

# 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
2 1
3 >>> 3.14
4 3.14
5 >>> "pie"
6 '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 define them with double quotes?

# Types

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

```
1 >>> type(1)
2 <class 'int'>
3 >>> type(3.14)
4 <class 'float'>
5 >>> type("pie")
6 <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`):

```
1 >>> 2##3
2 8
```

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

```
1 >>> "pie"##3
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   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:

```
1 >>> 2 + 2
2 4
3 >>> "Yo" + "lo!"
4 'Yolo!'
```

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

```
1 >>> 2 * 3
2 6
3 >>> "Yo" * 3
4 'YoYoYo'
5 >>> 3 * "Yo"
6 'YoYoYo'
```

# Expression Evaluation

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

```
1 >>> 2 + 4 * 10
2 42
```

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

```
1 >>> (2 + 4) * 10
2 60
```

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

```
1 >>> "Honey" + "Boo" * 2
2 'HoneyBooBoo'
```

- ▶ How could we slightly 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):

```
1 >>> a = "Ok"
2 >>> a
3 '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.

```
1 >>> 16_candles = "Molly Ringwald"
2     File "<stdin>", line 1
3         16_candles = "Molly Ringwald"
4             ^
5 SyntaxError: invalid syntax
```



# 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:

```
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.

```
1 >>> class = "CS 2316"
2     File "<stdin>", line 1
3         class = "CS 2316"
4         ^
5 SyntaxError: invalid syntax
```

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

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

- ▶ What happens 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:

```
1 >>> a = 'Littering and ... '  
2 >>> a  
3 'Littering and ... '  
4 >>> a = a * 2  
5 >>> a  
6 'Littering and ... Littering and ... '  
7 >>> a = a * 2  
8 >>> a                # I'm freakin' out, man!  
9 '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.

```
1 >>> int(2.9)
2 2
3 >>> float(True)
4 1.0
5 >>> int(False)
6 0
7 >>> str(True)
8 'True'
9 >>> int("False")
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12 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 == 1 # True
2 1 != 1 # False
3 1 >= 1 # True
4 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