

Values and Variables

Languages and Computation

Every powerful language has three mechanisms for combining simple ideas to form more complex ideas:(SICP 1.1)

- ▶ primitive expressions, which represent the simplest entities the language is concerned with,
- ▶ means of combination, by which compound elements are built from simpler ones, and
- ▶ means of abstraction, by which compound elements can be named and manipulated as units.

Today we'll begin learning Python's facilities for primitive expressions, combination, and elementary abstraction.

Values

An expression has a value, which is found by evaluating the expression. When you type expressions into the Python REPL, Python evaluates them and prints their values.

```
>>> 1
1
>>> 3.14
3.14
>>> "pie"
'pie'
```

The expressions above are literal values. A literal is a textual representation of a value in Python source code.

Types

All values have types. Python can tell you the type of a value with the built-in type function:

```
>>> type(1)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type("pie")
<class 'str'>
```

The Meaning of Types

Types determine which operations are available on values. For example, exponentiation is defined for numbers (like int or float):

```
>>> 2**3  
8
```

... but not for str (string) values:

```
>>> "pie"**3  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Python is Dynamically Typed

Python is dynamically typed, meaning that types are not resolved until run-time. This means two things practically:

1. Values have types, variables don't:

```
>> a = 1
>>> type(a)
<class 'int'>
>>> a = 1.1 # This would not be allowed in a statically typed
           language
>>> type(a)
<class 'float'>
```

2. Python doesn't report type errors until run-time. We'll see many examples of this fact.

Overloaded Operators

Some operators are overloaded, meaning they have different meanings when applied to different types. For example, `+` means addition for numbers and concatenation for strings:

```
>>> 2 + 2
4
>>> "Yo" + "lo!"
'Yolo!'
```

`*` means multiplication for numbers and repetition for strings:

```
>>> 2 * 3
6
>>> "Yo" * 3
'YoYoYo'
>>> 3 * "Yo"
'YoYoYo'
```

Expression Evaluation

Mathematical expressions are evaluated using precedence and associativity rules as you would expect from math:

```
>>> 2 + 4 * 10  
42
```

If you want a different order of operations, use parentheses:

```
>>> (2 + 4) * 10  
60
```

Note that precedence and associativity rules apply to overloaded versions of operators as well:

```
>>> "Honey" + "Boo" * 2  
'HoneyBooBoo'  
>>> ("Honey" + "Boo") * 2  
'HoneyBooHoneyBoo'
```


Variables

A variable is a name for a value. You bind a value to a variable using an assignment statement (or as we'll learn later, passing an argument to a function):

```
>>> a = "Ok"
>>> a
'Ok'
```

= is the assignment operator and an assignment statement has the form
<variable_name> = <expression>

Variable names, or identifiers, may contain letters, numbers, or underscores and may not begin with a number.

```
>>> 16_candles = "Molly Ringwald"
      File "<stdin>", line 1
        16_candles = "Molly Ringwald"
            ^
SyntaxError: invalid syntax
```

Keywords

Python reserves some identifiers for its own use.

```
>>> class = "CS 2316"
      File "<stdin>", line 1
        class = "CS 2316"
            ^
SyntaxError: invalid syntax
```

The assignment statement failed because `class` is one of Python's keywords:

<code>False</code>	<code>class</code>	<code>finally</code>	<code>is</code>	<code>return</code>
<code>None</code>	<code>continue</code>	<code>for</code>	<code>lambda</code>	<code>try</code>
<code>True</code>	<code>def</code>	<code>from</code>	<code>nonlocal</code>	<code>while</code>
<code>and</code>	<code>del</code>	<code>global</code>	<code>not</code>	<code>with</code>
<code>as</code>	<code>elif</code>	<code>if</code>	<code>or</code>	<code>yield</code>
<code>assert</code>	<code>else</code>	<code>import</code>	<code>pass</code>	
<code>break</code>	<code>except</code>	<code>in</code>	<code>raise</code>	

Assignment Semantics

Python evaluates the expression on the right-hand side, then binds the expression's value to the variable on the left-hand side. Variables can be reassigned:

```
>>> a = 'Littering and ... '  
>>> a  
'Littering and ... '  
>>> a = a * 2  
>>> a  
'Littering and ... Littering and ... '  
>>> a = a * 2  
>>> a          # I'm freakin' out, man!  
'Littering and ... Littering and ... Littering and ... Littering and  
... '
```

Note that the value of `a` used in the expression on the right hand side is the value it had before the assignment statement.

What's the type of `a`?

Type Conversions

Python can create new values out of values with different types by applying conversions named after the target type.

```
>>> int(2.9)
2
>>> float(True)
1.0
>>> int(False)
0
>>> str(True)
'True'
>>> int("False")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'False'
```

Strings

Three ways to define string literals:

- ▶ with single quotes: 'Ni!'
- ▶ double quotes: "Ni!"
- ▶ Or with triples of either single or double quotes, which creates a multi-line string:

```
>>> """I do HTML for them all,  
... even made a home page for my dog."""  
'I do HTML for them all,\neven made a home page for my dog.'
```

Strings

Note that the REPL echoes the value with a `\n` to represent the newline character. Use the `print` function to get your intended output:

```
>>> nerdy = """I do HTML for them all,  
... even made a home page for my dog."""  
>>> nerdy  
'I do HTML for them all,\nneven made a home page for my dog.'  
>>> print(nerdy)  
I do HTML for them all,  
even made a home page for my dog.
```

Strings

Choice of quote character is usually a matter of taste, but the choice can sometimes buy convenience. If your string contains a quote character you can either escape it:

```
>>> journey = 'Don\'t stop believing.'
```

or use the other quote character:

```
>>> journey = "Don't stop believing."
```

String Operations

Because strings are sequences we can get a string's length with `len()`:

```
>>> i = "team"  
>>> len(i)  
4
```

and access characters in the string by index (offset from beginning – first index is 0) using `[]`:

```
>>> i[1]  
'e'
```

Note that the result of an index access is a string:

```
>>> type(i[1])  
<class 'str'>  
>>> i[3] + i[1]  
'em'  
>>> i[-1] + i[1] # Note that a negative index goes from the end  
'me'
```


String Slicing

`[:end]` gets the first characters up to but not including end

```
>>> al_gore = "manbearpig"  
>>> al_gore[:3]  
'man'
```

`[begin:end]` gets the characters from begin up to but not including end

```
>>> al_gore[3:7]  
'bear'
```

`[begin:]` gets the characters from begin to the end of the string

```
>>> al_gore[7:]  
'pig'  
>>>
```

String Methods

`str` is a class (you'll learn about classes later) with many methods (a method is a function that is part of an object). Invoke a method on a string using the dot operator.

`str.find(substr)` returns the index of the first occurrence of `substr` in `str`

```
>>> 'foobar'.find('o')  
1
```

