Functions

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

- Can be used anywhere anytime by calling its name.
- Performs some useful task (e.g., rounding a real number to an integer).

Functions

Function Calls

You can ***call a function*** by its name – e.g, ***type*** is a function. The term in parens is called an ***argument***.

```
>>> type(32)
<type 'int'>
>>> type(3.14)
<type 'float'>
```


Type Conversion Functions

Python has many *built-in* functions.

```
>>> int('32')   # Converts a string to an int
32
>>> int('not a number')
ValueError: invalid literal for int(): not a number
>>> int(3.14)   # Truncates a float to an int
3
```

Math Functions

```
>>> import math  # imports the math module
>>> print math     # prints the module object
<module 'math' from '/usr/library/math.so'>
>>> print math.sqrt(5)  # Objects use dot notation
2.2360679775
>>> print math.log(8)  # base e logarithm
2.07944154168
>>> print math.log(8,2)  # base 2 logarithm
2.0
```


More Conversion Functions

```
>>> float('32.6')    # String to float
32.6
>>> string(3.14)     # Float to string
'3.14'

# Python uses these functions – e.g.
N =  input('how old are you')
print N
```


Composition

Programming elements can be combined together.

```
>>> math.exp(2)     # returns e**2
7.3890560989306504
>>> math.log(2)     # returns base e log of 2
0.69314718055994529
>>> math.exp(math.log(2))    # returns 2
2.0
>>> math.log(math.exp(2))
2.0
```


Arguments are values

The argument to a function can be any valid expression.

```
>>> math.sqrt(Math.pi * Math.pi)
3.1415926535897931
>>> n = 400
>>> print math.sqrt(n)
20.0
>>> math.sqrt('Math.pi' * 4)
TypeError: a float is required
```

Programmer-Defined Functions

Documentation Online

- Python math functions:
  - http://docs.python.org/library/math.html
- Python built-in functions
  - http://docs.python.org/library/functions.html

Let's Write Some 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!
```


Why functions?

- Enable you to break up a program into its logical parts.
- Make programs 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, even by other programs.

Functions

If you know who <someone> is, say, Joe, you would just write:

```
print 'Happy birthday to you!'
print 'Happy birthday to you!'
print 'Happy birthday, dear Joe.'
print 'Happy birthday to you!'
```

But this code has a lot of repetition!

Functions

If you know who <someone> is, say, Joe, you would just write:

```
print 'Happy birthday to you!'
print 'Happy birthday to you!'
print 'Happy birthday, dear Joe.'
print 'Happy birthday to you!'
```


Functions

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

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

Functions

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

```
>>> happy()
```


Functions

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

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


Functions

The lyrics for Joe can be simplified as:

```
happy()
happy()
print 'Happy birthday, dear Joe.'
happy()
```


Functions

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

Functions

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Joe.
Happy birthday to you!
```


Functions

The lyrics for Joe can be simplified as:

```
happy()
happy()
print 'Happy birthday, dear Joe.'
happy()
```

which will print:

Functions

We can also make this a function as well:

```
def singJoe():
  happy()
  happy()
  print 'Happy birthday, dear Joe.'
  happy()
```


Functions

Invoking the function `singJoe` gives:

```
>> singJoe()
```

Functions

We can also make this a function for Ann:

```
def singAnn():
  happy()
  happy()
  print 'Happy birthday, dear Ann.'
  happy()
```


Functions

Invoking the function `singJoe` gives:

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


Functions

What does this do?

```
singJoe()
print
singAnn()
```


Functions

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Joe.
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Ann.
Happy birthday to you!
```

Functions

Can we make a generic function to sing happy birthday to anyone and use that for both Joe and Ann anybody else?

Yes, using a *parameter*—a variable whose value is specified by the caller (or user).

Functions

I still see some repetition here. Can we make a generic function to sing happy birthday to anyone and use that for Joe and Ann somehow?

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')
```

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 `'Joe'` gives:

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


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!
```


Functions

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

```
>> sing('Ann')
```

Functions

Now, what does this do?

```
sing('Joe')
print
sing('Ann')
```


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`.

Functions

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Joe.
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Ann.
Happy birthday to you!
```