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

Values

Figure 1: Values

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, - denotationally: 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 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, 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?

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 = "CS 2316"
```

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

Statements vs Expressions

Expressions have values, statements only have effects – like binding a variable to a value or effecting control flow in a program. Assignment using `=` is a statement – it cannot be used where a value is expected.

```
>>> while guess = input("Guess a number: "):
...     if guess == "7":
...         break
  Input In [42]
    while guess = input("Guess a number: "):
          ^
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

Since version 3.8, Python provides the "walrus" operator, `:=`, which creates an assignment expression, where the assigned value is the value of the expression:

```
>>> while guess := input("Guess a number: "):
...     if guess == "7":
...         break
...
Guess a number: 1
Guess a number: 7
>>>
```

Note that `while` and `if` are statements – they don't produce values, they create effects.

We will see a few cases in future lessons where the walrus operator is helpful.

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 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'
```

Active Review

- How could we modify the expression above to evaluate to 'HoneyBooHoneyBoo' ?

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 # 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.

Active Review

Evaluate the following expressions in the Python REPL. Be sure to type them exactly as written.

- `2 + 3`
- `"2" + "3"`
- `"2" + 3`
- `2 + "3"`

Type Conversions

Convert a value to a different type 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'
```

Active Review

Modify the following expressions to produce the indicated results.

- `"2" + 3` (we want `"23"`)
- `2 + "3"` (we want `5`)

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 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

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

Active Review

- What is the value of `"foo" == "Foo"`?
- What is the value of `"foo" > "Foo"`?

Logical Operators

The values produced by logial operators are often shown in truth tables:

a	b	not a	a and b	a or b
False	False	True	False	False
False	True	True	False	True
True	False	False	False	True
True	True	False	True	True

Some examples:

```
1 True and True  # True
2 True and False # False
3 True or False  # True
4 False or False # False
5 not True       # False
```

Truth in Python

The zero values of built-in types 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.

- Every value in Python is either *truthy* or *falsey* and can be used in a boolean context.

Short-circuit Evaluation

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`

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

Active Review

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`
- `1 and "done"`
- `1 == 1 or 0`
- `1 == 1 and 0`
- `1 == (1 and 0)`

Sequences

Sequences are ordered collections of objects – lists, tuples, and strings.

```
>>> boys = ['Stan', 'Kyle', 'Cartman', 'Kenny']
>>> boys[0]
'Stan'
>>> empty = []
>>> also_empty = list()
```

Lists are mutable.

```
>>> boys[2] = 'Eric'
>>> boys
['Stan', 'Kyle', 'Eric', 'Kenny']
```

Tuples and strings are immutable.

```
>>> pair = 1, 2
>>> pair
(1, 2)
>>> pair[0] = 0
Traceback (most recent call last):
  Input In [37] in <cell line: 1>
    pair[0] = 0
TypeError: 'tuple' object does not support item assignment
```

Dictionaries and Sets

A dictionary is a map from keys to values.

```
>>> capitals = {}
>>> capitals['Georgia'] = 'Atlanta'
>>> capitals['Alabama'] = 'Montgomery'
>>> capitals
{'Georgia': 'Altanta', 'Alabama': 'Montgomery'}
>>> capitals['Georgia']
'Atlanta'
```

Sets have no duplicates, like the keys of a `dict`. They can be iterated over (we'll learn that later) but can't be accessed by index.

```
>>> names = set()
>>> names.add('Ally')
>>> names.add('Sally')
>>> names.add('Mally')
>>> names.add('Ally')
>>> names
{'Ally', 'Mally', 'Sally'}
>>> set([1,2,3,4,3,2,1])  # Removes duplicates
{1, 2, 3, 4}
```

Values, Variables, and Expressions

- Values are the atoms of computer programs
- Expressions produce values
- We combine values using operators and functions to form compound expressions
- Variables are identifiers that denote values
  - Identifiers also denote functions, classes, modules and packages
- Choose identifiers carefully to create beautiful, readable programs