Lecture 01: Fundamental Data Types

MPCS 51042-2 : Python Programming

Ron Rahaman

The University of Chicago, Dept of Computer Science

Oct 7, 2018

Table of Contents

Table of Contents

Course Objectives

- Object-oriented and functional programming in Python
- Vital standard library and third-party library tools
- Writing high-quality, reusable code

Prerequisites

- Procedural programming in any language
- Basic object-oriented concepts
- Your operating system's command line

Syllabus

Week	Date	Topics	Assignment Due
2	Oct 7	Basic Data Types	
3	Oct 14	More Data Types, Functions, Git	Homework 1
4	Oct 21	Advanced functions	Homework 2
5	Oct 28	In-class midterm (1/2 class), Iterators	
6	Nov 4	Object-Oriented Programming 1	Homework 3
7	Nov 11	Object-Oriented Programming 2	Homework 4
8	Nov 18	Exception Handling and Unit Testing	Homework 5
9	Nov 25	NumPy	Homework 6
10	Dec 2	Pandas	Homework 7
11	Dec 9	In-class Final	

Reading

Week	Topics	Reading
2	Basic Data Types	Lutz 2, 4, 5, 7, 8, 10-13
3	More Data Types, Functions, Git	Lutz, 8, 9, 16, 18; Chacon 2, 3
4	Advanced functions	Lutz 17, 19, 20
5	Iterators	Lutz 14
6	Object-Oriented Programming 1	Lutz 26, 28; Ramalho 1, 11, 12
7	Object-Oriented Programming 2	Ramahlo TBA
8	Exception Handling and Unit Testing	Lutz 33 - 36
9	NumPy	VanderPlas 2
10	Pandas	VanderPlas 3

Grading

Breakdown of overall grade:

- 7 weekly programming assignments (50% of grade)
- One in-class paper midterm (20% of grade)
- One in-class paper final (30% of grade)

No late assignments will be accepted.
Lowest homework grade will be dropped.

Staff and Office Hours

Name	Role	Hours	Location
Ron Rahaman	Lecturer	Thur 4:30pm - 6:30pm TBA 8:30pm - 9:30pm	Hyde Park Google Hangouts
Shashank Sharma	TA		Gleacher Center
Jack Gang	Grader		
Zhenhui Huang	Grader		
Anuved Verma	Grader		

Course Websites

- Piazza (`https://piazza.com/`): Discussions, lectures, homework 1
- GitLab (`https://mit.cs.uchicago.edu/`): Homeworks 2+
- Canvas (`https://canvas.uchicago.edu/`): Gradebook

Textbooks and Sources

- Mark Lutz. *Learning Python, 5th Ed.* O'Reilly Media.
- Luciano Ramalho. *Fluent Python*. O'Reilly Media.
- Jake VanderPlas. *Python Data Science Handbook.* Freely available at `https://jakevdp.github.io/PythonDataScienceHandbook/`
- Scott Chacon, Ben Straub. *Pro Git, 2nd Ed.* Freely available at `https://git-scm.com/book`
- The Python Docs: `https://docs.python.org/3.7/index.html`

Academic Honesty

- Cite any piece of code that is based on an outside source.
  - Student interactions (cite student's name)
  - Internet (cite URL)
  - Exceptions: Textbook and lecture code can be used without citing
- Never post course material in publicly accessible spaces. This material includes (but is not limited to):
  - Lecture notes, slides, and demos
  - Assignment statements
  - Student solutions
  - Instructor solutions
  - Midterm and final questions and solutions
- Piazza is not publicly accessible, so course material may be freely shared there.

Table of Contents

Advantages of Python

- Multi-paradigm
  - Procedural, object-oriented or functional.
  - Can choose whichever paradigm is most appropriate for your task.
- Extremely portable
- Powerful built-in data structures
- Language syntax doesn't obscure your algorithm.
- Execution speed is excellent for most tasks.
  - In CPython (most common implementation), built-in objects are written in optimized C.
  - Many third-party Python libraries also use optimized C for specialized tasks.

Uses of Python

- Database and web frameworks
  - Django, Flask, Google App Engine
- Numerical programming, data analysis, machine learning
  - NumPy, Pandas, scikit-learn
- Software development lifecycle
  - `unittest`, Buildbot, Sphynx
- Portable systems programming
  - The `os` and `sys` standard libraries

See more at `https://www.python.org/about/apps/` and `https://www.python.org/about/success/`.

Python 2 vs 3

In Dec 2008, Python 3.0 was introduced as the

- The first backwards incompatible of Python.
- Extensive new support for iterators, functional programming, and metaprogramming.
- See `https://docs.python.org/3/whatsnew/3.0.html`

A huge number of production codes still rely on Python 2.7

- End-of-life for Python 2.7 is Jan 1, 2020
- `https://devguide.python.org/#status-of-python-branches`

In this class, we will use Python 3.7

Table of Contents

What's in a Python installation?

- The interpreter: The program that runs your code
- The standard library: Modules specified by the language standard
  `https://docs.python.org/3.7/library/`
- Third-party libraries: Installed and managed by admins or user

Options for Installing Python

- Pre-installed by system
  - Not recommended
  - Often out-of-date. Sometimes difficult to manage modules.
- Your own system-wide installation
  - Not recommended
  - Sometimes difficult to manage modules.
- Separate installations per user and/or project
  - Highly recommended
  - Don't need admin privileges
  - Easy to keep up-to-date and manage modules
  - Easy to recreate reproducible, deployable environments

Conda, Anaconda, and Miniconda

- Conda
  - An open-source installation manager for Python and other languages
  - `https://docs.conda.io/en/latest/`
- Anaconda
  - A Python distribution with the interpreter, standard library, and ¿ 200 common third-party modules
  - Uses conda to install or manage modules
  - `https://docs.anaconda.com/anaconda/`
- Miniconda
  - A slimmer distribution with the interpreter, standard, library, and conda
  - Uses conda to install and manage modules
  - `https://docs.conda.io/en/latest/miniconda.html`

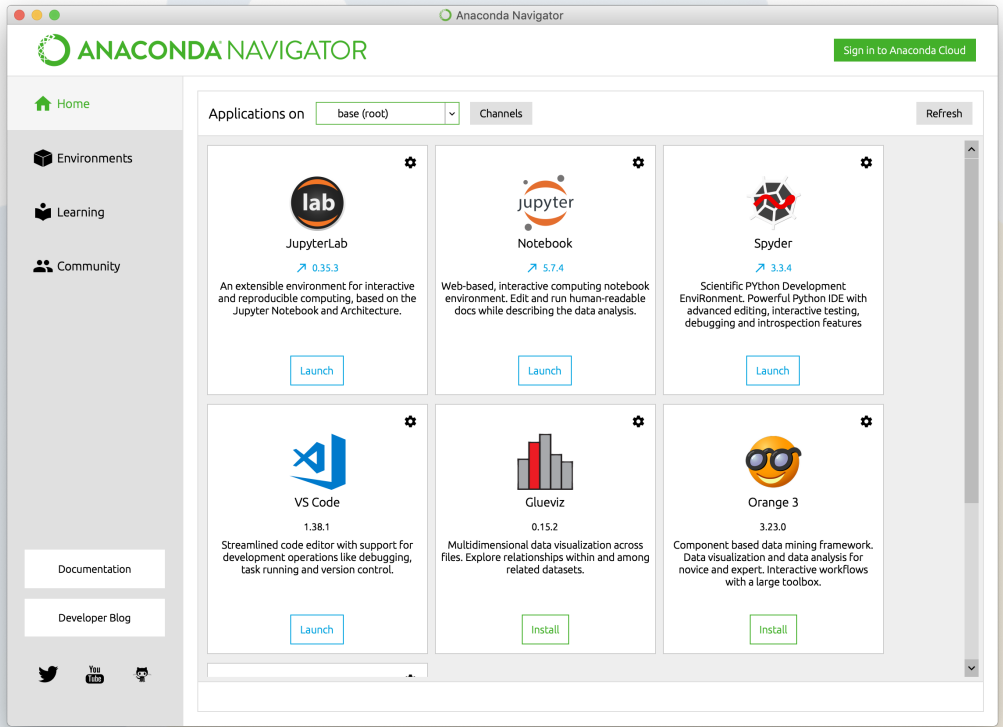
Installing Anaconda

1. Download at: `https://www.anaconda.com/distribution/`
2. Run the installer:
   `http://docs.anaconda.com/anaconda/install/`
3. Recommended (not required) options:
   - Don't install as admin.
   - Use default installation location, such as:
     - `\$HOME/anaconda3` on Mac and Linux
     - `C:\Users\username\Anaconda3` on Windows
   - Install for "Just Me (recommended)"
   - Unselect "Add Anaconda to my PATH environment variable"
   - Select "Register Anaconda as my default Python 3.7"

Table of Contents

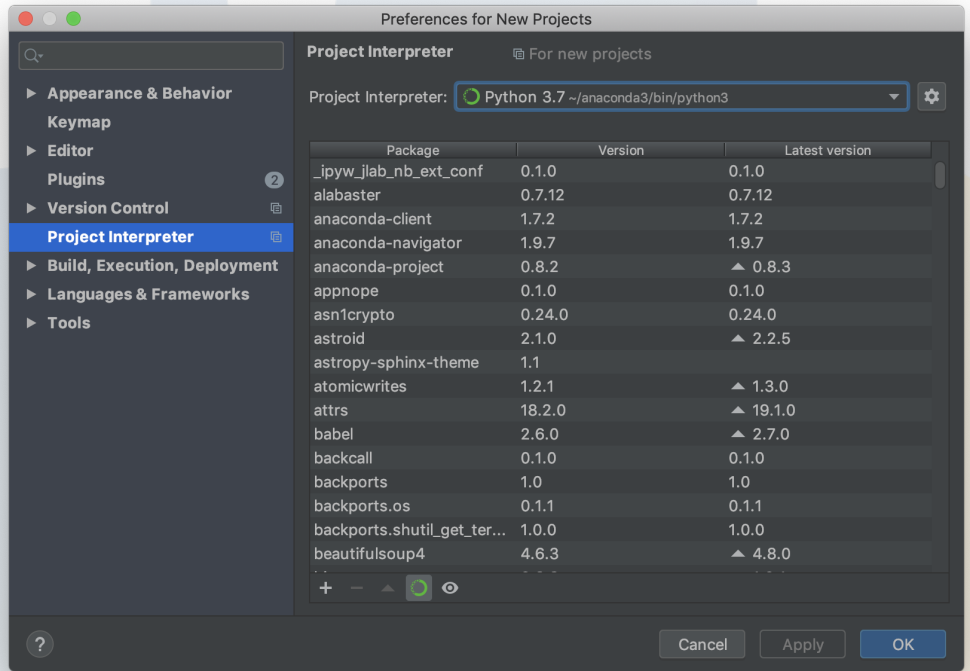
Anaconda Navigator

- Uniform experience across Windows, macOS, and Linux
- Interactive environments
  - JupyterLab
  - Jupyter Notebook
- IDEs
  - Spyder
  - VS Code



PyCharm

- A powerful cross-platform IDE
- Not installed with Anaconda, but thoroughly compatable with an Anaconda installation
- In “Preferences” → “Project Interpreter”’, choose your Anaconda interpreter from the dropdown menu



Using Python From the Shell on macOS and Linux

- Use a normal terminal app
- This is probably in your `.bashrc` or `.bash_profile`

```
# >>> conda initialize >>>
# !! Contents within this block are managed by 'conda init' !!
__conda_setup="$('/anaconda3/bin/conda' 'shell.bash' 'hook' 2> /dev/null)"
if [ $? -eq 0 ]; then
    eval "$__conda_setup"
else
    if [ -f "/anaconda3/etc/profile.d/conda.sh" ]; then
        . "/anaconda3/etc/profile.d/conda.sh"
    else
        export PATH="/anaconda3/bin:$PATH"
    fi
fi
unset __conda_setup
# <<< conda initialize <<<
```

- If not, run this command once in the terminal:

```
$ ~/anaconda3/conda init
```

Using Python from the Shell on Windows

- Easiest way is "Anaconda Powershell Prompt"
- Installed with Anaconda
- Opened from Start Menu

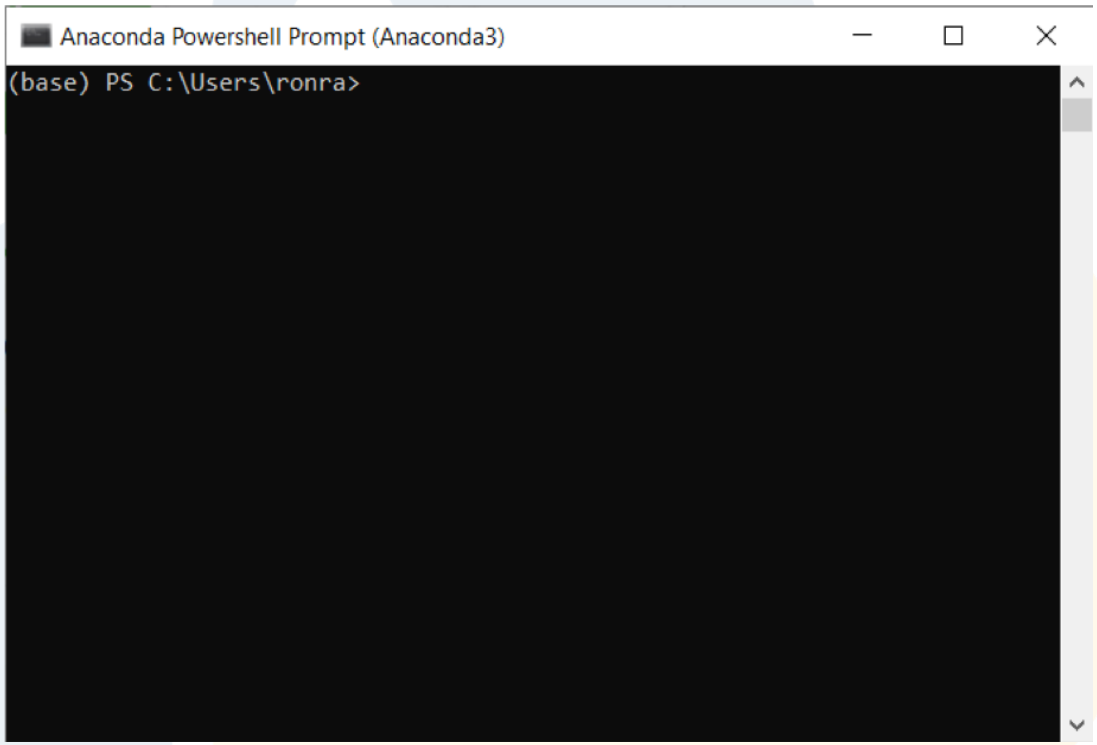


Table of Contents

Table of Contents

Running "Hello, world!"

Our "Hello, world!" is in a plaintext file called "hello.py".

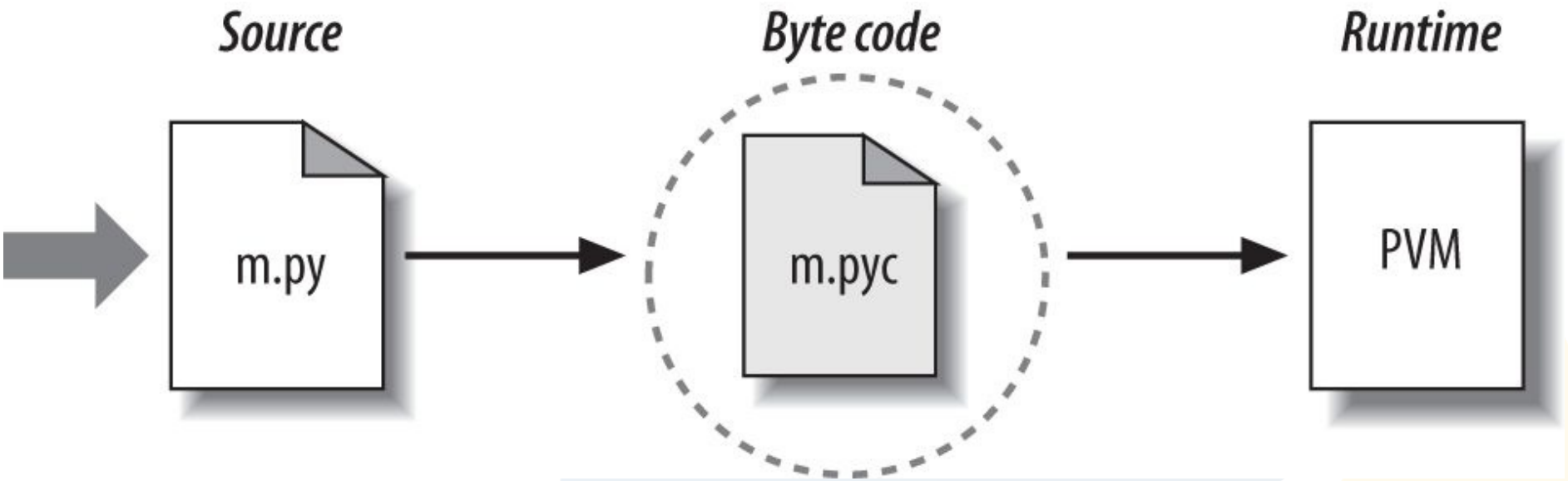
hello.py

```
print("Hello, world!")
```

The interpreter (a program named `python3`) executes our script:

```
$ python3 hello.py
Hello, world!
```

Runtime Execution Model

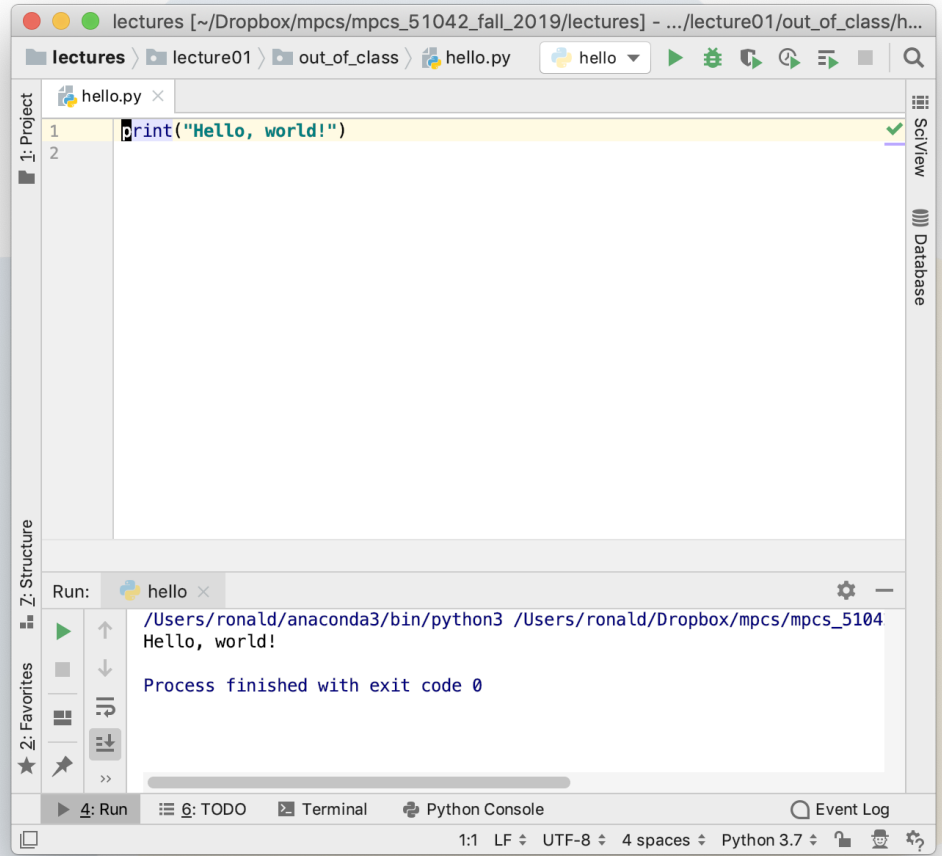


When you run the interpreter:

1. The Python source code is compiled into byte code.
   - A hardware-independent instruction set.
   - Cached on disk. Recompiled if source or interpreter change.
2. The byte code is executed by the Python Virtual Machine (PVM).
   - Runs byte code as hardware-specific CPU instructions.

Running Inside Your IDE

- Running from IDE has the same execution model as on the command line.
- The interpreter is run as a subprocess.



Interactive Sessions

- In an interactive session, lines of code are executed one at a time
- Running `python3` with no args will start a session in the shell.
  - Shows stdout, stderr, and the return value (if any) of each line

```
>>> print('Hello, world!')  # This shows stdout
Hello, world!
>>> 'Hello, world!'         # This shows a return value
'Hello, world!'
```

- Running `ipython3` starts iPython in the shell.

```
In [1]: print('Hello, world!')  # This shows stdout
Hello, world!

In [2]: 'Hello, world!'         # This shows return value
Out[2]: 'Hello, world!'
```

- Most IDEs can open an interactive session, too.

Jupyter Notebooks and JupyterLab

- Jupyter Notebooks and JupyterLab are GUI-based interactive sessions
- Easy to launch from Anaconda Navigator
- Saves notebooks as portable .ipynb files

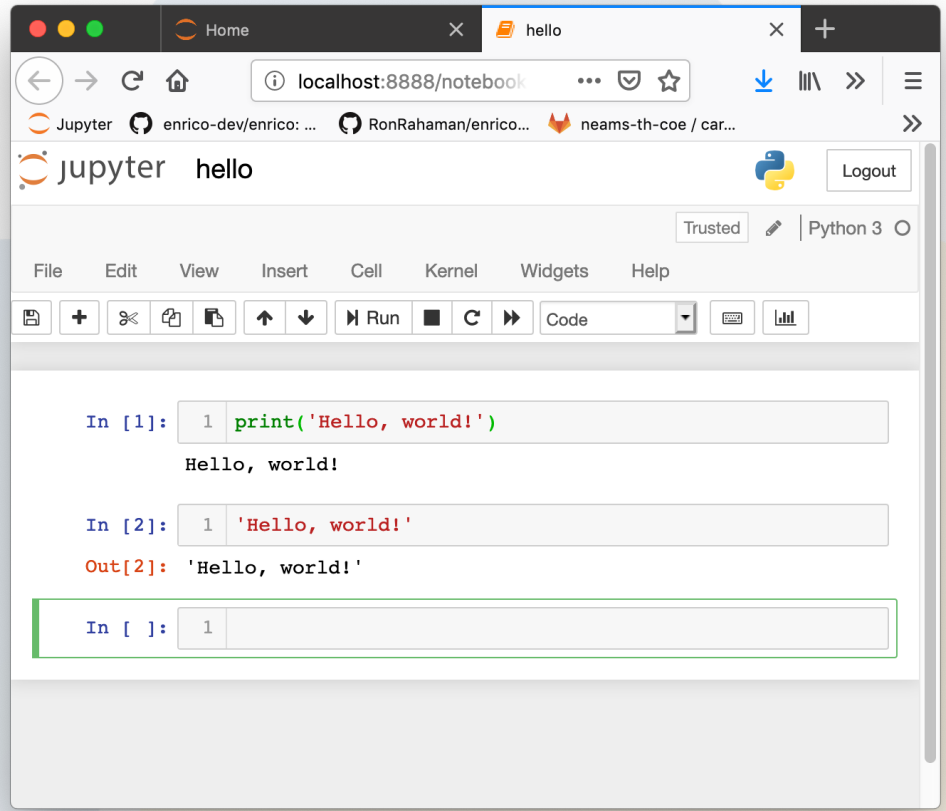


Table of Contents

Dynamically-Typed Variables

In Python, variables are not declared with a specific type.

deceptively_simple.py

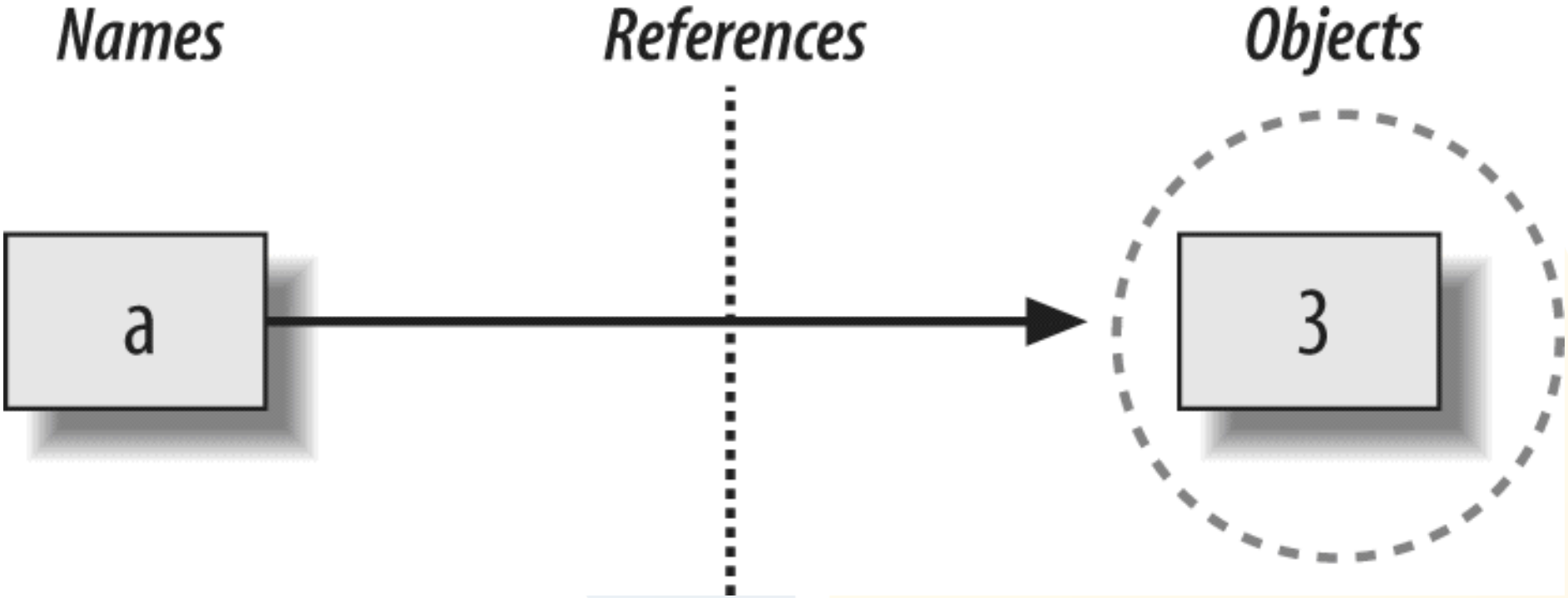
```
a = 3
print(a)
```

In the assignment statement:

1. A new integer object is created for 3.
2. A new variable name is created for a.
3. A reference is created from the name to the object.

Variable names do not have types. Objects do.

What Happens After `a = 3`



Reference: *Learning Python, 3rd Ed.*, Chapter 6

Table of Contents

Table of Contents

General Categories For Built-in Types

For types in the same category, many similar operations are available:

Category	Specific Types	Operations
Numeric	`int`, `float`, `complex`	Addition, multiplication, ...
Sequence	`str`, `list`, `tuple`	Indexing, slicing, concat, ...
Mapping	`dict`	Indexing by key, sorting by key, ...
Set	`set`	Membership, union, difference, ...

Reference: `https://docs.python.org/3/library/stdtypes.html`

Table of Contents

Numeric Types

- The built-in types are integers, floats, and complex numbers:

```
int1 = 3
int2 = 0x1f         # Hex, oct, and binary (0x, 0o, 0b)

float1 = 3.0
float2 = 1e-3       # Scientific notation

cplex1 = 3.0+4.0j
cplex2 = 3+4j       # Real/imag parts are always floats
cplex3 = 4j         # The real part is optional
```

- Types can be explicitly converted using the `int()`, `float()`, and `complex()` functions
- Specialized numeric types (such as `Decimal` and `Fraction`) can be imported from standard library modules.

Sizes of Numeric Types

- Integers have "unlimited" ranges, internally implemented through two types
  - Values $< |2,147,483,647|$ are represented in 32-bit signed format
  - Larger values are represented by a variable-size signed integer type
  - Conversion from 32-bit to variable size is handled by the interpreter and hidden from the user
  - This all but eliminates integer overflow
- Floats are represented in 64-bit ("double precison") floating point format
  - Limits are about $\pm 10^{308}$
  - For comparison, there are an estimated 10^{82} atoms in the observable universe

Numeric Operations

- The usual operations (+, -, *, %) as well as:
  - Exponentiation: `x ** y`
  - True division: `x / y`
  - Floor division: `x // y`
- Usual operator precedence is respected
  - `https://docs.python.org/3/reference/expressions.html#operator-precedence`
- When mixing types, the result is up-converted to the highest type.
  - `int` < `float` < `complex`

```
>>> 6 * 3.0
18.0
>>> 6.0 * (3+4j)
(18+24j)
```

True vs. Floor Divison

There are two division operators in Python:

- `x / y` always performs true division (keeps remainder)
- `x // y` always performs floor division (truncates result)

```
>>> 10 / 4.0          # Result is a float w/remainder
2.5
>>> 10 / 4            # Result is also a float w/remainder
2.5
>>> 10 // 4.0         # Result is a truncated float
2.0
>>> 10 // 4           # Result is a truncated int
2
```

The `math` Module: A First Look at Importing

- Other common operations (floor, ceil, log, trig) and constants (π, e) are provided by the standard library `math` module.
- A module is a namespace for variables, functions, and classes
- After import, a module's tributes are referenced via the module name:

```
import math
x = math.log2(8.0)      # Base-2 log of 8
```

- You can refer to attributes directly via the `from x import y` syntax.
- Use sparingly to avoid polluting the current namespace.

```
from math import log2
x = log2(8.0)           # Same function
```

Table of Contents

Strings in Python

Strings are immutable sequences of characters or bytes.

- Sequence: Maintains an order of elements
- Immutable: Cannot be changed in-place. A string transformation always returns a new object.

In normal usage, strings are interpreted as character sequences. Can be manipulated as raw bytes, if necessary.

String literals

- Double- and single-quotes both define strings:

```
s1 = 'Ron\'s class'    # Have to escape the single-quote
s2 = "Ron's class"     # Same, but don't need escape
```

- The usual escape characters (`\n`, `\t`) are understood:

```
s3 = "Today is:\nMonday"     # A newline character
```

- Raw strings (`r'...'`) ignore escape characters:

```
s4 = r's/\t/ {4}/g'    # Useful for regular expressions
```

- Triple-quoted strings include any newlines

```
s4 =  \
""" Example docstring for some function
Args:
    param1: The first parameter.
"""
```

Sequence Operations on Strings

Strings support sequence operations such as:

- `s1 + s2` concatenates two strings
- `s1 * 3` repeats a string
- `len(s1)` returns the length of the string
- `s1 in s2` tests membership

These operations will apply to other sequence objects, too.

```
s = "over and "
s *= 3              # "over and over and over and "
s += "over"         # "over and over and over and over"
len(s)              # 31
"over" in s         # True
"under" in s        # False
```

Indexing with Strings

Indexing (`s[i]`) fetches a reference to a single element:

- Python supports positive indices on $[0, \texttt{len(s)})$
- Also supports negative indices on $[\texttt{-len(s)}, 0)$

```
>>> s = 'queso'
>>> s[0]
'q'
>>> s[-2]
's'
```

Indexing out-of-range raises an `IndexError` at runtime.

```
>>> s[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Slicing with Strings

Slicing (`s[i:j]`) fetches a copy of a substring on `[i, j)`:

```
>>> s = 'jalapeno'
>>> s[1:4]
'ala'
>>> s[4:-1]
'pen'
```

If `i` is omitted, the left bound is the beginning (inclusive)

```
>>> s[:4]
'jala'
```

If `i` is omitted, the right bound is the end (inclusive)

```
>>> s[2:]
'lapeno'
```

Extended Slicing with Strings

Extended slicing (`s[i:j:k]`) allows strided and reverse-indexing.

A positive stride (`k`) return the subsequence: `[i, i+k, i+2*k, ...)`

- The lower bound `i` is inclusive and defaults to `0`
- The upper bound `j` is non-inclusive and defaults to `len(s)`

```
>>> s = 'abcdefghi'
>>> s[1:7:2]
'bdf'
>>> s[::2]
'acegi'
```

Extended Slicing with Strings, cont.

With a negative stride `k`, the list is traversed in reverse order, and the upper and lower bounds are switched. We extract the subsequence: `[i, i-k, i-2*k, ...)`

- The upper bound `i` is inclusive
- The lower bound `j` is non-inclusive

```
>>> s = 'abcdefghi'
>>> s[5:1:-2]
'fd'
>>> s[::-2]
'igeca'
```

Some Useful String Methods

Some methods of string instances (where `s` is the instance):

- `s.startswith(prefix)`, `s.endswith(suffix)` : Return `True` if `s` starts/ends with the given prefix/suffix.
- `s.upper()`, `s.lower()`, `s.casefold()`: Return an uppercase/lowercase/casefolded copy of `s`.
- `s.lstrip()`, `s.rstrip()`, `s.strip()`: Return a copy of `s` with the leading/trailing/leading-and-trailing whitespace removed.
- `s.replace(old, new)`: Return a copy where all occurrences of `old` are replaced with `new`.
- `s.split(sep=None)`: Return a list of the words in the string, using `sep` as the delimiter. If `sep` is not given, then split on whitespace.
- `s.join(iterbl)`: Return a string which is the concatenation of the strings in an iterable, using `s` as the delimiter.

Complete list: `https://docs.python.org/3/library/stdtypes.html#string-methods`

String Demos

1. Remove unnecessary whitespace from this line of input:

```
s1 = "   I'm a sentence to parse.    \n"
```

2. Convert Windows newlines to UNIX newlines:

```
s2 = "I'm a Windows line.\r\n"
```

3. Convert tabs to spaces:

```
s3 = "for i in s:\n\tprint(i)\n"
```

Table of Contents

String Formatting

- A string object has the `format()` method that returns a new string with substituted values
- The original string is a template that can have several forms.
  - Relative position: Note that int is auto converted

```
>>> "I ate {} {} and {}".format(15, 'apples', 'bananas')
'I ate 15 apples and bananas'
```

  - Keyword

```
>>> "I ate {count} {food1} and {food2}".format(
...         food1='apples', food2='bananas', count=15)
'I ate 15 apples and bananas'
```

  - f-strings: New in Python 3.6

```
>>> count=15; food1='apples'; food2='bananas'
>>> f'I ate {count} {food1} and {food2}'
'I ate 15 apples and bananas'
```

String Formatting, cont.

There are many extra options for the string formatting method which include:

- Displayed precision
- Hex, binary, or octal representation for ints
- Whitespace padding and alignment

For reference, see:

- *Learning Python*, Chapter 7, "String Formatting Method Calls"
- `https://docs.python.org/3.7/library/string.html#format-specification-mini-language`

The `print()` Function

- The `print()` function prints objects to stdout (by default)
- Takes an arbitrarily number of objects and keyword arguments.
- Prints objects on one line, separated by a space (by default)

```
>>> print('over', 'under', 'through')
over under through
```

- The values are converted to strings automatically

```
>>> x = 3
>>> y = 7
>>> print(x, '/', y, '=', x/y)
3 / 7 = 0.42857142857142855
```

- What's a good way to limit precision of output?

Keyword Arguments for `print()`

- sep : The string that separates objects (default: `' '`)

```
>>> print('over', 'under', 'through', sep=' and ')
over and under and through
```

- end : The string to print at end (default: `'\n'`)

```
>>> print('over', 'under', 'through', end='!\n')
over under through!
```

- file : An open file or stream to write to (default: `sys.stdout`)

```
>>> import sys
>>> print('over', 'under', 'through', file=sys.stderr)
over under through
```

- flush : Flush output immediately (default: `False`)

Input from console

The `input()` built-in function does the following:

- Prompt the user for input, with an optional message displayed
- Reads the user's input until a newline is entered
- Returns the input as a string (with the newline removed)

What's the problem here and how do we fix it?

input_basic.py

```
print("Here's my addition machine!")
x = input("Enter the first operand: ")
y = input("Enter the second operand: ")
print("The sum is", x + y)
```

Table of Contents

Lists

A `list` is an ordered collections of references to other objects.

- Sequences: Can use all the sequence operations we covered above.
- Heterogenous: Can mix different object types in a single list.
- Can be arbitrarily nested.
- Dynamically-resizable: Can shorten or lengthen as needed.
- Mutable: Can change elements in-place.

Sequence Operations on Lists

Lists support the same sequence operations we saw on strings:

- `s1 + s2` concatenates two lists
- `s1 * 3` repeats a list
- `len(s1)` returns the length of the list
- `s1 in s2` tests membership

```
L = [1, 2]
L += ['a', 'b'] # [1, 2, 'a', 'b']
L *= 2          # [1, 2, 'a', 'b', 1, 2, 'a', 'b']
len(L)          # 8
'a' in L        # True
3 in L          # False
```

Indexing and Slicing with Lists

Lists support indexing, slicing, and extended slicing. Recall that slicing returns a copy of the subsequence.

```
>>> L = [10, 11, 12, 13, 14, 15]
>>> L[1]
11
>>> L[1:4]
[11, 12, 13]
>>> L[::-1]
[15, 14, 13, 12, 11, 10]
>>> L[:]
[10, 11, 12, 13, 14, 15]
```

Example: Nested Lists

A nested list can (poorly) emulate a matrix with row-major ordering:

```
>>> m = [[1,2,3],[4,5,6],[7,8,9]]
>>> m[2][0]    # Get one element
7
>>> m[1][:]    # What's this result?
>>> m[:][1]    # What about this one?
```

Third-party libraries (e.g., NumPy) provide matrix classes with many better characteristics.

Mutable vs. Immutable Objects

Recall that strings are immutable but lists are mutable.

- One consequence: String elements/subsequences cannot be changed, but list elements/sequences can:

```
>>> S = "Now it's Sunday morning"
>>> L = ["Now", "it's", "Sunday", "morning"]
>>> S[9:] = "Monday evening"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> L[2:] = ["Monday", "evening"]
>>> L
['Now', "it's", 'Monday', 'evening']
```

String methods (like `replace`) produce a new string instance. Many list methods change the list in-place.

Methods for both Immutable and Mutable Sequences

Some methods that apply to both strings and lists (and all sequence objects). Here, `s` is an instance of a sequence.

- `s.index(x)`: Return the index of the first item whose value equals `x`. Optional arguments for start/end index of search.
- `s.count(x)`: Return the number of occurences of items equal to `x`.

See also: `https://docs.python.org/3/library/stdtypes.html#typesseq-common`

Methods for Mutable Sequences

Some methods of list instances (where `L` is the instance). All of these change the list in-place. Also apply to other mutable sequences.

- `L.append(item)`: Add a single item to the end of the list.
- `L.extend(iterbl)`: Append every item of the iterable to the list.
- `L.insert(i, x)`: Insert item `x` at position `i`.
- `L.pop()`: Remove and return the last item in the list.
- `L.pop(i)`: Remove and return the item at position `i`.
- `L.remove(x)`: Remove the first item whose value equals `x`
- `L.sort()`: Sort list in place. Optional arguments for reverse sort, etc.
- `L.reverse()`: Reverse list in place.

See also: `https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types`

append vs. extend

- The append method is used to add one element to a list.

```
>>> months = ['Jan', 'Feb', 'Mar']
>>> months.append('Apr')
>>> print(months)
['Jan', 'Feb', 'Mar', 'Apr']
```

- The extend method is used to add multiple elements to a list.

```
>>> months.extend(['May', 'Jun'])
>>> print(months)
['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
```

Misusing `append` and `extend`

- What happens if we append a list onto a list?

```
>>> months = ['Jan', 'Feb', 'Mar']
>>> months.append(['Apr', 'May'])
```

- What happens when we extend a string onto a list?

```
>>> months = ['Jan', 'Feb', 'Mar']
>>> months.extend('Apr')
```

Exercise: Stacks and Queues

- Show how list methods can be used to implement a stack data structure.
- Show how list methods can be used to implement a queue data structure.

The `split` method

- `split` is a string method returns a new list of strings, based on a given separator.

```
>>> line = "Ron Rahaman,rahaman@cs.uchicago.edu,05/25/1985"
>>> fields = line.split(',')
>>> print(fields)
['Ron Rahaman', 'rahaman@cs.uchicago.edu', '05/25/1985']
```

- With no separator specified, the string is split on any whitespace, with consecutive whitespace treated as one separator.

```
>>> sentence = "Here is a string\twith \nwhitespace."
>>> words = sentence.split()
>>> print(words)
['Here', 'is', 'a', 'string', 'with', 'whitespace.']
```

The `join` method

The string method `s.join(L)` takes a list, `L`, and joins the elements into a new string, using the string `s` as the separator.

```
>>> words = ['Here', 'is', 'a', 'sentence.']
>>> sentence = ' '.join(words)
>>> print(sentence)
Here is a sentence.
```

Exercise: Formatting sentences

Write a program that re-formats sentences according to the conventions of your 70-year-old typing instructor:

- The program prompts for some sentences.
- The returned sentence is formatted with one space between words and two spaces between sentences.

Table of Contents

Truth Testing and Boolean Operators

Truth testing includes:

- Symbols: `==`, `!=`, `>`, `<`, `>=`, `<=`,
- Keywords:
  - `in` tests if an element is in a collection
  - `is` tests if two objects are identical (as opposed to equal)

Boolean operators include:

- Keywords for usual logical operations: `not`, `and`, `or`. These short-circuit.
- Symbols for bitwise operations: `~`, `&`, `|`,

Boolean values are built-in constants: `True` and `False`

- Many other values can be interpreted as true or false: `https://docs.python.org/3/reference/expressions.html#operator-precedence`
- It's common to use `None` and empty containers in truth testing. Use other values sparingly.

Precedence of Boolean Operations

- The `not`, The `and`, and `or` operators have different precendences.
- See `https://docs.python.org/3.7/reference/expressions.html#operator-precedence`
- Precedence can be adjused with parenthesis
- Example: What is the value of this?

```
True or False and False
```

Comparing Sequence

- Sequences are compared using lexicographical ordering (`https://docs.python.org/3.7/tutorial/datastructures.html#comparing-sequences-and-other-types`)
- Start comparing first items in each sequence
  - If comparison returns false, stop and return false
  - If comparison returns true, and there are remaining items in both sequences, repeat with next pair items items
  - If the comparison returns true, and there are no more items in both sequences, return true
  - If comparison returns true, and only one sequence has no more items, the shorter sequence is treated as lesser
- What are the truth values of these?

```
[1, 2, 3] == [1, 2, 3]
[1, 2, 0] < [1, 2, 3]
[1, 2, 4] > [1, 2, 3, 4]
```

If/else Statements

General syntax of an if/else statement where:

- Both the `elif` and `else` statements are optional.
- Parentheses around test are optional.
- A block is single- or multi-line statements with the same indentation level. Multiples of 4 spaces are required by standard.

```
if test1:
    block1
elif test2:
    block2
else:
    block3
```

While Loops

A **while** loop can contain many optional statements. All statements are optional except **while** test:

```
while test1:
    statements1      # Runs in every iteration.
    if test2:
        break        # Go directly to statements4.
    if test3:
        continue     # Go directly to next iteration.
    statements2      # Runs if break or continue
                     #     was not encountered.
else:
    statements3      # Runs if loop exits without a break.
statements4          # Runs no matter how loop exits.
```

Example: A simple interactive prompt

Write a program that:

- Repeatedly prompts the user for a number.
- After each prompt, shows the cumulative sum of all the numbers so far.
- If the user enters something other than a number, raise an error message and reprompt.
- If the word "exit" is entered, end the program.

Example: Online mean

Write a program that: Write a program that:

- Repeatedly prompts the user for a number.
- After each prompt, shows the mean and variance of all values so far.
- If the user enters something other than a number, raise an error message and reprompt.
- If the word "exit" is entered, end the program.

Use Welford's online algorithm (`https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's_online_algorithm`) to accomplish this.

Table of Contents

Iterables and For Loops

The `for` loop supports any iterable object:

- A physically-stored collection of data
- An object that generates elements on-the-fly, one at a time

Many, many built-ins (strings, lists, dicts, tuples, files) are iterable.

In every iteration of this `for` loop, the variable `item` refers to the subsequent object in `collection`. Other statements are optional

```
for item in iterable:
    statements1          # Run in every iteration
    if test: break       # Immediately go to statements3
    if test: continue    # Immediate go to next iteration
    statements2          # Run if continue not encountered
else:
    statements           # Run if break not encountered
statements3              # Run no matter how loop exits
```

Using Lists in For Loops

- Find the sum of a list

```
nums = [1, 4, 6, 0, 2]
total = 0
for x in nums:
    total += x
```

- Find the max value of a list

```
biggest = nums[0]
for x in nums[1:]:
    if x > biggest:
        biggest = x
```

Better ways to do both of these...

Useful Functions that Consume/Produce Iterators

- `range(start, stop[, step])` returns integers from `start` (inclusive) to `stop` (non-inclusive) with optional step.
- `enumerate(iterbl[, start])` returns a count, value pair in each iteration.
- `zip(iter1, iter2[, iter3, ...])` aggregates iterables.
- `sorted(iterbl)` returns a list of sorted items in iterable
- `reversed(seq)` returns an iterator to traverse sequence in reverse
- `sum(iterbl)`, `max(iterbl)`, `min(iterbl)` returns the sum/maximum/minimumm of all items in iterable
- `list(iterbl)` produces a list from the entire iterable

See also: `https://docs.python.org/3/library/functions.html#built-in-functions`

Using `range()`

A `range()` object is an iterable that represents a sequence of integers.

- `range(start)` generates integers from $[0, \texttt{start})$
- `range(start, stop)` generates integers from $[\texttt{start}, \texttt{stop})$
- `range(start, stop, step)` supports positive or negative `step`
  - Both cases follow recurrence $x_{i+1} = x_i + \texttt{step}$
  - For positive `step`, output is $[\texttt{start}, \texttt{stop})$ where $\texttt{start} > \texttt{stop}$
  - For negative `step`, output is $[\texttt{start}, \texttt{stop})$ where $\texttt{start} < \texttt{stop}$

What are these results?

```
>>> list(range(5))
>>> list(range(1, 6))
>>> list(range(0, 30, 5))
>>> list(range(0, 10, 3))
>>> list(range(0, -5, -1))
```

https://docs.python.org/3.7/library/stdtypes.html#ranges

Using `enumerate()`

Suppose you wanted to get each element of a list and its count. Some naive ways are:

```
months = ['Jan', 'Feb', 'Mar']

# Method 1
for i in range(len(months)):
    print(i+1, ':', months[i])

# Method 2
i = 1
for m in months:
    print(i, ':', m)
    i += 1
```

Method 1 is not a general solution for all iterables. Why?

Using `enumerate()`, cont.

- The `enumerate(iterable, start)` returns an another iterable which yields a pair of (count, item) values from the input `iterable`
- If `start` is omitted, it defaults to 0

```
months = ['Jan', 'Feb', 'Mar']

for i, m in enumerate(months, start=1):
    print(i, ':', m)
```

Using `zip()` to Transpose a Table

How would you pivot (or transpose) this table?

```
camera_ids =   ['CHI149',              'CHI045',      'CHI021']
camera_addrs = ['4909 N CICERO AVE', '445 W 127TH', '2900 W OGDEN']
violations =   [293291,                233144,        186225]

table = [camera_ids, camera_addrs, violations]

for row in table:
    print(*row, sep=', ') # Don't need to know this syntax
```

Output:

```
CHI149, CHI045, CHI021
4909 N CICERO AVE, 445 W 127TH, 2900 W OGDEN
293291, 233144, 18622
```

Using `zip()` to Transpose a Table, cont.

```
pivot_table = []
for pivot_row in zip(camera_ids, camera_addrs, violations):
    pivot_table.append(list(pivot_row))

print("Camera ID", "Camera Addr", "Violations", sep=', ')
for row in pivot_table:
    print(*row, sep=', ') # Don't need to know this syntax
```

Output:

```
Camera ID, Camera Addr, Violations
CHI149, 4909 N CICERO AVE, 293291
CHI045, 445 W 127TH, 233144
CHI021, 2900 W OGDEN, 186225
```