Valued functions

Valued functions

So far all functions we designed output the results of computation via print statements:

```
def foo(<parameters>):
  <statements>
  print ...
```

However, functions can also *return* (and thus represent) values.

Valued functions

This function `max` *returns* the maximum of two given numbers:

```
def max(x, y):
  if x > y: return x
  else: return y
```

Here, the maximum value will be *returned* to the caller.

Valued functions

When we call a function that returns a value, a function call is an *expression* representing a *value* being returned.

Example.

`max(3, 4)` is an expression representing a value returned by the function `max` with arguments `3` and `4` (that is, `4`).

Valued functions

What would this output?

```
def max(x, y):
  if x > y: return x
  else: return y

print max(3, 4)
```

Output: `4`

Valued functions

What would this output?

```
def max(x, y):
  if x > y: return x
  else: return y

print max(3, 4) + max(5, 6)
```

Output: `10`

Valued functions

What would this output?

```
def max(x, y):
  if x > y: return x
  else: return y

print max(3, max(4, 5))
```

Output: `5`

Valued functions

This function `is_even` returns a boolean value: `True` or `False`.

```
def is_even(n):
  if n % 2 == 0: return True
  else: return False
```

Here, `x % y` gives the remainder of `x / y`.

Valued functions

What would this output?

```
def is_even(n):
  if n % 2 == 0: return True
  else: return False

print is_even(1823)
```

Output: `False`

Valued functions

What would this output?

```
def max(x, y):
  if x > y: return x
  else: return y

def is_even(n):
  if n % 2 == 0: return True
  else: return False

print is_even(max(3, 4))
```

Output: `True`

Valued functions

This boolean-valued function can be used in an `if` statement. What would this output?

```
def is_even(n):
  if n % 2 == 0: return True
  else: return False

n = input('Enter a number: ')
if is_even(n):
  print 'You entered an even number.'
else:
  print 'You entered an odd number.'
```


Incremental development

Incremental development

Complex code should be developed *incrementally*, adding and testing only a small amount of code at a time.

Incremental development

Example. A function to return the average of three numbers.

Step 1. Start with correct syntax.

```
def average(n1, n2, n3):
  return 0.0

print average(1, 2, 4)  # Always returns 0
```


Incremental development

Step 2. Compute the sum and print it.

```
def average(n1, n2, n3):
  sum = n1 + n2 + n3
  print 'sum of', n1, ',', n2, ',', n3, 'is', sum
  return 0.0

print average(1, 2, 4)  # Is the sum correct?
```

Incremental development

Step 3. Compute the average and print it.

```
def average(n1, n2, n3):
  sum = n1 + n2 + n3
  print 'Sum of', n1, ',', n2, ',', n3, 'is', sum
  average = sum / 3
  print 'Average is', average
  return 0.0

print average(1, 2, 4)  # Is the average correct?
```


Incremental development

Step 4. Fix a bug.

```
def average(n1, n2, n3):
  sum = n1 + n2 + n3
  print 'Sum of', n1, ',', n2, ',', n3, 'is', sum
  average = sum / 3.0
  print 'Average is', average
  return 0.0

print average(1, 2, 4)  # The average is correct!
```


Incremental development

Step 5. Return the average.

```
def average(n1, n2, n3):
  sum = n1 + n2 + n3
  # print 'Sum of', n1, ',', n2, ',', n3, 'is', sum
  average = sum / 3.0
  # print 'Average is', average
  return average

print average(1, 2, 4)  # The average is correct!
```

Incremental development

Step 6. Clean up and document.

```
def average(n1, n2, n3):
  "Returns the average of its 3 parameters."
  sum = n1 + n2 + n3
  average = sum / 3.0
  return average

print average(1, 2, 4)  # The average is correct!
```


Incremental development

- Add code in small increments.
- Run your partial solution.
- Test and verify before adding more code.
- Use local variables to print partial results.
- Remove scaffolding and clean up code.
- Document your code.

Recursive valued-functions

Many recursive valued-functions also have the same basic structure.

```
def roo(<parameters>):
  if <boolean expression>:
    <base-case statements>
    <recursive statements>
```

Recursive valued-functions

Many recursive valued-functions also have the same basic structure.

```
def roo(<parameters>):
  if <boolean expression>:
    <base-case statements>
    <recursive statements>
```

Base-case statements must be *non-recursive*, i.e., it must not refer to the function itself.

Recursive valued-functions

Many recursive valued-functions also have the same basic structure.

```
def roo(<parameters>):
  if <boolean expression>:
    <base-case statements>
    <recursive statements>
```

Recursive statements refer to the function itself, but it must approach to the base case, typically involving "smaller" arguments.