Values and Variables

Georgia Tech

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.

Georgia Tech

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'>
```

Georgia Tech

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 resoved 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'
```

Georgia Tech

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'
```

Georgia Tech

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
```

Georgia Tech

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 becuase class is one of Python's keywords:

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

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'
```

Exercise

Create variables named

- `exam1`, `exam2`, and `exam3` and assign them literal values,
- `exam_avg` which is assigned the value of an expression computing the average of the exams above, and
- `grade_report`, which is a string of the form "Your exam average is X", where X is the value of `exam_avg`

Georgia Tech

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,\neven 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]
'me'
>>> i[-1] + i[1] # Note that a negative index goes from the end
'me'
```

Georgia Tech

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'
>>>
```

Georgia Tech

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 occurence of `substr` in `str`

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

Exercise: using the `find` method and string slicing, write an expression that returns the user name from an email address, e.g., "bob@aol.com" => "bob". Do the same for the host name, e.g., "aol.com".

Values, Variables, and Expression

- Values are the atoms of computer programs
- We (optionally) combine values using operators and functions to form compound expressions
- We create variables, which are identifiers that name values, define other identifiers that name functions, classes, modules and packages
- By choosing our identifiers, or names, carefully we can create beautiful, readable programs

Georgia Tech