Functions

Functions

A ***function*** is a subprogram—a small program inside a program—with a name.

- Can be used anywhere anytime by calling its name.
- Most useful when we want to repeat the same basic task (e.g., rounding a real number to an integer).

Why functions?

- Make easier to read and debug.
- Make programs shorter by eliminating repetitive code; if you make a change, you only have to make it in one place.
- Can be used for other purposes in the future…

Functions

Example. Suppose you want to write a program to print out these famous lyrics:

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear <someone>.
Happy birthday to you!
```


Functions

To save repetition, we can define a function for printing "Happy birthday to you!" in the following way.

```
def happy():
  print 'Happy birthday to you!'
```

This function is named `happy`.

Functions

To *call* or *invoke* this function `happy`, all you need to do is:

```
>>> happy()
Happy birthday to you!
```

Functions

How can we make a generic function to sing happy birthday to anyone?

We use a *parameter*—a variable whose value is specified by the caller (or user).

Functions

Consider this function for anyone:

```
def sing(anyone):
  happy()
  happy()
  print 'Happy birthday, dear', anyone + '.'
  happy()
```

Here, the value of anyone is to be specified by the caller of this function.

Functions

Invoking the function `sing` with `'Joe'` gives:

```
>> sing('Joe')
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Joe.
Happy birthday to you!
```

Here, `'Joe'` is passed as an argument to `sing`'s parameter `anyone`.

Functions

Invoking the function `sing` with `'Ann'` gives:

```
>> sing('Ann')
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Ann.
Happy birthday to you!
```

Here, `'Ann'` is passed as an argument to `sing`'s parameter `anyone`.

Parameters and arguments

A variable inside a function whose value is specified by a caller is called a *parameter*.

A value being passed by a caller to a parameter is called an *argument*.

A parameter is always a *variable*.

An argument is always a *value*.

Parameters and arguments

Consider this very simple function.

```
def foo(x):
  print x
  print x
```

What does this do?

This function simply prints `x` twice.

Parameters and arguments

What does this do?

```
>>> def foo(x):
      print x
      print x
>>> foo(1823)
1823
1823
```


Parameters and arguments

What does this do?

```
>>> def foo(x):
      print x
      print x
>>> foo('Trinity')
Trinity
Trinity
```


Parameters and arguments

What does this do?

```
>>> def foo(x):
      print x
      print x
>>> n = 3
>>> foo(n+4)
7
7
```

Parameters and arguments

What does this do?

```
>>> def foo(x):
      print x
      print x
>>> foo('Trin'*2)
TrinTrin
TrinTrin
```


Parameters and arguments

If a function has *n* parameters, a caller must pass *n* arguments.

Arguments are assigned to parameters according to the order alone.

Parameters and arguments

Consider this very simple function.

```
def goo(x, y):
  print '1st value:', x
  print '2nd value:', y
```

What does this do?

This function simply prints the values of the parameters `x` and `y`.

Parameters and arguments

What does this do?

```
>>> def goo(x, y):
      print '1st value:', x
      print '2nd value:', y
>>> goo(3, 4)
1st value: 3
2nd value: 4
```


Parameters and arguments

What does this do?

```
>>> def goo(x, y):
      print '1st value:', x
      print '2nd value:', y
>>> a = 5
>>> b = 6
>>> goo(a, b)
1st value: 5
2nd value: 6
```


Parameters and arguments

What does this do?

```
>>> def goo(x, y):
      print '1st value:', x
      print '2nd value:', y
>>> x = 7
>>> y = 8
>>> goo(y, x)
1st value: 8
2nd value: 7
```

Variables and parameters

Variables and parameters defined inside a function are *local*: they can only be used inside the function.

Parameters and arguments

What does this do?

```
>>> def hoo(x, y):
      n = x
      print '1st value:', n
      print '2nd value:', y
>>> n = 1
>>> y = 2
>>> hoo(y, n)
1st value: 2
2nd value: 1
```


Simple repetition

Consider this very simple function.

```
def twice(x):
  print x
  print x
```

This function simply prints `x` twice.

Can we somehow generalize this function so that, for any given integer `n`*, it prints* `x` *for* `n` *times?*

Simple repetition

We want something like this:

```
>>> n_times('Trinity', 2)
Trinity
Trinity
>>> n_times('Trinity', 4)
Trinity
Trinity
Trinity
Trinity
```


Simple repetition

This function should look like:

```
def n_times(x, n):
  somehow repeat printing x for n times.
```


Simple repetition

This function should look like:

```
def n_times(x, n):
  print x
  print x
    .
    .
    .
  print x
```

Can we use . . . in Python?

Simple repetition

Loop constructs allow a program to repeat executing sequences of instructions.

The most basic form of loop construct is the *for* statement:

```
for i in range(n):
  <statements>
```

Python repeats `<statements>` `n` times.

Simple repetition

How should we implement this in Python?

```
def n_times(x, n):
  somehow repeat printing x for n times.
```

Here is a solution:

```
def n_times(x, n):
  for i in range(n):
    print x
```


Simple repetition

What does this do?

```
>>> def ioo(n):
      for i in range(n):
        print i
>>> ioo(4)
0
1
2
3
```

Simple repetition

The most basic form of loop construct is the *for* statement:

```
for i in range(n):
  <statements>
```

Python repeats `<statements>` `n` times.

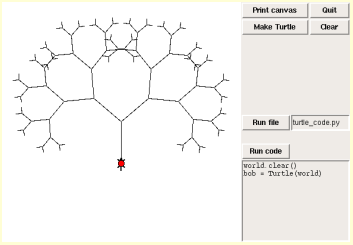
Simple repetition

The most basic form of loop construct is the *for* statement:

```
for i in range(n):
  <statements>
```

As Python repeats `<statements>`, the value of `i` actually grows from `0` through `n-1`.

This week's laboratory



Read Chapter 4 before coming to the lab.