Functions

WTP 2010
Computer Science
Day 4: Thursday, July 1

Weather forecast in Paris

Saturday	21°C
Sunday	19°C
Monday	23°C
Tuesday	26°C

…

$$f = {}^{9}/_{5}\, c + 32$$

temperature in Fahrenheit

temperature in Celsius

Converting Celsius to Fahrenheit

$$f = {}^{9}/_{5}\, c + 32$$

```
tempC = 21
tempF = ((9.0 / 5.0) * tempC) + 32.0
print 'Saturday:', tempF, '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 C
temp_tues_C = 26  # Tuesday's forecast in C
...
```


Converting Celsius to Fahrenheit

repetitive!

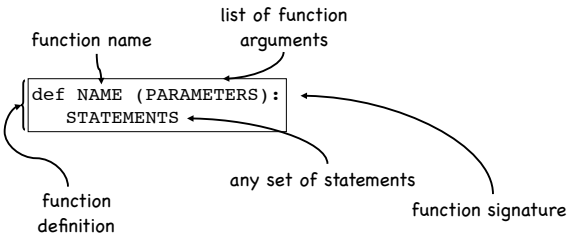
```
temp_sat_C = 21   # Saturday's forecast in C
temp_sun_C = 19   # Sunday's forecast in C
temp_mon_C = 23   # Monday's forecast in C
...

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'
...
```

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

easy to make mistakes

Functions

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



Defining a function

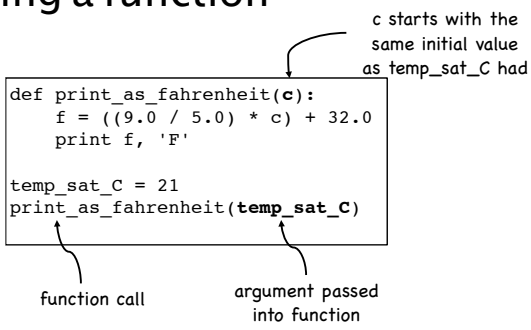
function name, follows same naming rules as variables

name for each parameter

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

function body

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

```
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)
```

Function calls are like detours in the execution flow.

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
```


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')
```

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

```
print_forecast_intro()
```


Returning a value

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

return statement

```
return EXPRESSION
```

A return statement ends the function immediately.

any expression, or nothing

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)
```

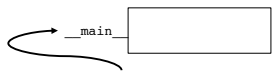

Functions can call functions

```
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)
```


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)
```

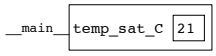


Each function call gets its own **stack frame**.

Scoping in functions

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

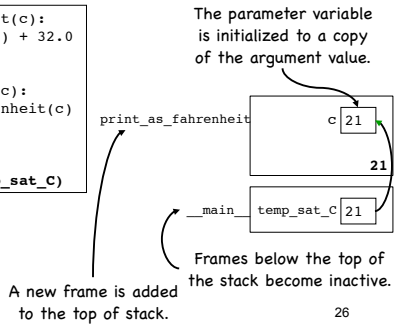
```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

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

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Scoping in functions

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

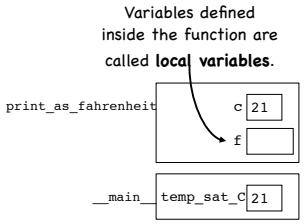
```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

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

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Scoping in functions

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

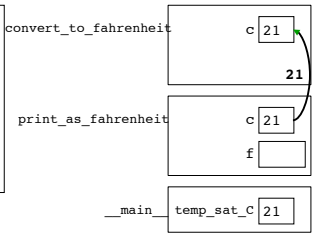
```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

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

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Scoping in functions

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

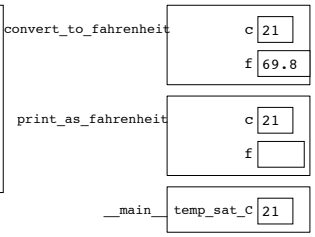
```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

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

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



Scoping in functions

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

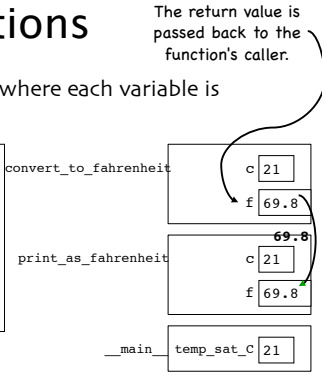
```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

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

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



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)
```

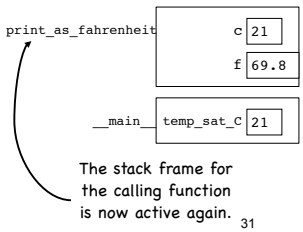


Scoping in functions

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.

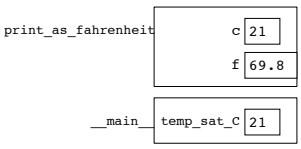


The stack frame for the calling function is now active again.

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)
```

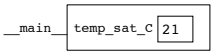



```
69.8 F
```


Scoping in functions

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

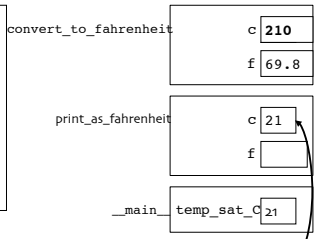
```
def convert_to_fahrenheit(c):
    f = ((9.0 / 5.0) * c) + 32.0
    return f

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

temp_sat_C = 21
print_as_fahrenheit(temp_sat_C)
```



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)
```



unaffected by changes to c in convert_to_fahrenheit

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.

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'
```

Can change to "Fahrenheit" with only one change.

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')
```

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?

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')
```

Oh, printing as Fahrenheit!

Today's exercises

- Writing and using functions
- Following function execution flow
- Math operations
- Fancy formatting
- Readings for tomorrow: Sections 8.6-10, 10.6-9, 11.1-4