SI 506 Lecture 03

Topics

1. Comments (single line, multiline)
2. Values (objects) and types
3. Variables (labels) and variable assignment
4. Built-in functions print(), type(), len()
5. Basic arithmetic operations (add, subtract, multiply, divide)

Vocabulary

- **Boolean**. A type (`bool`) or an expression that evaluates to either `True` or `False`.
- **Built-in Function**. A function defined by the Standard Library that is always available for use.
- **Dictionary**. An associative array or a map, wherein each specified value is associated with or mapped to a defined key that is used to access the value.
- **Expression**. An accumulation of values, operators, and/or function calls that return a value. `len(< some_list >)` is considered an expression.
- **Immutable**. Object state cannot be modified following creation. Strings and tuples are immutable.
- **Mutable**. Object state can be modified following creation. Lists are mutable.
- **Operator**. A symbol for performing operations on values and variables. The assignment operator (`=`) and arithmetic operators (`+`, `-`, `*`, `/`, `**`, `%`, `//`).
- **Sequence**. An ordered set such as `str`, `list`, or `tuple`, the members of which (e.g., characters, elements, items) can be accessed.
- **Statement**. An instruction that the Python Interpreter can execute. For example, assigning a variable to a value such as `name = 'arwhyte'` is considered a statement.
- **Tuple**. An ordered sequence that cannot be modified once it is created.

1.0 Comments

```
# A single line comment <-- commences with hash (#) character

"""
This is a block comment comprising a multi-line string. This is actually a
string
constant that is denoted by the use of triple quotation marks.
"""
```

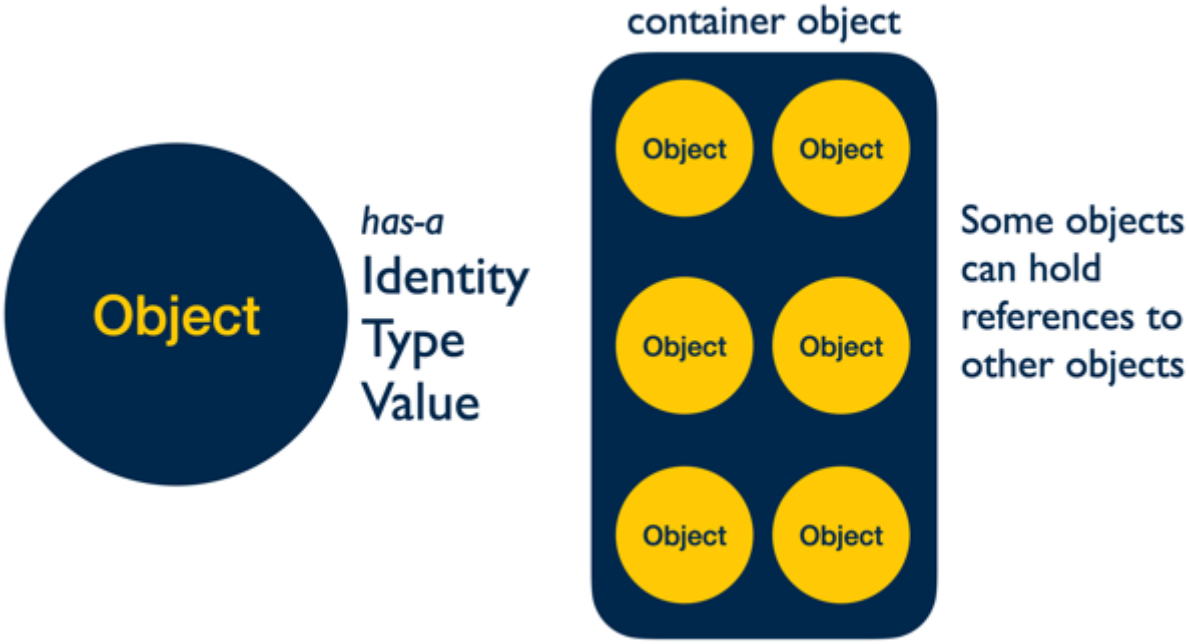
2.0 Values (Objects) and Types

> "Everything is an Object"

Jake VanderPlas, *A Whirlwind Tour of Python* (O'Reilly Media, Inc., 2016)

> "_Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects."

Python Software Foundation, "The Python Language Reference"



Python is an object-oriented programming language. This means that the Python data model represents strings, integers, floating point numbers, containers (e.g., `list`, `tuple`, `set`), mappings (`dict`), functions, class instances, modules and other types as **objects**.

The basic characteristics of a Python object can be summarized as follows:

1. Every object possesses an *identity* (memory address), *type*, and *value*.
2. An object's *type* determines its behavior as well as defines the possible values it may contain.
3. The value of some objects can be modified. An object's *mutability* (e.g., *mutable* = capable of modification; *immutable* = incapable of modification) is determined by its *type*.
4. Objects are never explicitly destroyed; memory management and "garbage-collection" is typically ceded to the Python interpreter without the need for manual intervention.

2.1 Numbers: integer, float (decimal)

```
506 # integer

.25 # float
```

2.2 Sequences (Ordered Set)

- string (immutable)
- list (mutable)
- tuple (immutable)

```
'Welcome to SI 506' # string

['arwhyte', 'amjaha', 'dsewhite', 'raynez', 'torchont', 'ykamat'] # list
```

```
with five elements

(504, 506, 507, 618) # tuple with three items
```

2.3 Dictionary (associative array)

A dictionary is composed of key-value pairs. Insertion order is maintained.

```
{
    'course': 'SI 506',
    'instructor_count': 1,
    'gsi_count': 5,
    'ia_count': 2
} # four key-value pairs
```

2.4 Boolean

```
True
False
```

2.5 None

`None` is an object of type `<class 'NoneType'>` and represents `null` or the absence of a concrete value.

⚠️ Note that `None` does not equal 0.

```
None
```

3.0 Variable (name, label, pointer)

A Python *variable* is a name or label that refers to an object in memory. Jake VanderPlus describes the concept in *A Whirlwind Tour of Python*:

> "... variables are simply pointers [to objects], and the variable names themselves have no attached type information."

Or as Naomi Cedar writes in *The Quick Python Book*, Third Edition (Manning Publications, 2018):

> "The name variable is somewhat misleading . . .; name or label would be more accurate."

You use the assignment (=) operator to *assign* a value to a variable or *bind* the name (i.e., pointer, label) to the object (e.g., `variable_name = < object >`).

```

```

```
num = 506

welcome_message = 'Welcome to SI 506'

teaching_team = ['arwhyte', 'amjaha', 'dsewhite', 'raynez', 'torchont',
'ykamat']

chorus = """
Hail! to the victors valiant
Hail! to the conquering heroes
Hail! Hail! to Michigan
the leaders and best!
"""
```

4.0 Variable Naming Rules and Conventions

Default convention: lowercase word(s) or recognizable abbreviation (e.g., num, val, var); separate words with an underscore.

❗ Readability and comprehensibility matters.

4.1 Good

```
# Choose lowercase
uniqname = 'arwhyte'

# Separate words with underscore (_)
course_code = 'SI 506'

# Use plural form to indicate a set or sequence
course_codes = ['SI 506', 'SI 507', 'SI 618']

# Ok to use recognizable abbreviations like num[ber], val[ue] or
var[iable].
num = 27

# "is_", "has_" Boolean true/false
is_enrolled = True
has_mask = True

# All caps designates a module level constant (special case)
BASE_URL = 'https://si506.org/'

# Function definition specifying two parameters x and y (a foreshadowing
of the weeks ahead)
def multiply(x, y):
    return x * y # arithmetic

# Call the function and pass two numeric arguments
product = multiply(14, 27)
```

```
print(f"product = {product}") # formatted string literal (f-string)

# Built-in enumerate() function adds a counter < i > when looping over <
course_codes >
for i, code in enumerate(course_codes, 1):
    print(f"{i}. {code}")
```

4.2 Bad (But Legal)

```
# Opaque
c = 'SI 506'
cc = 'SI 506'

# Reserve CamelCase for class names.
CourseCode = 'SI 506'

# Unnecessarily verbose; difficult to read.
c_o_u_r_s_e_c_o_d_e = 'SI 506'

# Difficult to read; guaranteed to annoy.
cOUrsE_cOdE = 'SI 506'
```

❗ Avoid prefixing or suffixing variable names with single (_) or double underscores (__) — known in the Python community as a "dunder" — until you gain experience as a Python programmer.

Variable names prefixed with a single underscore like `_course_code` are, by convention, considered private member variables in a class. Variable names prefixed with a double underscore like `__course_code__`, gets renamed at runtime by the Python interpreter in a process known as "name mangling".

💡 These and other naming conventions that employ leading and/or trailing underscores are *out of scope* for SI 506. That said, if you want to learn more on the subject see D. Bader, "The Meaning of Underscores in Python" (dbader.org, nd).

4.3 Ugly (Illegal)

The Python Interpreter will raise a `SyntaxError` at runtime whenever it encounters the following illegal names:

❗ Python keywords are reserved and cannot be used as variable names.

```
# Illegal: keyword used as a variable name (language-specific identifiers
reserved by Python)

class = 'SI 506'

# Illegal: variable name commences with a numeric value.
```

```
506_umsi = 'SI 506'

# Illegal: variable name commences with a special character (e.g., `@`,
`%`, `$`, `&`, `!`)

$number = 506

# Illegal: variable name includes a dash (`-`).

course-list = ['SI 506', 'SI 507', 'SI 618']

# Illegal: variable name includes whitespace.

course name = 'SI 506' # illegal; uncomment to test
```

❗ Also avoid use of built-in function names as variable names. Name clashes may occur in your code. If you do opt to use or "shadow" such names add a trailing underscore character to the name (`_`) per the PEP 08 recommendation or opt for a different name (`len_` or `length` for `len`).

```
# Shadowing; risk name clash with built-in functions
id = 506
str = 'Go Blue'
min = 0
max = 27
len = 6

# Alternative names
id_ = 506

str_ = 'Go Blue'
val = 'Go Blue'

min_ = 0
min_val = 0

max_ = 27
max_val = 27

len_ = 6
length = 6
```

5.0 Built-in Functions (`print()`, `type()`, `len()`)

The Python Interpreter includes a number of built-in functions that are always available for you to call.

💡 A function is a defined block of code that performs (ideally) a single task. Functions only run when they are explicitly called. A function can be defined with one or more *parameters* that allow it to accept *arguments* from the caller in order to perform a computation. A function can also be designed to return a

computed value. Functions are considered "first-class" objects in the Python eco-system. You will soon write your own functions; for now we introduce a select number of built-in functions for you to use.

5.1 `print()`: print passed in object to the screen

```
# Passing a hard-coded string.
print('SI 506 rocks!')

# Passing a variable name which points to a string.
print(welcome_message)

# Passing a variable name which points to a multiline string.
print(chorus)
```

5.2 `type()`: determine object's data type

```
data_type = type(num)
print(data_type) # returns <class 'int'>

data_type = type(welcome_message)
print(data_type) # returns <class 'str'>

data_type = type(teaching_team)
print(data_type) # returns <class 'list'>
```

5.3 `len()`: check length of a sequence (i.e., number of elements)

```
# Count characters in string (including whitespace).
chars_count = len(welcome_message)
print(chars_count)

# Count number of elements in list.
team_count = len(teaching_team)
print(team_count)
```

6.0 Basic Arithmetic (addition, subtraction, multiplication, division)

Python supports math operations. The order of operations is expressed conveniently by the acronym **PEMDAS**: Parentheses, Exponentation, Multiplication | Division (same precedence), Addition | Subtraction.

1. Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * (3-1)` is 4, and `(1+1)**(5-2)` is 8. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even though it doesn't change the result.

2. Exponentiation has the next highest precedence, so `2 ** 1 + 1` is 3 and not 4, and `3 * 1 ** 3` is 3 and not 27.

3. Multiplication and both division operators have the same precedence, which is higher than addition and subtraction, which also have the same precedence. So `2*3-1` yields 5 rather than 4, and `5-2*2` is 1, not 6.

4. Operators with the same precedence (except for **) are evaluated from left-to-right. In algebra we say they are left-associative. So in the expression `6-3+2`, the subtraction happens first, yielding 3. We then add 2 to get the result 5. If the operations had been evaluated from right to left, the result would have been `6-(3+2)`, which is 1.

6.1 Variable assignment

```
# Counts
lecturer_count = 1
gsi_count = 5
ia_count = 2
lab_section_count = 8
student_count = 250
```

6.2 Addition (+ operator)

```
team_count = lecturer_count + gsi_count + ia_count
print(f"team_count = {team_count}")
```

6.3 Subtraction (- operator)

```
instructor_count = team_count - ia_count
print(f"instructor_count = {instructor_count}")
```

6.4 Multiplication (* operator)

```
max_enrollment = lab_section_count * 50
print(f"max_enrollment = str({max_enrollment})")
```

6.5 Floating point division (/ operator)

Return a decimal value (a float).

```
avg_lab_size = student_count / lab_section_count
print(f"average lab size = {avg_lab_size}")
```

6.6 Floor division a.k.a integer division (`//` operator)

Return an integer value (ignore fractional values).

```
avg_lab_size = student_count // lab_section_count
print(f"average lab size = {avg_lab_size}")
```