Announcements

- Homework
  - I'm still grading HW3, sorry. Hoping to get scores send out by tonight
  - HW4 has been posted. You should be able to start looking at Problem 3 after today.
- Polling: `rembold-class.ddns.net`

Review Question

How would the fraction $\frac{9}{16}$ be represented in binary floating point?

A) (1001, -100)

B) (111, -101)

C) (1111, -111)

D) (1100, 101)

Solution: (1001, -100)

The Story So Far

- We've already looked at
  - numbers
  - variable assignments
  - input and output
  - comparisons
  - loops
- This enough to be Turing complete!
- What more do we want?
  - Improved usability!
    - Ability to reuse code
    - Ability to generalize code

Function Objectives

- Already familiar with a small host of built-in functions:
  - `print`
  - `abs`
  - `len`
  - etc

Function Objectives

- Already familiar with a small host of built-in functions:
  - `print`
  - `abs`
  - `len`
  - etc
- You don't care what code happens when you run those, you care that they perform a particular task for you

Function Objectives

- Already familiar with a small host of built-in functions:
  - `print`
  - `abs`
  - `len`
  - etc
- You don't care what code happens when you run those, you care that they perform a particular task for you
- You pass them some input (or inputs) and expect the expression to return some output (or outputs)
  - Looks like: `<output> = function(<input>)`
  - `5 == abs(-5)`
  - `11 == len('long string')`

Function Objectives

- Already familiar with a small host of built-in functions:
  - `print`
  - `abs`
  - `len`
  - etc
- You don't care what code happens when you run those, you care that they perform a particular task for you
- You pass them some input (or inputs) and expect the expression to return some output (or outputs)
  - Looks like: `<output> = function(<input>)`
  - `5 == abs(-5)`
  - `11 == len('long string')`
- Would be very useful to be able to define our own functions

Defining Functions

- We can define a new function like:

```
def <name of function> (<list of formal parameters>):
    <body of function>
```

Defining Functions

- We can define a new function like:

```
def <name of function> (<list of formal parameters>):
    <body of function>
```

- Syntax: Starts with `def`, then function name, then parenthesis, then colon

Defining Functions

- We can define a new function like:

```
def <name of function> (<list of formal parameters>):
    <body of function>
```

- Syntax: Starts with `def`, then function name, then parenthesis, then colon
- Pieces:

Defining Functions

- We can define a new function like:

```
def <name of function> (<list of formal parameters>):
    <body of function>
```

- Syntax: Starts with `def`, then function name, then parenthesis, then colon
- Pieces:
  - name of function: mandatory – what you want the function to be called

Defining Functions

- We can define a new function like:

```
def <name of function> (<list of formal parameters>):
    <body of function>
```

- Syntax: Starts with `def`, then function name, then parenthesis, then colon
- Pieces:
  - name of function: mandatory – what you want the function to be called
  - list of formal parameters: optional – potential parameters or "inputs" you want to pass to the function

Defining Functions

- We can define a new function like:

```
def <name of function> (<list of formal parameters>):
    <body of function>
```

- Syntax: Starts with `def`, then function name, then parenthesis, then colon
- Pieces:
  - name of function: mandatory – what you want the function to be called
  - list of formal parameters: optional – potential parameters or "inputs" you want to pass to the function
  - body of function: mandatory – whatever you want the function to do. May or may not include a `return` statement

Varied Uses

- As a way to group code

```
def print_stuff():
    print('Stuff 1')
    print('Stuff 2')
```

Varied Uses

- As a way to group code

```
def print_stuff():
    print('Stuff 1')
    print('Stuff 2')
```

- As a way to provide input to a group of code

```
def print_mult_of_A(A):
    print(A, 2*A, 3*A, 4*A, 5*A)
```

Varied Uses

- As a way to group code

```
def print_stuff():
    print('Stuff 1')
    print('Stuff 2')
```

- As a way to provide input to a group of code

```
def print_mult_of_A(A):
    print(A, 2*A, 3*A, 4*A, 5*A)
```

- As a machine that takes some input and provides some output for later use

```
def calc_year_savings(init_amt):
    for i in range(12):
        init_amt += init_amt*0.03/12
    return init_amt
```

Functional Vocab

Formal Parameters: The variable names used in the defining of the function which will be assigned to actual literals upon function call

Actual Parameters or Arguments: The numbers or other literals passed to the function which are assigned the formal parameter variable names

Function Definition: Where the function is actually defined, in terms of formal parameters

Function Call (or Invocation): Where the function is used in the code, with accompanying arguments passed to the function.

The Flow

```
1  def year_savings(init_amt, int_rate):
2      for i in range(12):
3          init_amt *= (1+int_rate/12)
4      return init_amt
5
6  amount = 100
7  rate = 0
8  while rate < 1:
9      print("Savings:",
10         year_savings(amount, rate))
11     rate += 0.05
12
13 print('All done!')
```

- Function definition with formal parameters (1)

The Flow

```
1  def year_savings(init_amt, int_rate):
2      for i in range(12):
3          init_amt *= (1+int_rate/12)
4      return init_amt
5
6  amount = 100
7  rate = 0
8  while rate < 1:
9      print("Savings:",
10         year_savings(amount, rate))
11     rate += 0.05
12
13 print('All done!')
```

- Normal code (6-9)

The Flow

```
1  def year_savings(init_amt, int_rate):
2      for i in range(12):
3          init_amt *= (1+int_rate/12)
4      return init_amt
5
6  amount = 100
7  rate = 0
8  while rate < 1:
9      print("Savings:",
10         year_savings(amount, rate))
11     rate += 0.05
12
13 print('All done!')
```

- Function call with current `amount` and `rate` bound to formal parameters (9-10)
  - Iterate across 12 months calculating compound interest (2-3)
  - Return final value (4)

The Flow

```
1  def year_savings(init_amt, int_rate):
2      for i in range(12):
3          init_amt *= (1+int_rate/12)
4      return init_amt
5
6  amount = 100
7  rate = 0
8  while rate < 1:
9      print("Savings:",
10         year_savings(amount, rate))
11     rate += 0.05
12
13 print('All done!')
```

- Resume main code block and increase rate (11)

The Flow

```
1  def year_savings(init_amt, int_rate):
2      for i in range(12):
3          init_amt *= (1+int_rate/12)
4      return init_amt
5
6  amount = 100
7  rate = 0
8  while rate < 1:
9      print("Savings:",
10         year_savings(amount, rate))
11     rate += 0.05
12
13 print('All done!')
```

- Function definition with formal parameters (1)
- Normal code (6-9)
- Function call with current `amount` and `rate` bound to formal parameters (9-10)
  - Iterate across 12 months calculating compound interest (2-3)
  - Return final value (4)
- Resume main code block and increase rate (11)

Common Mistakes

- Make sure you have parentheses in your function definition, even if you are not supplying any parameters!

Common Mistakes

- Make sure you have parentheses in your function definition, even if you are not supplying any parameters!
- That colon and indenting your function code block are still mandatory, just like with `if` statements or loops

Common Mistakes

- Make sure you have parentheses in your function definition, even if you are not supplying any parameters!
- That colon and indenting your function code block are still mandatory, just like with `if` statements or loops
- You do not need a `return` statement, the function will automatically return `None` if it reaches the end of the function code without encountering one

Common Mistakes

- Make sure you have parentheses in your function definition, even if you are not supplying any parameters!
- That colon and indenting your function code block are still mandatory, just like with `if` statements or loops
- You do not need a `return` statement, the function will automatically return `None` if it reaches the end of the function code without encountering one
- `return` statements will break out of the function code as soon as they are called, rejoining the main code flow.

Common Mistakes

- Make sure you have parentheses in your function definition, even if you are not supplying any parameters!
- That colon and indenting your function code block are still mandatory, just like with `if` statements or loops
- You do not need a `return` statement, the function will automatically return `None` if it reaches the end of the function code without encountering one
- `return` statements will break out of the function code as soon as they are called, rejoining the main code flow.
  - You can't return something midway through a function and then return more things later. It all has to be at once.

Common Mistakes

- Make sure you have parentheses in your function definition, even if you are not supplying any parameters!
- That colon and indenting your function code block are still mandatory, just like with `if` statements or loops
- You do not need a `return` statement, the function will automatically return `None` if it reaches the end of the function code without encountering one
- `return` statements will break out of the function code as soon as they are called, rejoining the main code flow.
  - You can't return something midway through a function and then return more things later. It all has to be at once.
  - You can use this in certain situations as a clever `break` statement, getting you out of a loop earlier than you otherwise would

Example: Summing Primes

Example

Let's look at the problem where we'd like to find all the positive pairs of prime numbers that sum to 100. Here we'll define a function called `is_prime` to help us out and greatly simplify our code.