

Values and Variables

Christopher Simpkins

`chris.simpkins@gatech.edu`

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'>
```

Types determine which operations are available on values. For example, exponentiation is defined for numbers (like `int` or `float`) but not for `str` (string) values.

```
>>> 2**3
8
>>> "pie"**3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

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:

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

We say "a gets the value '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](#):

False	<code>class</code>	<code>finally</code>	<code>is</code>	<code>return</code>
None	<code>continue</code>	<code>for</code>	<code>lambda</code>	<code>try</code>
True	<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 = "Ok"
>>> a
'Ok'
>>> a = a * 3
>>> a
'OkOkOk'
>>> a = a * 3
>>> a
'OkOkOkOkOkOkOkOkOkOk'
```

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'
```

- floats are truncated when converted to ints
- The bool literals True and False can be converted to numbers
- Type conversions won't parse a str to interpret it as a number or boolean

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.
```

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

Strings have many operations. Because they are sequences we can get a string's length:

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

access characters by index (offset from beginning) using []:

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

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

```
>>> type(i[1])  
<class 'str'  
>>> i[3] + i[1]  
'me'  
>>> i[-1] + i[1]  
'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 Functions

`str` is a class (you'll learn about classes soon) with many functions. Access a function on a string using the dot operator.

```
>>> south_park = "stan kyle cartman kenny"
>>> kids = south_park.split()
>>> kids
['stan', 'kyle', 'cartman', 'kenny']
>>> ", ".join(kids)
'stan,kyle,cartman,kenny'
>>> kids
['stan', 'kyle', 'cartman', 'kenny']
>>> [s.capitalize() for s in kids]
['Stan', 'Kyle', 'Cartman', 'Kenny']
```

There are many more functions on strings. Review the book and play around to become comfortable with them.

