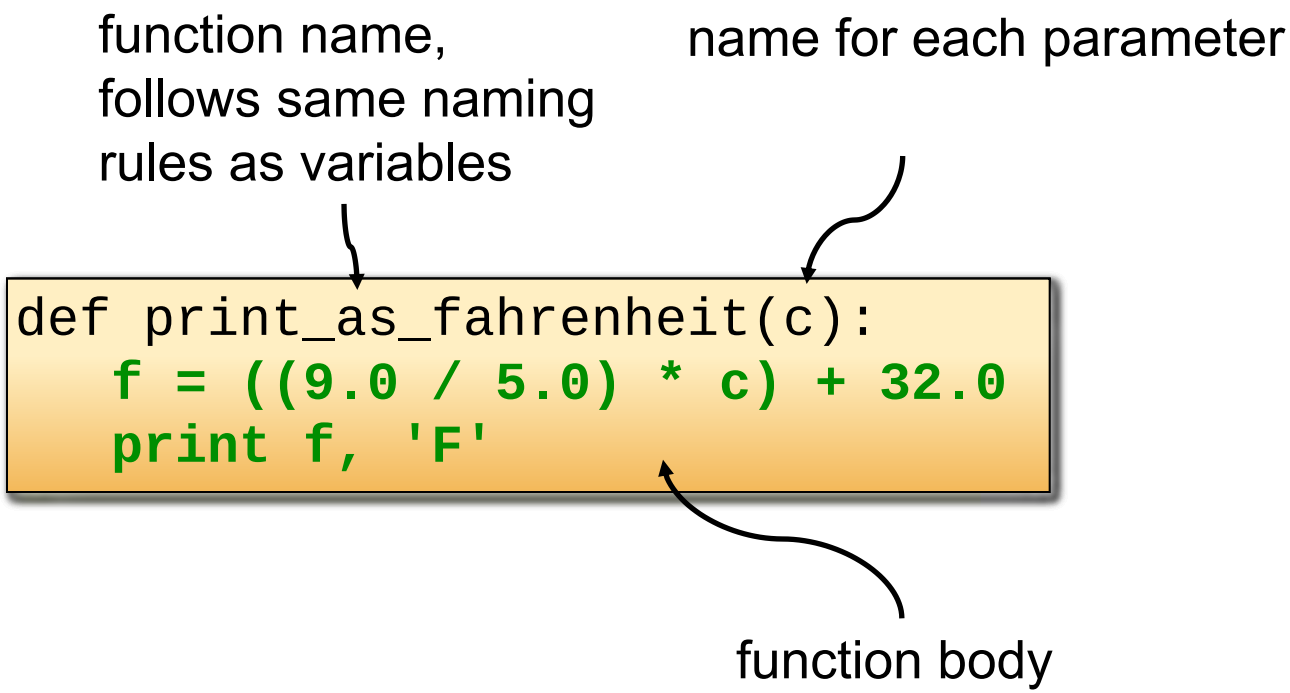
Functions

Functions

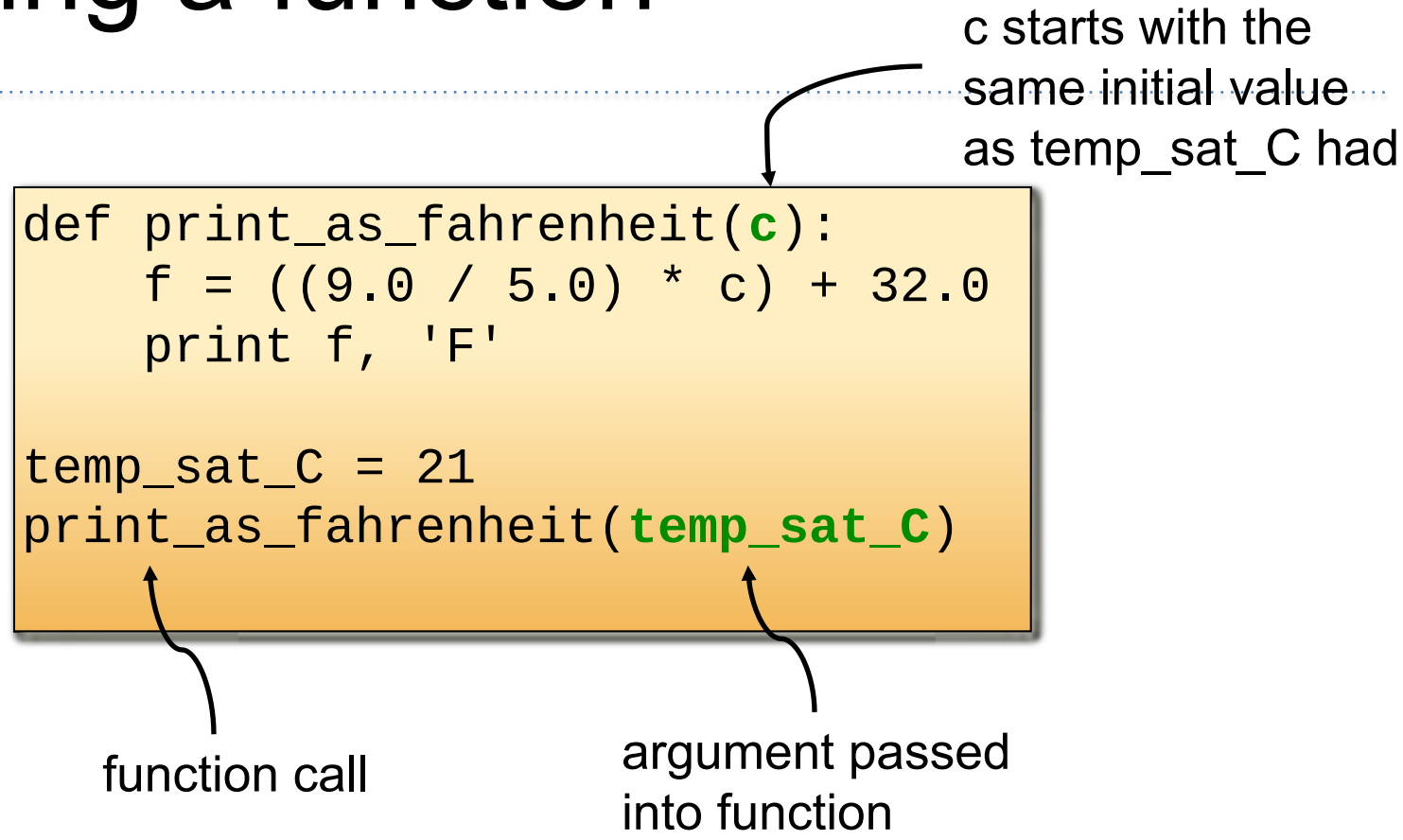
- A function is a sequence of statements that has been given a name.



Defining a function

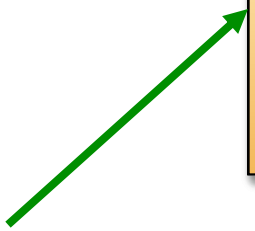


Calling a function



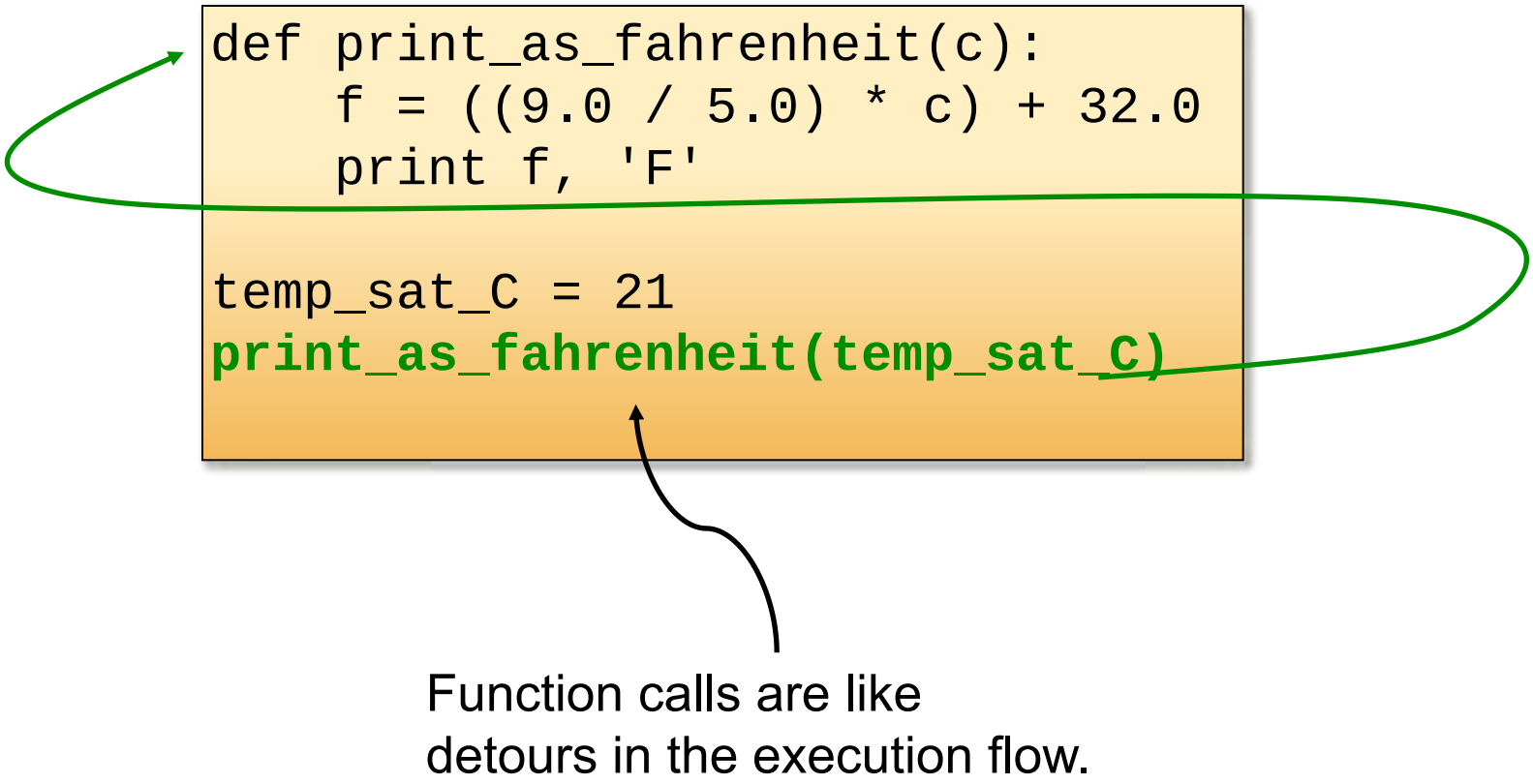
Flow of execution

```
def print_as_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Program execution always starts at the first line that is *not* a statement inside a function

Flow of execution



Flow of execution

```
def print_as_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

Flow of execution

```
def print_as_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

```
69.80000000000001 F
```

Different numbers of parameters

```
def print_as_fahrenheit (c, day):
    f = ((9.0 / 5.0) * c) + 32.0
    print day + ':', f, 'F'
```

```
print_as_fahrenheit(21, 'Saturday')
```

```
Saturday: 69.80000000000001 F
```

```
def print_forecast_intro():
    print 'Welcome to your weather forecast!'
```

```
print_forecast_intro()
```

```
Welcome to your weather forecast!
```

Different numbers of parameters

```
def print_as_fahrenheit (c, day):
    f = ((9.0 / 5.0) * c) + 32.0
    print day + ':', f, 'F'
```

- What happens here?

```
print_as_fahrenheit(21)
```

```
TypeError: print_as_fahrenheit() takes exactly 2 arguments
(1 given)
```

```
print_as_fahrenheit(21, 'Saturday', ‘Sunday’)
```

```
TypeError: print_as_fahrenheit() takes exactly 2 arguments
(3 given)
```

```
print_as_fahrenheit('Saturday', 21)
```

```
TypeError: can't multiply sequence by non-int of type 'float'
```

Returning a value

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f
```


```
return EXPRESSION
```

A return statement ends the function immediately.

any expression, or nothing

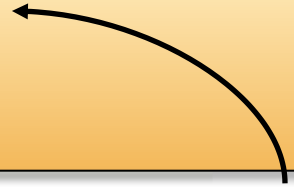
What is the output here?

```
def convert_to_fahrenheit(c):
 print ‘Celsius:’ + c
   f = ((9.0 / 5.0) * c) + 32.0
 return f
 print ‘Fahrenheit:’ + f

convert_to_fahrenheit(27)
```

```
Celsius: 27
```

More than one return statement

```
def absolute_value(c):
    if c < 0:
        return -c
    else:
        return c
```



If c is negative, the function returns here.

More than one return statement

```
def absolute_value(c):
    if c < 0:
        return -c
    return c
```

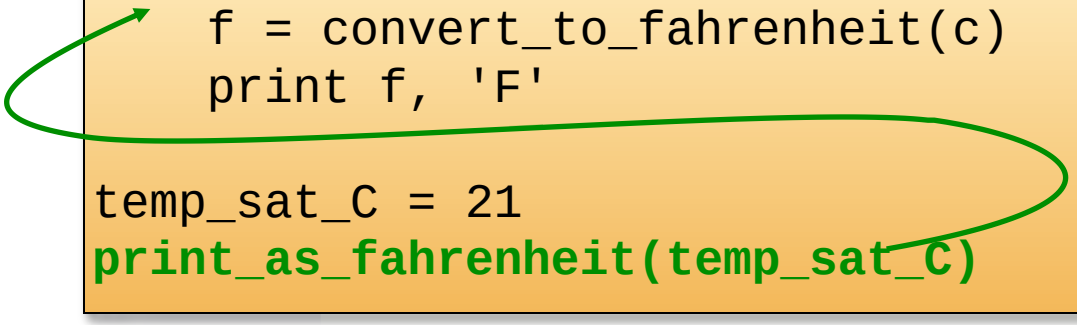
Good rule: Every path through the function must have a return statement.
If you don't add one, Python will add one for you that returns nothing (the value None).

Functions can call functions

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

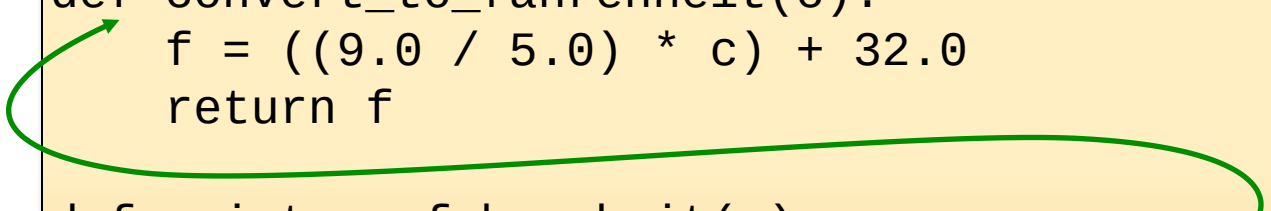
Functions can call functions

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Functions can call functions

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

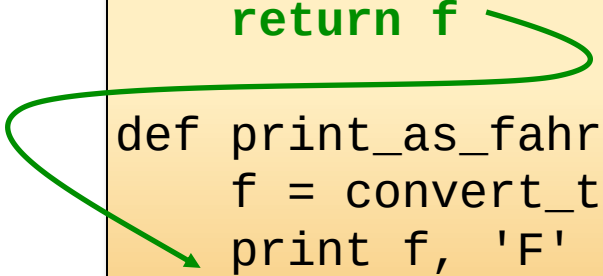


Functions can call functions

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

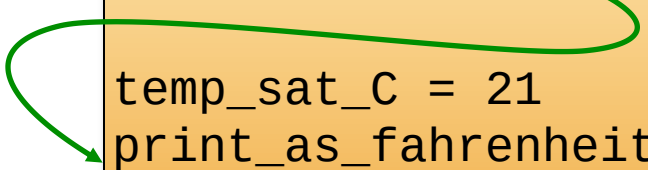
Functions can call functions

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Functions can call functions

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Functions can call functions

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

What is wrong here?

this function has to be defined before it is called

```
NameError: name
'print_as_fahrenheit'
is not defined
```

```
temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f
```

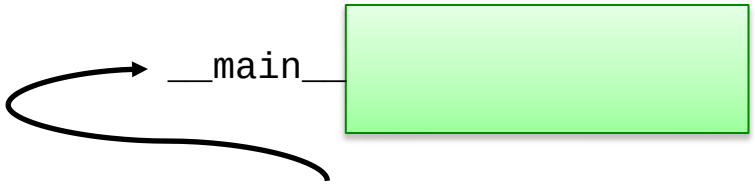
what about this one?

The two functions are in the same level. Therefore, one function can call the other functions even if it is defined after the calling function.

Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Each function call gets its own stack frame.

28

Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

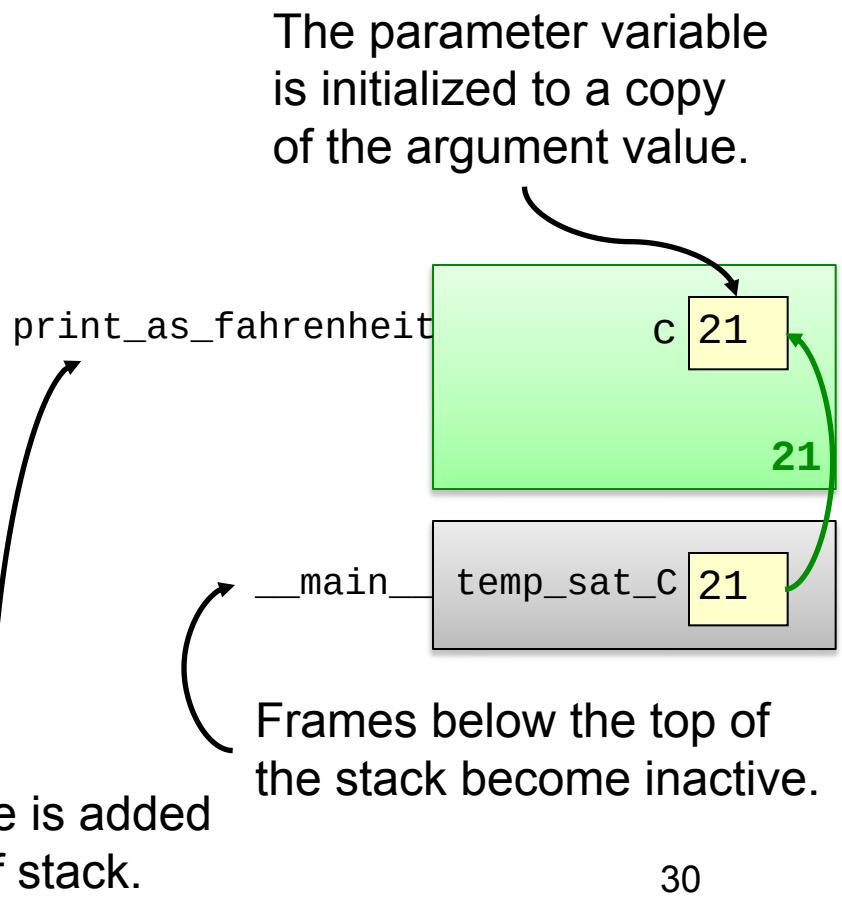
```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

29

Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



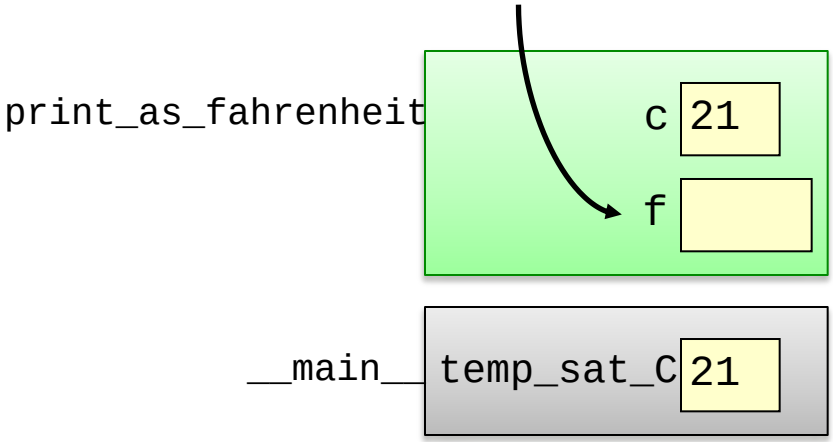
30

Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

Variables defined inside the function are called local variables.

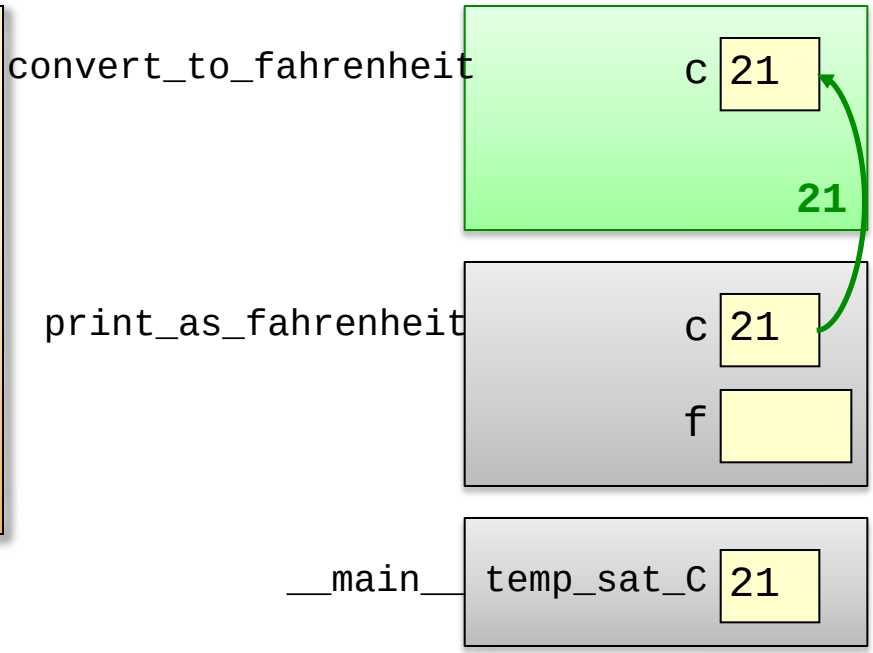


31

Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

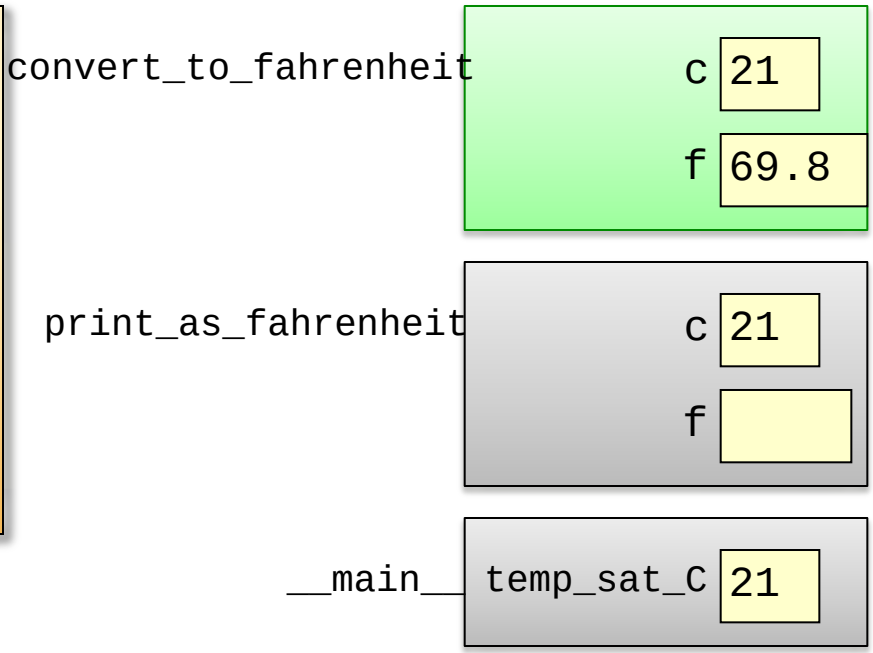


32

Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



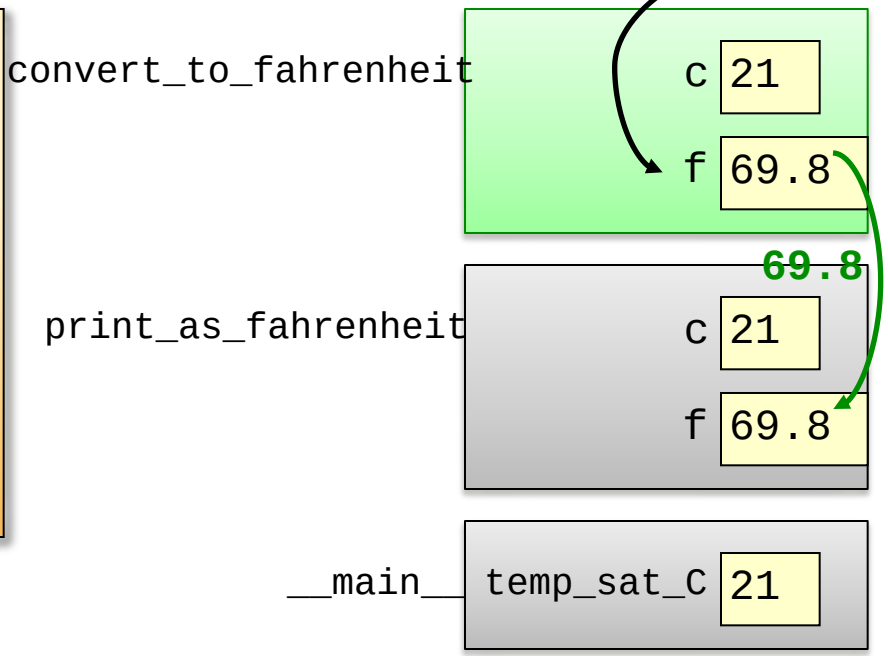
33

Scoping in functions

The return value is passed back to the function's caller.

- A stack diagram keeps track of where each variable is defined, and its value.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



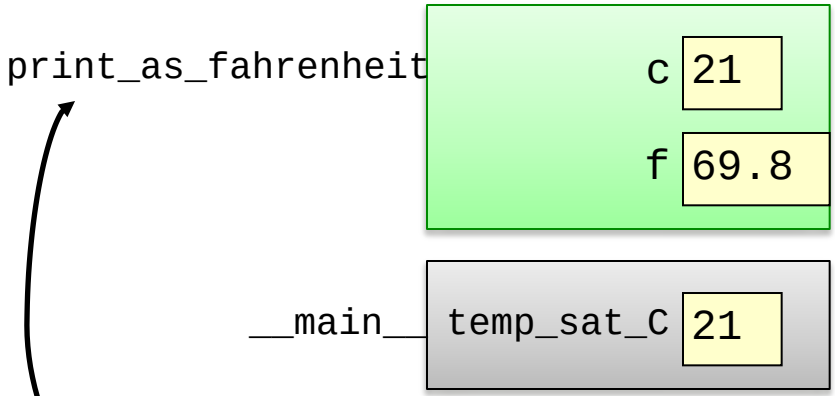
34

Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

When a function returns, its stack frame is popped off the stack.



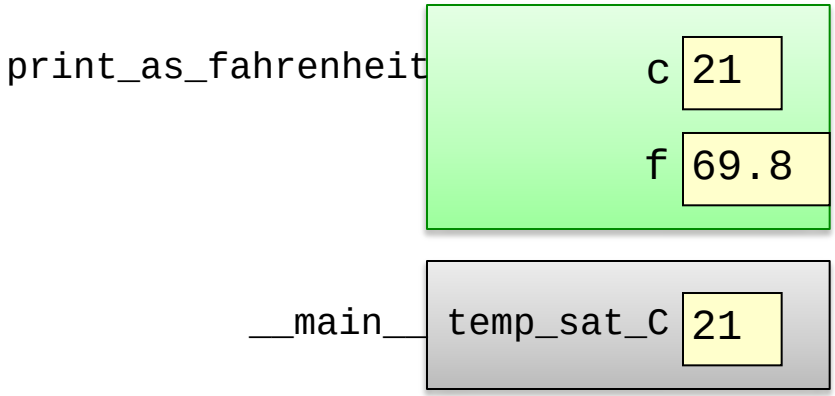
The stack frame for the calling function is now active again.

35

Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



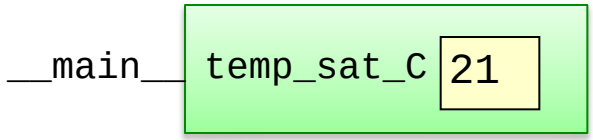

```
69.8 F
```


36

Scoping in functions

- A stack diagram keeps track of where each variable is defined, and its value.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```

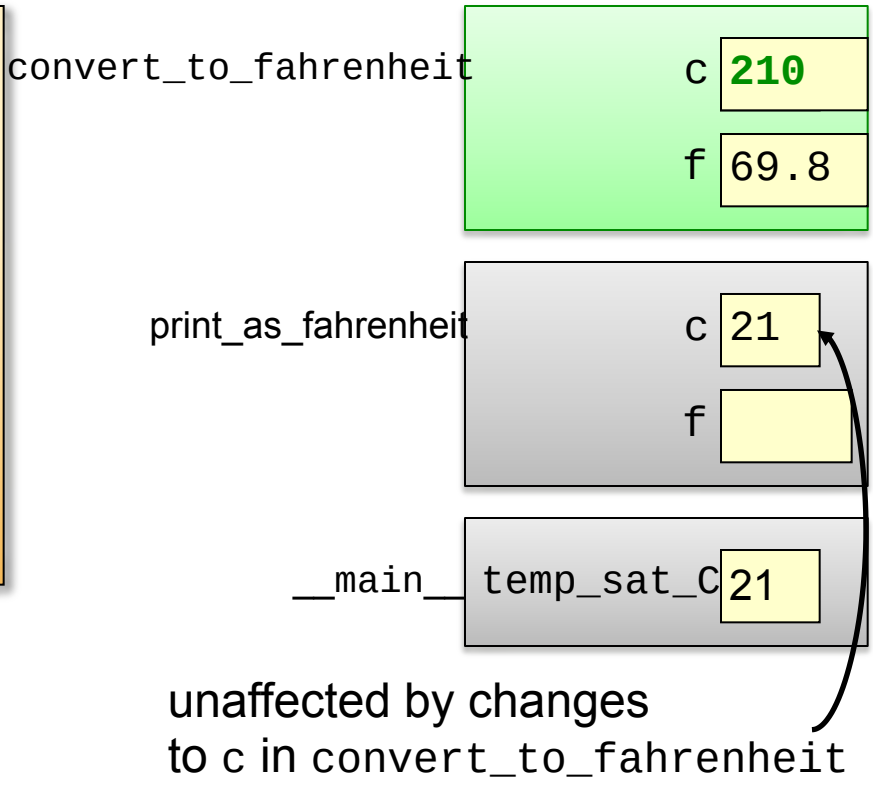


37

Tricky issues with scoping

- Changes to a variable in the current scope do not affect variables in other scopes.

```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    c = c * 10
    return f

def print_as_fahrenheit(c):
    f = convert_to_fahrenheit(c)
    print f, 'F'

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



38

Why use functions?

- Generalization: the same code can be used more than once, with parameters to allow for differences.

BEFORE

```
temp_sat_F = ((9.0 / 5.0) * 21) + 32.0
print 'Saturday:', temp_sat_F, 'F'

temp_sun_F = ((9.0 / 5.0) * 19) + 32.0
print 'Sunday:', temp_sun_F, 'F'

temp_mon_F = ((9.9 / 5.0) * 23) + 33.0
print 'Monday:', temp_mon_F, 'F'
```

Would not have made this typo.

AFTER

```
def print_as_fahrenheit(c, day):
    f = ((9.0 / 5.0) * c) + 32.0
    print day + ':', f, 'F'

print_as_fahrenheit(21, 'Saturday')
print_as_fahrenheit(19, 'Sunday')
print_as_fahrenheit(23, 'Monday')
```

Only type these lines once.

39

Why use functions?

- Maintenance: much easier to make changes.

BEFORE

```
temp_sat_F = ((9.0 / 5.0) * 21) + 32.0
print 'Saturday:', temp_sat_F, 'F'

temp_sun_F = ((9.0 / 5.0) * 19) + 32.0
print 'Sunday:', temp_sun_F, 'F'

temp_mon_F = ((9.9 / 5.0) * 23) + 33.0
print 'Monday:', temp_mon_F, 'F'
```

AFTER

```
def print_as_fahrenheit(c, day):
    f = ((9.0 / 5.0) * c) + 32.0
    print day + ':', f, 'F'

print_as_fahrenheit(21, 'Saturday')
print_as_fahrenheit(19, 'Sunday')
print_as_fahrenheit(23, 'Monday')
```

Can change to "Fahrenheit" with only one change.

40

Why use functions?

- Encapsulation: much easier to read and debug!

BEFORE

```
temp_sat_F = ((9.0 / 5.0) * 21) + 32.0
print 'Saturday:', temp_sat_F, 'F'

temp_sun_F = ((9.0 / 5.0) * 19) + 32.0
print 'Sunday:', temp_sun_F, 'F'

temp_mon_F = ((9.9 / 5.0) * 23) + 33.0
print 'Monday:', temp_mon_F, 'F'
```

What are we doing here?

We're printing as Fahrenheit!

AFTER

```
def print_as_fahrenheit(c, day):
    f = ((9.0 / 5.0) * c) + 32.0
    print day + ':', f, 'F'

print_as_fahrenheit(21, 'Saturday')
print_as_fahrenheit(19, 'Sunday')
print_as_fahrenheit(23, 'Monday')
```


41

Method overloading

We want to return the sum of two or three variables.

Java makes this difficult.

```
//In Java:
Private int add(int a, int b)
{
    return a+b
}

Private int add(int a, int b, int c)
{
    return a+b+c
}

Private double add(double a, double b)
{
    return a+b+c
}

//Etc…
//repeat for float, byte, short, double,
int, and every combination of 2 or 3
```

Method Overloading

Python makes this easy.

```
def add(a, b, c=0)
  return a+b+c
```

```
#This doesn't work:
def add(a, b):
    return a+b
def add(a, b, c):
    return a+b+c:
```