Lecture 01: Fundamental Data Types

MPCS 51042-1 : Python Programming

Ron Rahaman

The University of Chicago, Dept of Computer Science

Oct 1, 2018

Table of Contents

Table of Contents

Course Objectives

- Learn object-oriented and functional programming in Python.
- Learn vital standard library and third-party library tools.
- Learn to write high-quality, reusable code.

Prerequisites

MPCS "Immersion Programming" and "UNIX Bootcamp" or equivalent knowledge:

- Experience with procedural programming in any language
- Familiarity with basic object-oriented concepts
- Proficiency with your operating system's command line

Ask Ron if you have doubts about your existing knowledge.

Syllabus

Week	Date	Topics	Assignment Due
1	Oct 1	Intro and Data Types	
2	Oct 8	More Data Types and Program Units	Homework 1
3	Oct 15	Functional Programming	Homework 2
4	Oct 22	Object-Oriented Programming 1	Homework 3
5	Oct 29	In-class midterm	
6	Nov 5	Object-Oriented Programming 2	Homework 4
7	Nov 12	Quality Assurance (testing, docs)	Homework 5
8	Nov 19	Numeric and Data Processing	Homework 6
9	Nov 26	Mystery Special Topic!	Homework 7
10	Dec 3	No class	Homework 8
11	Dec 10	In-class Final	

Course Structure

Breakdown of overall grade:

- 8 weekly take-home coding 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.

Staff and Office Hours

Name	Hours	Location
Ron Rahaman	Thursdays, 4:30pm - 6:30pm	Hyde Park, Room TBA
Rob Alef	Sundays, 10:00am - 12:00pm	Gleacher Center, cafe
Grader 1		
Grader 2		

Websites

- Piazza (`https://piazza.com/`) will be used for discussions and distributing homeworks assignments, lecture notes, etc. Ask Ron if you have not been added.
- GitLab (`https://mit.cs.uchicago.edu/`) will be used for submitting homeworks. More about using it later in presentation.

Textbooks

Required:

- Mark Lutz, *Learning Python*, 5th Edition.
- Scott Chacon, Ben Straub. *Pro Git*, 2nd Edition. Freely available at `https://git-scm.com/book`

Highly Recommended:

- Luciano Ramalho, *Fluent Python*.
- Brian Jones, David Beazley. *Python Cookbook*, 3rd Edition.

Academic Honesty

- Cite every piece of code that is based on an outside source. This includes:
  - Student interactions (cite student's name)
  - Textbook (cite title, chapter, section)
  - Internet (cite URL)
- 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
- Language syntax doesn't obscure your algorithm.
- Powerful built-in data structures
- Execution speed is excellent for most tasks.
  - In CPython (most common implementation), built-in objects are written in highly optimized C.
  - When necessary, user-defined objects can be linked to high-performance libraries written in another language.

Downsides of Python

- Specialized tasks may perform slowly with Python standard library.
  - Numeric/data heavy work, parallel programming.
  - May need to implement in a compiled language and link to Python.
  - Good news: Many third-party libraries are available for these tasks.

Uses of Python

- Portable systems programming
  - The `os` and `sys` standard libraries
- GUIs
  - tkinter, Qt, .NET
- Database and web frameworks
  - Django, Flask, Google App Engine
- Numeric and data analysis
  - NumPy, Pandas
- Component integration
  - Interoperability with C/C++ on CPUs and GPUs
- Software development lifecycle
  - `unittest`, Buildbot, Sphynx

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 first backwards incompatible of Python. In Python 3.0 and higher, we now have:

- Several keyword statements are now functions.
- Many methods return iterators instead of lists.
- Better support for metaprogramming and functional programming.
- See the changelog for Python 3.0: `https://docs.python.org/3/whatsnew/3.0.html`

There are still many codes that rely on Python 2.

- There is built-in support for running Python 2.7 code in a Python 3 environment. See: `https://docs.python.org/2/library/__future__.html`
- There is no built-in support for running Python 3 code in a Python 2 environment.

Only Python 3 In This Class

In this class, we will be using Python 3 exclusively.

- Virtually no discussion of differences with Python 2.7
- We will assume you're using Python 3.6 (Dec 2016).
- However, virtually everything we need is in Python 3.3 (Sept 2012).
- We will make it very clear if we use features newer than 3.3.

Table of Contents

Strategies for Managing Installations

Your installation will include both Python and common third-party libraries. There are two general ways to manage these:

- Install system-wide using a package manager
  - APT on Ubuntu
  - MacPorts, Homebrew on macOS
- Install environments for individual users and projects (recommended)
  - virtualenv: good solution for managing your own libraries
  - Anaconda: good all-in one solution with the most common libraries

For this course, we recommend (not require) you use Anaconda.

Installing Anaconda

1. Download at: `https://www.anaconda.com/download/`
2. Follow instructions at:
   `http://docs.anaconda.com/anaconda/install/`
3. Recommended (not required) options:
   - Don't install as admin.
   - If prompted, install for just you (not all users).
   - If prompted, add Anaconda to your PATH environment variable.
   - If prompted, make Anaconda your default Python 3.6.

Your Development Environment

- Rapid prototyping:
  - iPython interactive prompt
  - Jupyter Notebook (not for actual homework submissions)
- Text editor & console:
  - Atom
  - Sublime
  - Emacs
  - Vim
- IDEs:
  - Visual Studio Code
  - Eclipse
  - PyCharm

Table of Contents

Table of Contents

Running "Hello, world!"

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

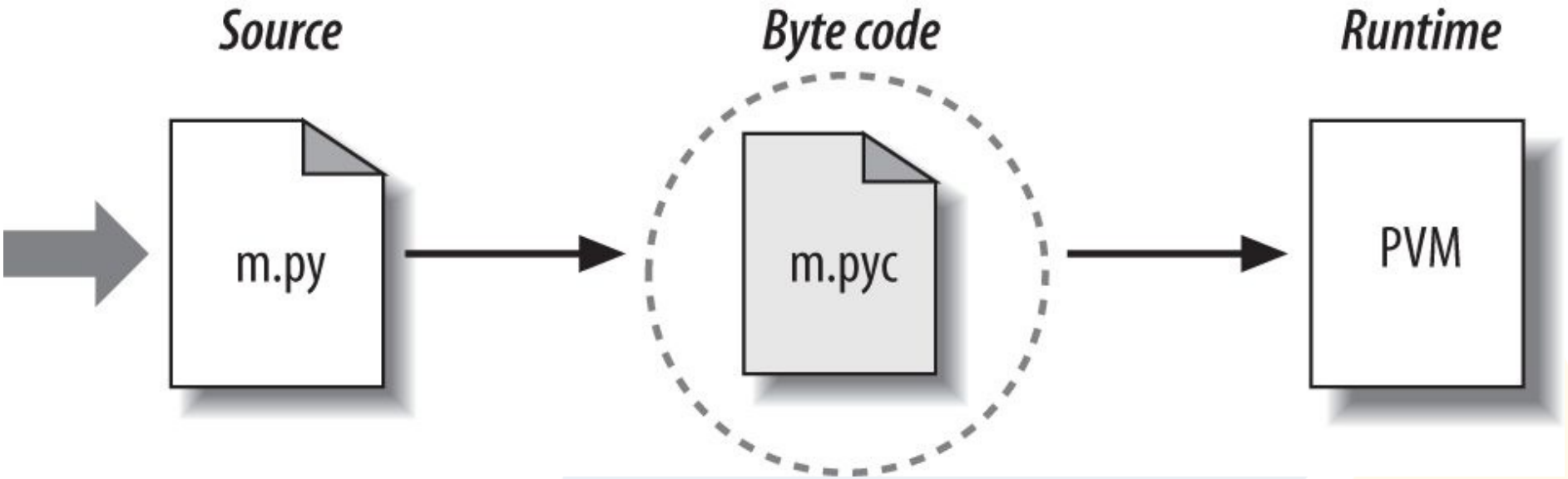
hello.py

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

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

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

CPython's Runtime Execution Model



When you run the interpreter:

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

Table of Contents

Dynamically-Typed Variables

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

deceptively_simple.py

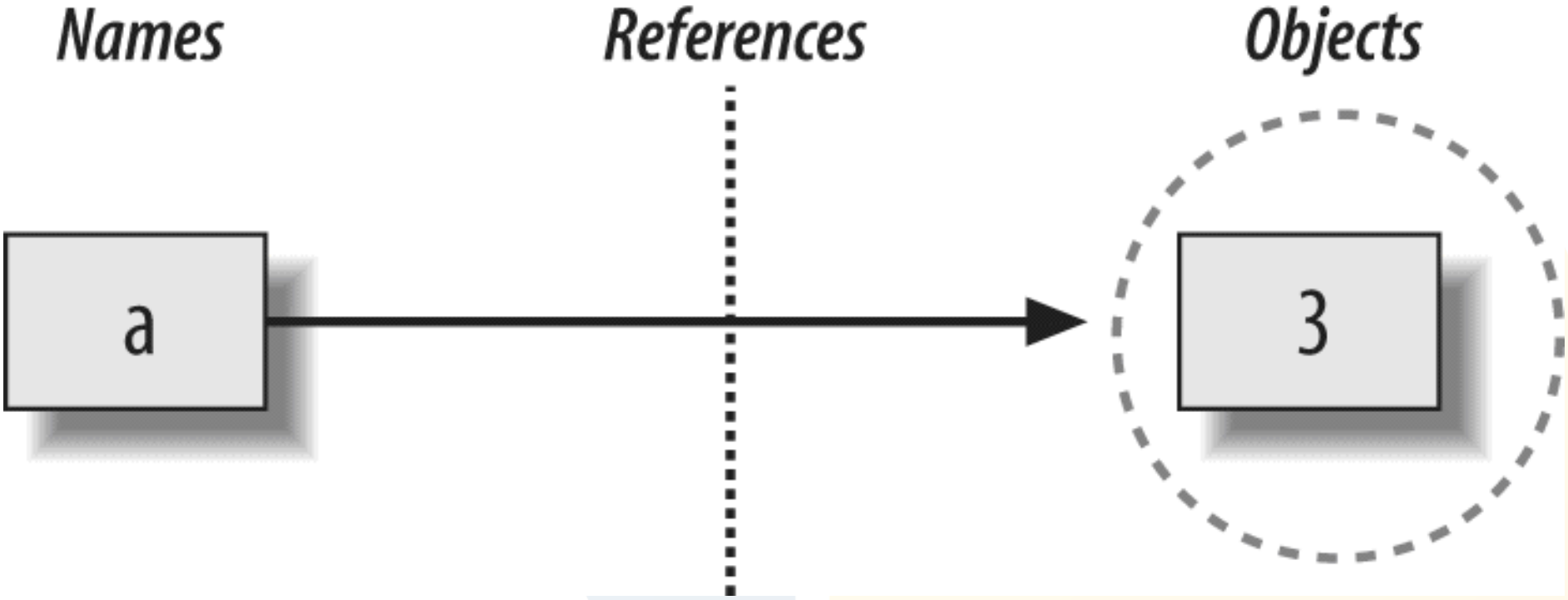
```
a = 3
print(a)
```

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         # Can use hex, oct, and bin literals

float1 = 3.0
float2 = 1e-3       # Can use 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.

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 bytes.

- Sequence: Maintains an order of elements
- Immutable: Cannot be changed in-place. String transformations always produce new instances.

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 and Slicing with Strings

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

- Python supports 0-based indexing on the interval [0, `len(s)`)
- Also supports negative indexing on [`-len(s)`, 0)

```
>>> s = 'spam'
>>> s[0], s[-2]
('s', 'a')
```

Slicing (`s[i:j]`) fetches a copy of a subsequence:

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

```
>>> s[1:3], s[1:], s[:-1]
('pa', 'pam', 'spa')
```

Extended Slicing with Strings

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

A positive stride (`k`) xtracts 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`

Some Demos

From *Python Cookbook* String demos:

- 2.2: Discover file extensions or URL schemes
- 2.4, 2.5, 2.6: Finding and Replacing Text
- 2.11: Stripping Unwanted Characters
- 2.14: Ways to Concatenate Strings

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:

```
>>> 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]
```

Recall that slicing returns a copy of the subsequence.

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][:]   # Get a copy of the whole row
[4, 5, 6]
>>> m[:][1]   # Doesn't get the column  :(
[4, 5, 6]
```

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`

Some Demos

- `L.append()` vs `L.extend()`
- `L.sort()` vs `L.reverse()`

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 e 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
```

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`

Some Demos

Section numbers from *Python Cookbook*

- `L.sort()` vs `sorted(L)`
- `L.reverse()` vs `reversed(L)`
- The scope of loop variable and changing loop variable
- 1.20: Transforming and Reducing Data
- 3.11: Picking Things at Random
- 4.10: Iterating over Index-Value Pairs
- 4.11: Iterating over Multiple Sequences

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.

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.

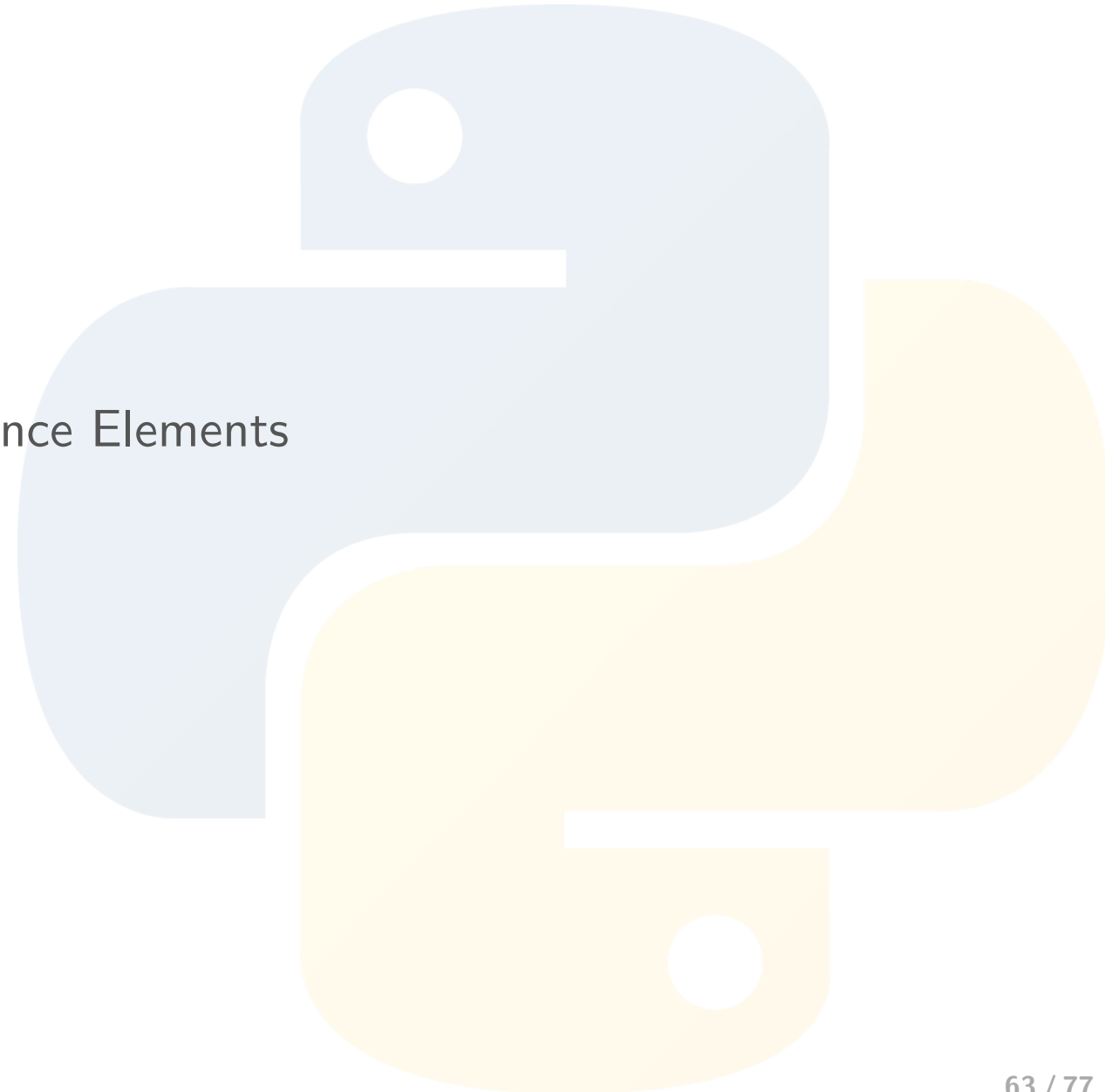
```
if test1:
    block1
elif test2:
    block2
else:
    block3
```

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.
```

Some Demos



From *Python Cookbook*

- 1.16: Filtering Sequence Elements

Some Demos

From *Python Cookbook* String demos:

- 2.15: Variable Interpolation in 3.6 (see also: `https://docs.python.org/3/whatsnew/3.6.html#pep-498-formatted-string-literals`

Table of Contents

Formatted Output

Formatting is done with a template string and the string formatting method:

```
# By relative position.  Note that int is auto converted
print("I ate {} {} and {}".format(15, 'apples', 'bananas'))

# By absolute position
print("I ate {2} {0} and {1}".format(
        'apples', 'bananas', 15))

# By keyword
print("I ate {count} {food1} and {food2}".format(
        food1='apples', food2='bananas', count=15))

# New in Python 3.6: formatted string literals
count=15; food1='apples'; food2='bananas'
print(f'I ate {count} {food1} and {food2}')
```

String Formatting, cont.

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

- Number of decimals for floats
- 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/library/string.html#format-specification-mini-language`

Input from console

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

- Prompt the user with a specified string
- Gets a line from standard input
- Returns a string (with newline stripped)

```
while True:
    x = input('Enter an integer: ')
    if x == 'exit': break
    x = int(x)
    print("{} * 2 = {}".format(x, x*2))
print("Goodbye!")
```

See also:

- *Learning Python*, Chapter 10, "A Quick Example: Interactive Loops"
- `https://docs.python.org/3/library/functions.html#input`

Table of Contents

Local and Remote Repositories

Git is a SCM (software configuration management) system. A Git repository records changes to files in a project.

- In Git, there are two kinds of repositories:
  1. The working copy: A copy of the repository on your local machine (laptop, desktop, etc.). Changes are committed (saved) to your working copy
  2. The remote: This usually lives on a web server. After saving changes to a working copy, you push (upload) them to the remote. If the remote is updated from another user, you can pull (download) those changes From your remote to the working copy.
- Note that you never commit changes directly to the remote.
- There can be (and usually are) several working copies and several remotes for the same repository. They can be on different machines, have different access permissions, etc.

The CS Department GitLab Server

- For this class, your remote repos will be hosted on the CS department's GitLab server: `https://mit.cs.uchicago.edu`
- When you arrive at the website, you login with your CNET ID.
- When you login, you should be presented with
  - Your private course repository, which will be used for submitting assignments: `https://mit.cs.uchicago.edu/mpcs51042-aut-18/cnet-id`
  - A class-wide repository: `https://mit.cs.uchicago.edu/mpcs51042-aut-18/mpcs51042-aut-18`

SSH Authentication

Creating a working copy from the remote is called cloning.
To have permissions clone your repo, you must setup SSH authentication:

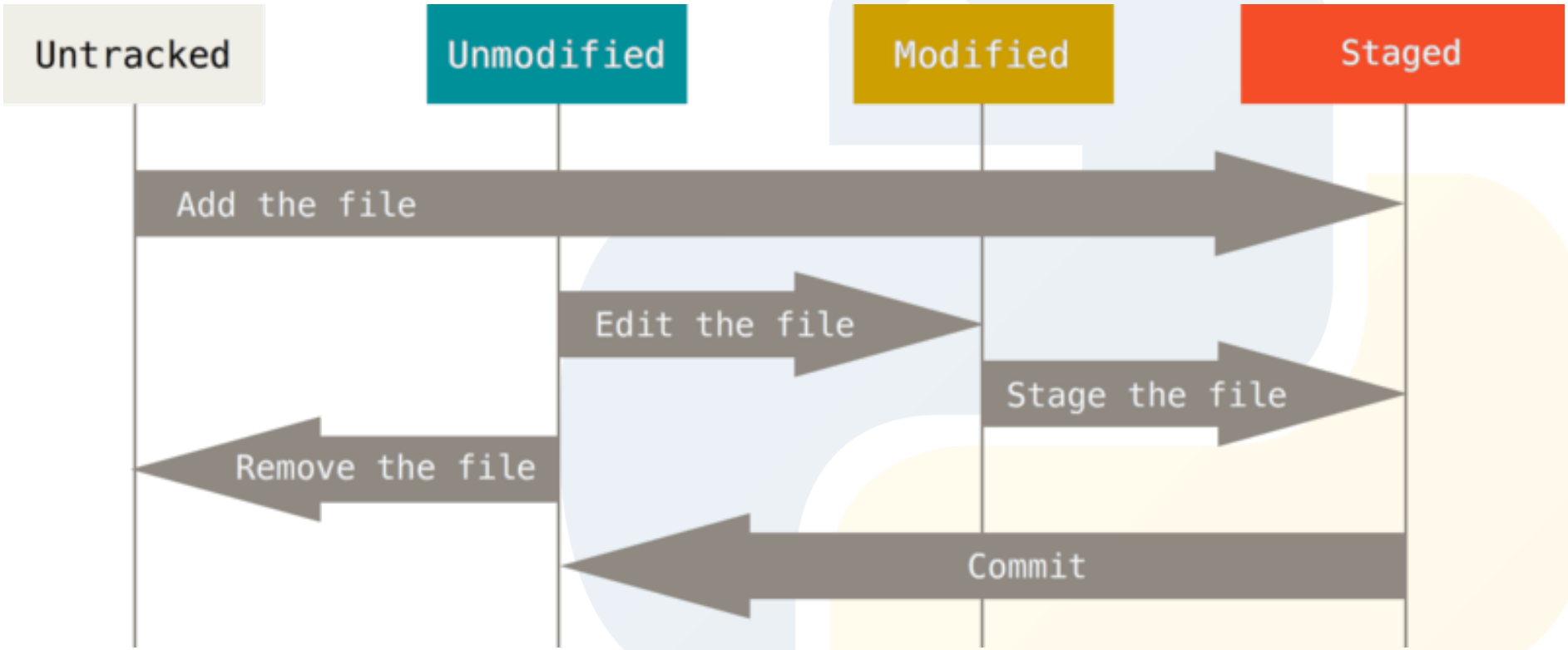
- Generate or locate your SSH public key:
  `https://mit.cs.uchicago.edu/help/ssh/README`
- Add your public key to GitLab:
  `https://mit.cs.uchicago.edu/profile/keys`

File Tracking Status

On your working copy, files are in one of the following states:

- Untracked: Not (yet) a part of the repository.
- Unmodified: Part of the repo. Current state is up-to-date with the working copy.
- Modified: Part of the repo. Changes are not up-to-date with the working copy and will not be committed.
- Staged: Part of the repo. Changed are not up-to-date and will be committed.

File Tracking Status, cont.

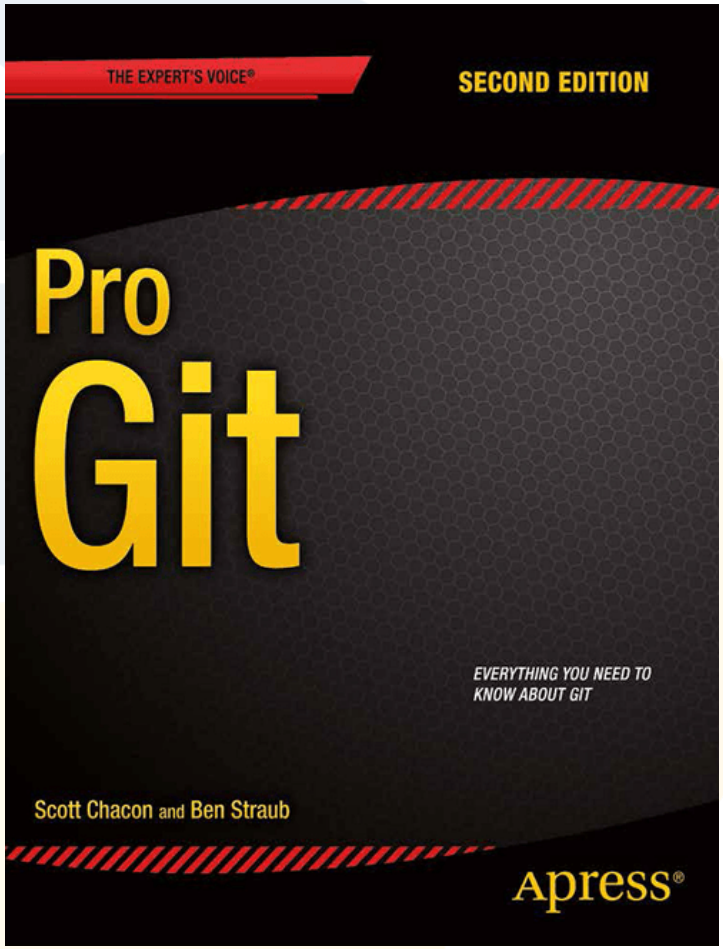


Basic Workflow

```
# 1. Clone your repo
git clone git@mit.cs.uchicago.edu:mpcs51042-aut-18/mpcs51042-a
# 2. Enter your new repo
cd mpcs51042-aut-18
# 3. Start your assignment.  Use whatever editor/IDE you want
vim demo.py
# 4. Stage your new file
git add demo.py
# 5. Maybe make some more changes
vim demo.py
# 6. Stage the latest changes
git add demo.py
#7. Commit your changes
git commit
#8. Push your changes
git push origin master
```

Git Reference

- *Pro Git* is the canonical textbook.
- Available for free at:
  `https://git-scm.com/book`



Required Reading

- *Learning Python*, 5th Ed.
  - Skim Chapters 1 - 3 as needed
  - Chapter 4 (Introducing Python Object Types), just intro and parts about string, lists, sequences.
  - Chapter 5 (Numeric Types)
  - Chapter 7 (String Fundamentals)
  - Chapter 8 (Lists and Dictionaries), just parts about lists and sequences.
  - Skim Chapters 10 - 12 as needed
  - Chapter 13 (while and for Loops), esp. for loops
- *Pro Git*
  - All of Chapter 2
  - Skim Chapters 1 and 3 as needed.