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 expresions, 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.

- Do strings always get printed with single quotes even if we diefine them with double quotes?

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'>
```

- What's the type of '1'?

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'
```

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'
```

- How could we slighlty modify the expression above to evaluate to '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
```

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:

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

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

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
```

- What hapens if you try to use a variable name on the list of keywords?
- What happens if you use `print` as a variable name?

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'
```

- What happens if you evaluate the expression `integer('1')` ?

Boolean Values

There are 10 kinds of people:

- those who know binary, and
- those who don't.

Python Booleans

In Python, boolean values have the `bool` type. Four kinds of boolean expressions:

- `bool` literals: `True` and `False`
- `bool` variables
- expressions formed by combining non-`bool` expressions with comparison operators
- expressions formed by combining `bool` expressions with logical operators

Comparison Expressions

Simple boolean expressions formed with comparison operators:

- Equal to: `==`, like $=$ in math
  - Remember, `=` is assignment operator, `==` is comparison operator!
- Not equal to: `!=`, like $\neq$ in math
- Greater than: `>`, like $>$ in math
- Greater than or equal to: `>=`, like $\geq$ in math ...

Examples:

```
1 == 1 # True
1 != 1 # False
1 >= 1 # True
1 > 1  # False
```

Truth in Python

These values are equivalent to `False`:

- boolean `False`
- `None`
- integer `0`
- float `0.0`
- empty string `""`
- empty list `[]`
- empty tuple `()`
- empty dict `{}`
- empty set `set()`

All other values are equivalent to True.

Logical Expressions

Boolean expressions can be combined using logical operators and, or, not.

```
1 (1 == 1) and (1 != 1) // False
2 (1 == 1) or (1 != 1) // True
```

Logical expressions use short-circuit evaluation:

- `or` only evaluates second operand if first operand is `False`
- `and` only evaluates second operand if first operand is `True`

What are the values of the following expressions?

- `True and False`
- `True and 0`
- `True and []`
- `True and None`
- `type(True and None)`
- `False or 1`
- `True or 1`

Guard idiom: `(b == 0)or print(a / b)`, or `(b != 0)and print(a / b)`

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

Your turn:

- Exercise 1