Python's basic constructs

Values

Numbers, letters, words, … are collectively called *values*.

Values

Numbers, letters, words, … are collectively called *values*.

Example. The following are all values.

```
1823
3.141592653
'Trinity College'
```


Types

Values are categorized into different *types*: integers, floating-point (real) numbers, strings, …

Each value belongs to a single type (i.e., no value belongs to more than one type).

Variables

A *variable* is a name that refers to a value.

An *assignment* statement creates a variable and assigns a value to it.

Example.

```
>>> n = 1823
```


Variables

A *variable* is a name that refers to a value.

An *assignment* statement creates a variable and assigns a value to it.

Example.

```
>>> n = 1823
>>> print n
```


Variables

A *variable* is a name that refers to a value.

An *assignment* statement creates a variable and assigns a value to it.

Variables

A *variable* is a name that refers to a value.

An *assignment* statement creates a variable and assigns a value to it.

Example.

```
>>> n = 1823
>>> print n
```


Variables

A *variable* is a name that refers to a value.

An *assignment* statement creates a variable and assigns a value to it.

Example.

```
>>> n = 1823
>>> print n
1823
```

Variables and types

The *type* of a variable is the type of its value.

Example.

```
>>> n = 1823
```


Variables and types

The *type* of a variable is the type of its value.

Example.

```
>>> n = 1823
>>> type(n)
<type 'int'>
```


Variables and types

The *type* of a variable is the type of its value.

Variables and types

The *type* of a variable is the type of its value.

Example.

```
>>> n = 1823
>>> type(n)
```

Variable names

Example.

```
temp = 33
exam3 = 89
my_college = 'Trinity College'
```


Variable names

A variable can be named by any word (except for certain reserved words).

A variable name can contain any letter or digit, but it must begin with a letter.

To name a variable using more than one word, it is customary to use _ to connect words.

Statements

A *statement* is the smallest unit of code that can be executed as a stand-alone instruction.

Example.

```
n = 1823
```


Statements

A *statement* is the smallest unit of code that can be executed as a stand-alone instruction.

Expressions

An *expression* is a code fragment that has a value.

Example.

`1823 + n` (assuming `n` has a value)

Expressions

An *expression* is a code fragment that has a value.

Statement and expressions

An *expression* is usually part of a statement.

Expressions

An *expression* is a code fragment that has a value.

Non example.

```
n = 1823
```

Functions

Statement and expressions

An *expression* is usually part of a statement.

Example. This statement

```
n = 11 * (2 + 3)
```

contains an expression `2 + 3`.

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

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

But this code has a lot of repetition!

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:

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

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


Functions

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

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

Functions

The lyrics for Joe can be simplified as:

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

which will print:

Functions

The lyrics for Joe can be simplified as:

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


Functions

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


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


Functions

Invoking the function `singJoe` gives:

```
>> singJoe()
```


Functions

What does this do?

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


Functions

We can also make this a function for Ann:

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

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

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

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?

Yes, using 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!
```


Functions

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

```
>> sing('Joe')
```


Functions

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

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


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`.

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

Now, what does this do?

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


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!
```