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.

By the end of this lesson you will

- know what a value is and how to create one,
- know what a variable is and how to use them as simple means of abstraction.
- know what a type is and how it constrains what you can do with expressions, and
- know what an expression is how to combine them produce new values,

Values

Figure 1: Values

Values and Expressions

value a well-defined chunk of data in memory

expression a sequence of symbols that can be *evaluated* to produce a value

When you an expression into the Python REPL, Python evaluates it and prints its value.

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

The simplest expressions are *literal* values, as in the examples above.

literal the textual representation of a value in source code.

Compound expressions combine values using operators. Here the `+` operator combines the two literal values `2` and `3` – the *operands* – to produce the value `5`:

```
>>> 2 + 3
5
```

Have a Python REPL session open for this lesson so you can follow along and try your own ideas.

Types

You can think of a type

- structurally: as an interpretation of the bits comprising a chunk of data,
- denotationaly: as a set of values, or
- abstraction-based: as the set of operations available for a type.

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'>
```

Active Review

- What's the type of `'1'`?

Variables

Think of a variable as a name for a value. You bind a value to a variable using an assignment statement (or by passing an argument to a function), after which the variable *denotes* the value:

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

= is the assignment operator. An assignment statement has the form:

<variable_name> = *<expression>*

You can unbind a variable with the `del` function.

```
>>> del(a)
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Variable Names

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

Active Review

- What happens when you execute this assignment statement?

```
>>> 16_candles = "Molly Ringwald"
```

Keywords

Python reserves keywords for its own use.

```
>>> from keyword import kwlist
>>> import math
>>> numrows = 5
>>> numcols = math.ceil(len(kwlist) / numrows)
>>> for row in range(numrows):
...     for col in range(0, numrows * numcols, numrows):
...         kw = kwlist[row+col] if row+col < len(kwlist) else ''
...         print(f'{kw:<12}', end='')
...     print()
...
False        assert       continue     except       if           nonlocal     return
None         async        def          finally      import       not          try
True         await        del          for          in           or           while
and          break        elif         from         is           pass         with
as           class        else         global       lambda       raise        yield
```

Active Review

- What happens when you execute this assignment statement?

```
>>> class = "Professional Python"
```

- What happens if you use `print` as a variable name?
- How can you fix it?

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 # would be disallowed 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.

Aside: The Sizes of Types

One of the convenient things about Python is that you don't have to worry about overflow or underflow[1]. For example, as in mathematics, the set `int` is unbounded:

```
>>> import sys
>>> x = sys.maxsize
>>> x
9223372036854775807 # That's ~ 9.2 quintillion, i.e., 9.2e+18
>>> x = x + 1
>>> x
9223372036854775808
>>>
```

But you should consider `sys.maxsize`, the word size of your processor (64 bits in this example, since `sys.maxsize` $= 2^{63} - 1$), to be the practical limit, because it's the theoretical limit [2] of addressable RAM and thus the largest possible (but certainly impractical) array you could store in main memory and therefore, as you'll learn later, the largest possible list index.

In many other programming languages, size limits can crop up in sometimes amusing ways, Gangnam Style!

[1] In regular Python you don't have to worry about type size limits, but in scientific Python, which relies on libraries written in C, C++ and Fortran you do.

[2] Not strictly true, but practically true.

Types as Sets of Operations

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'
```

This is the primary way to think about types in Python.

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 sequences, like `str`s:

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

Values, Variables, and Expressions

- Values are the atoms of computer programs
- Variables are identifiers that denote values
  - Identifiers also denote functions, classes, modules and packages
- Choose identifiers carefully to create beautiful, readable programs