

# **Introduction to CM Introduction to Pyt**

**Week 1**

**Sanja Damjanovic**

**CPME 180A DSA in Py**

# **Week 1 Outline**

- Introduction to CMPE180A
- Getting to know everyone
- Python overview
- Tutorial

# **Sanja Damjanović, PhD**

## **CMPE 180A Instructor**

- Dipl.-ing EE, MSc EE @ University of Belgrade, Serbia
- PhD EE @ University of Twente, The Netherlands
- Expertise:
  - Deep learning, Computer vision, Digital signal processing, Image proce
- Work Experience:
  - Ruhr University Bochum (DE), Wageningen University and Research (NL)
  - applied researcher: telecommunications, agro-technology, consumer re
  - university lecturer: deep learning, computer vision, digital signal proces
- Countries:
  - USA, The Netherlands, Germany, Serbia
- Interests:
  - generative AI, multimodal LLMs, computer vision, teaching, public spe

- # **CMPE 180A Calendar**
- ## **Off-campus, meetings via Zoom**
- Lectures:
    - 5.30 pm - 9.30 pm PT on Tuesdays
    - 07/15/2025 through 09/23/2025
  - Office hours: 7pm-8pm PT on Mondays
    - Please try to join on time. If you plan to c
    - I strongly encourage attending to get valuable assignments

# CMPE 180A Course Sche

Week #	Date	Topics	Week #
1	7/15/2025	Introduction to CMPE180A  Introduction to Python, built-in datatypes, conditional expressions, comprehension syntax	6
2	7/22/2025	Control Flow, Functions, Recursion, Algorithm Analysis, Object-Oriented Programming	7
3	7/29/2025	Array-based sequences, Stacks, Queues, Deques, Linked List	8
4	8/5/2025	Trees and Tree Algorithms	9
5	8/12/2025	Midterm, Project Introduction	10
			11

# Grading

- Credit (CR) or No Credit (NC)

Grade	Grade
Credit (CR)	84-100
No credit (NC)	0-83

# **Assignments and Quizzes**

- 4 homework assignments (4x 10 points)
- Midterm (15 points)
- The comprehensive final (30 points)
- Team term project (15 points)

# Assignment Deadlines

- Homework assignment submission deadline
  - ~2 weeks
- Deadline extension
  - Possible due to individual circumstances
  - Request individual deadline extension via [sanja.damjanovic@sjsu.edu](mailto:sanja.damjanovic@sjsu.edu)
  - Max 3 extension tokens

# Resources and Reading

## [1] Problem Solving with Algorithms and Data Structures using Python

Luther College, freely available online via

<https://runestone.academy/runestone/books/published/pythonds3/index.html>

## [2] Data Structures and Algorithms in Python,

by Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, availa

<https://www.proquest.com/legacydocview/EBC/4946360?accountid=10361>

## [3] Python Cookbook, 3rd Edition, David Beazley, Brian K. Jones (Free a

<https://www.oreilly.com/>)

## [4] Think Python, 3rd edition, Allen B. Downey, freely available online via

<https://allendowney.github.io/ThinkPython/>

# Lectures

- Recap+
- New topic introduction
- In-class quiz questions for understanding check
- Tutorial
- In-class assignments
- 2 breaks: 15min @~6:30pm, 10min @~8.00pm
- Real-time online attendance recommended
- Lecture recordings available in Canvas, see PANOPTO
- Slides available before the beginning of the lecture in Canvas
- Check announcements regularly

# Introduce yourself

- Where do you live
- Your educational and work background
- Share your hobby or something interesting
- Your experience with programming and you

# **General Introduction**

# **Python**

## **Creator and timeline**

- introduced by Guido van Rossum in 1990
- a living language, it has undergone many changes
- Python 2 released in 2000, Python 3 in 2008



02/25/2019

**Guido van Rossum #341 Lex Fridman**

# Python

- Python:
  - <https://www.python.org/>
- Python Documentation
  - <https://www.python.org/doc/>
- Python Enhancement Proposals (PEPs)
  - <https://peps.python.org/>

# Python Enhancement Proposals

## the style guide for how to format Python code

[peps.python.org/pep-0008/](https://peps.python.org/pep-0008/)

[Python Enhancement Proposals](#)

[Python](#) » [PEP Index](#) » PEP 8

### Contents

- [Introduction](#)
- [A Foolish Consistency is the Hobgoblin of Little Minds](#)
- [Code Lay-out](#)
  - [Indentation](#)
  - [Tabs or Spaces?](#)
  - [Maximum Line Length](#)
  - [Should a Line Break Before or After a Binary Operator?](#)
  - [Blank Lines](#)
  - [Source File Encoding](#)
  - [Imports](#)
  - [Module Level Dunder Names](#)
- [String Quotes](#)
- [Whitespace in Expressions and Statements](#)
  - [Pet Peeves](#)
  - [Other Recommendations](#)
- [When to Use Trailing Commas](#)
- [Comments](#)
  - [Block Comments](#)
  - [Inline Comments](#)
  - [Documentation Strings](#)
- [Naming Conventions](#)
  - [Overriding Principle](#)
  - [Descriptive: Naming Styles](#)
  - [Prescriptive: Naming Conventions](#)
    - [Names to Avoid](#)
    - [ASCII Compatibility](#)
    - [Package and Module Names](#)

## PEP 8 – Style Guide

**Author:** Guido van Rossum <guido@python.org>  
          Coghlan <ncoghlan@gmail.com>

**Status:** [Active](#)

**Type:** [Process](#)

**Created:** 05-Jul-2001

**Post-History:** 05-Jul-2001, 01-Aug-2001

### ► Table of Contents

## [Introduction](#)

This document gives coding conventions for the Python distribution. Please see the complete [the C implementation of Python](#).

This document and [PEP 257](#) (Docstring Convention) are the basis for the Python style guide, with some additions from Barry's original style guide.

This style guide evolves over time as additions and corrections are made, and becomes obsolete by changes in the language itself.

Many projects have their own coding style guides. These guides take precedence for that project.

# **Programming languages**

## **In general**

- Low level vs high level
  - program instructions on the machine level or
- General vs targeted to application domain
  - widely applicable or are fine-tuned to a domain
- Interpreted versus compiled
  - refers to whether the sequence of instructions in source code, is executed directly (by an interpreter) or converted (by a compiler) into a sequence of

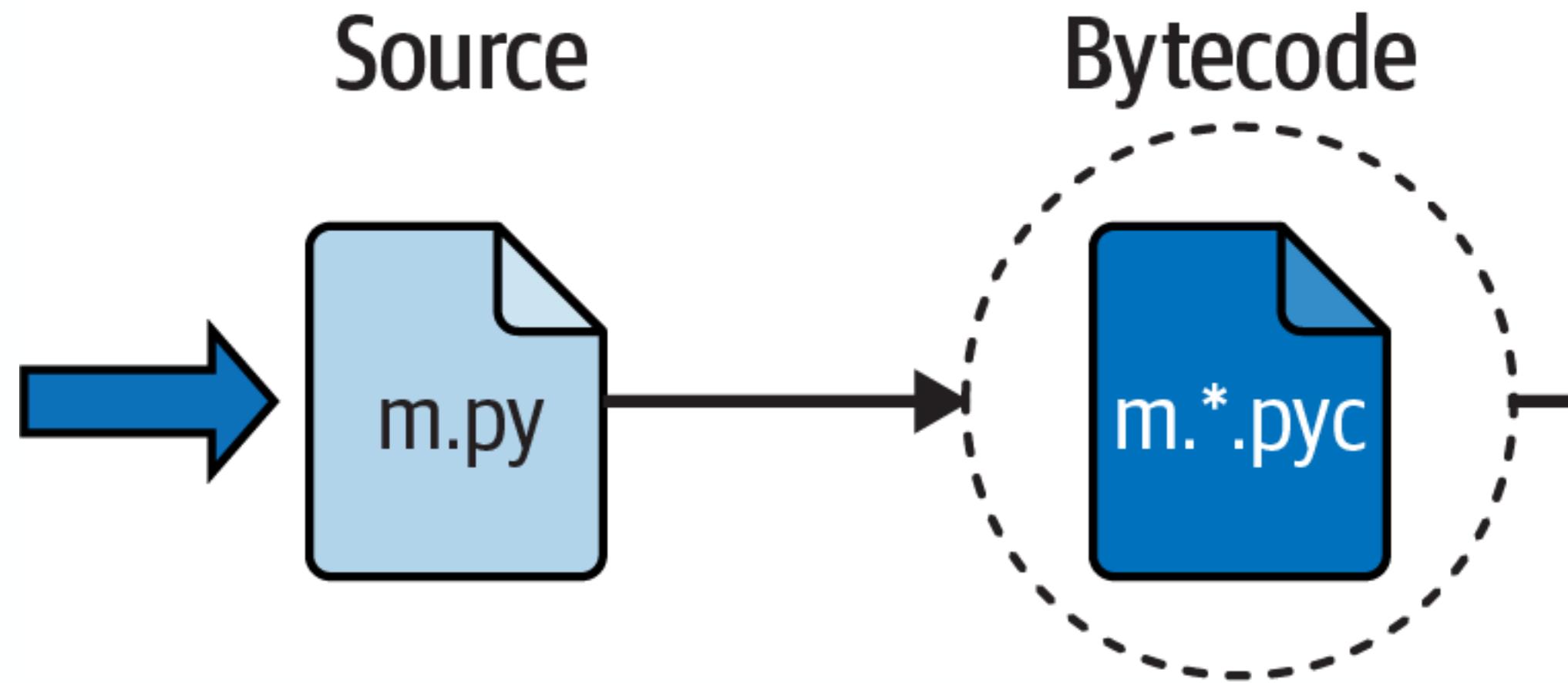
# Python

- Low level vs **high level**
- **General** vs targeted to application domain
- **Interpreted** vs compiled
- Python is a general-purpose programming language that can be used effectively to build almost any kind of program, from simple scripts to complex web applications that access to the computer's hardware.

# Python

- Easy to get started with
- Readable and concise syntax
- Rich standard library and built-in data structures
- Interactive, interpreted, cross-disciplinary
- Rapid prototyping
- Application areas: Data science, machine learning, Cloud/Admin, web development, ...

# Python's traditional execution



- Bytecode: a lower-level, platform-independent representation of Python code, ‘compiled’ to or loaded from .pyc files
- PVM: Python Virtual Machine executes the code

# **Python**

- Rapid prototyping
- Numeric and scientific programming
- Object oriented language
- Free & open
- Portable

# Python Disadvantage

- Python's *execution speed* may not always be faster than lower-level languages such as fully compiled languages like C/C++.

# Python Installation

- Anaconda ([anaconda.org](https://anaconda.org)):
  - Pre-compiled meta-distribution
  - Includes many scientific libraries
  - Can create environments
- [python.org](https://python.org)
  - Install libraries

# Python 3

**<https://www.python.org/>**

- Install and run on your computer Python (+IDE):
  - Command line <https://www.python.org/downloads/> (pre-installed)
  - Anaconda <https://www.anaconda.com/>
  - JupyterLab (notebook) <https://jupyter.org/install>
  - VS Code (<https://code.visualstudio.com/>)
  - Google Colab <https://colab.google/>
- File extensions: \*.py, \*.ipynb
- A module is a .py file containing Python definitions and statements
- Modules are imported using ‘import’ statement

# Python program

**\*.py**

- A Python program (script, source code):
  - a sequence of definitions and commands (statements)
  - a plain text file with suffix .py
- executed by the Python interpreter in something like an environment
- Objects are of certain type e.g. int/float
- Objects and operators can be combined to form expressions
  - Expression evaluates to an object of some type

# Python libraries/packages

- Import Python libraries/packages/modules using:
- Module:
  - \*.py file
- Packages:
  - numPy: Scientific computing
  - pandas: open source data analysis and manipu
  - scikit-learn: machine learning
  - SciPy, matplotlib, ...

**Pytho**

# Python

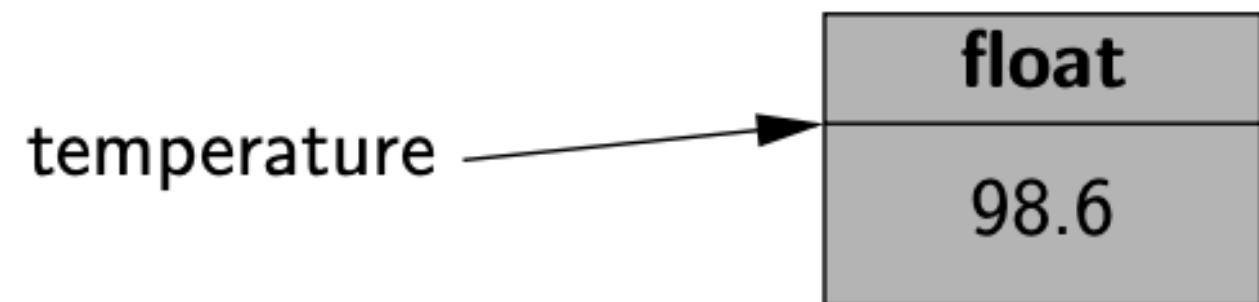
- Syntax relies on the use of white spaces
  - An individual statement ends with a new line
  - Comments are ignored by the Python interpreter

# Name (identifier)

## Identifiers, Objects, and the Assignment Statement

- Assignment statement:

- temperature = 98.6
- bindings of names/identifiers to objects



- Names/identifiers: case-sensitive, cannot begin with a number
- The identifier **temperature** is associated with an instance of a float object

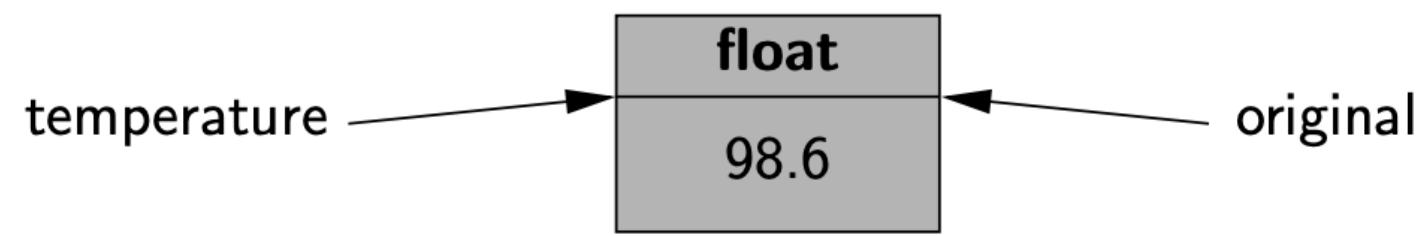
# Identifiers in Python

- ❑ 33 specially reserved words that are identifiers:

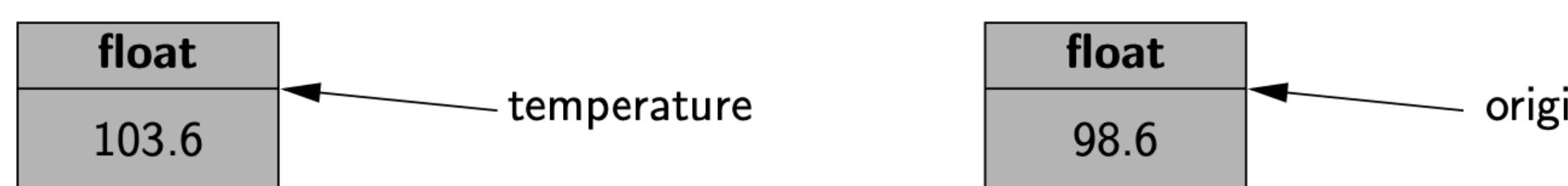
Reserved Words							
False	as	continue	else	from	is	lambda	None
None	assert	def	except	global	not	nonlocal	True
True	break	del	finally	if	or	raise	and
and	class	elif	for	import	pass	return	as
class	delattr	exec	lambda	raise	try	with	del
delattr	dict	filter	len	range	try	yield	dict
dict	dir	float	list	reversed	zip	yield	dir
dir	enumerate	float	map	set	abs	all	any
enumerate	float	map	set	slice	abs	all	any
float	for	max	slice	sorted	all	any	bool
for	format	min	sorted	staticmethod	any	bool	bool
format	function	next	super	staticmethod	bool	bool	bytes
function	getattr	object	vars	super	bytes	bytes	bytes
getattr	globals	property	yield	super	bytes	bytes	bytes
globals	hasattr	range	yield	super	bytes	bytes	bytes
hasattr	help	round	yield	super	bytes	bytes	bytes
help	int	str	yield	super	bytes	bytes	bytes
int	issubclass	sum	yield	super	bytes	bytes	bytes
issubclass	iter	tuple	yield	super	bytes	bytes	bytes
iter	len	type	yield	super	bytes	bytes	bytes
len	list	vars	yield	super	bytes	bytes	bytes
list	locals	zip	yield	super	bytes	bytes	bytes
locals	next	zip	yield	super	bytes	bytes	bytes
next	object	zip	yield	super	bytes	bytes	bytes
object	property	zip	yield	super	bytes	bytes	bytes
property	range	zip	yield	super	bytes	bytes	bytes
range	reversed	zip	yield	super	bytes	bytes	bytes
reversed	slice	zip	yield	super	bytes	bytes	bytes
slice	super	zip	yield	super	bytes	bytes	bytes
super	tuple	zip	yield	super	bytes	bytes	bytes
tuple	vars	zip	yield	super	bytes	bytes	bytes
vars	yield	zip	yield	super	bytes	bytes	bytes
yield	yield	zip	yield	super	bytes	bytes	bytes

# Aliases and Reassignment

- Identifiers *temperature* and *original* are aliases

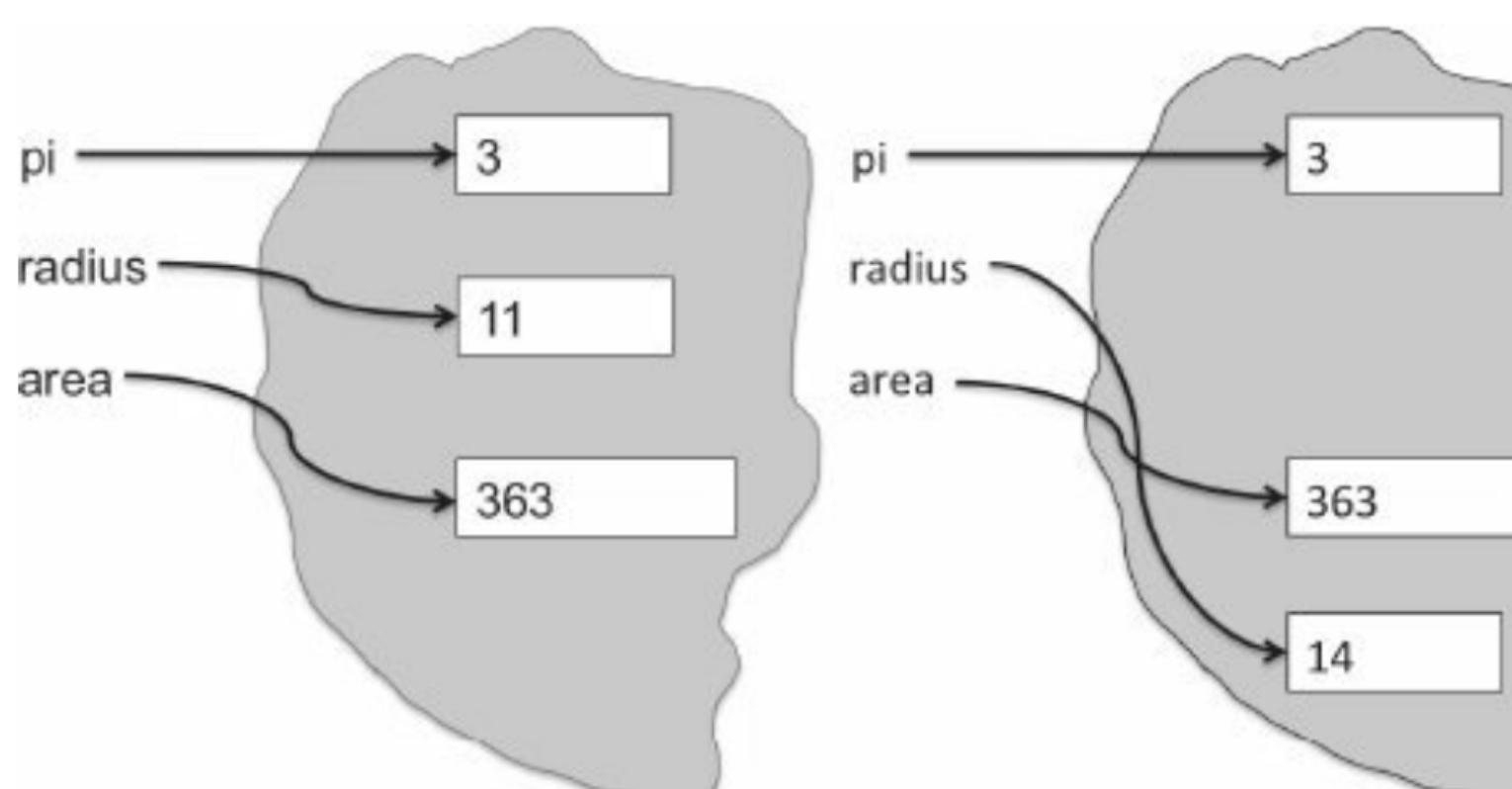


- $temperature = temperature + 5.0$
- The *temperature* identifier has been assigned a new value, but it continues to refer to the previously existing memory location.



# Dynamically typed language

- No advance declaration associating an identifier with a type
- An identifier can be associated with any type and can be reassigned to another object of the same (or different) type



# Python's built-in classes

- ❑ the **int** class for integers
- ❑ the **float** class for floating-point numbers
- ❑ the **str** class for characters and strings

→ Tutorial

# An Example Program

```
print('Welcome to the GPA calculator.')
print('Please enter all your letter grades,
print('Enter a blank line to designate the end of grades')
# map from letter grade to point value
points = {'A+':4.0, 'A':4.0, 'A-':3.67, 'B+':3.33,
          'B':3.0, 'B-':2.67, 'C+':2.33, 'C':2.0, 'C-':1.67, 'D+':1.33, 'D':1.0, 'D-':0.67}
num_courses = 0
total_points = 0
done = False
while not done:
    grade = input( )                                # read a line
    if grade == '':
        done = True
    elif grade not in points:                      # unknown grade
        print("Unknown grade '{0}' being ignored".format(grade))
    else:
        num_courses += 1
        total_points += points[grade]
if num_courses > 0:                                # avoid division by zero
    print('Your GPA is {:.3}'.format(total_points / num_courses))
```

# Objects

- The process of creating a new instance **instantiation**.
- To instantiate an object we usually invoke the class:  
 $w = \text{Widget}()$ 
  - This is assuming that the constructor does not require parameters.
- If the constructor does require parameters such as  
 $w = \text{Widget}(a, b, c)$
- Many of Python's built-in classes also support the creation of new instances. For example, the command  
 $\text{temperature} = 98.6$  results in the creation of a new instance.

# Calling Methods

- ❑ Python supports functions a syntax **sorted(data)**, in which case data is passed as an argument to the function.
- ❑ Python's classes may also define methods (also known as member functions) that belong to a specific instance of a class using the `self` parameter.
- ❑ For example, Python's list class has a `sort` method that can be invoked with a syntax like `listObject.sort()`.
  - This particular method rearranges the elements in a list so that they are sorted.

# Built-In Classes

Class	Description
<code>bool</code>	Boolean value
<code>int</code>	integer (arbitrary magnitude)
<code>float</code>	floating-point number
<code>list</code>	mutable sequence of objects
<code>tuple</code>	immutable sequence of objects
<code>str</code>	character string
<code>set</code>	unordered set of distinct objects
<code>frozenset</code>	immutable form of <code>set</code> class
<code>dict</code>	associative mapping (aka dictionary)

- ❑ A class is immutable if each object has a fixed value upon instantiation and cannot subsequently be changed. For example, the **float** class is immutable.

# The `bool` Class

- ❑ The **bool** class is used for logical values. It contains the only two instances of that class, represented by the literals:  
    True     and     False
- ❑ The default constructor, `bool( )`, returns False.
- ❑ Python allows the creation of a Boolean value from any nonboolean type using the syntax `bool( value )`. The interpretation depends upon the type of the parameter.
  - Numbers evaluate to False if zero, and True otherwise.
  - Sequences and other container types evaluate to False if empty and True if they contain at least one item.

# The int Class

- The **int** class is designed to represent arbitrary magnitude.
  - Python automatically chooses the integer type based upon the magnitude of the value.
- The integer constructor, `int( )`, returns an integer.
  - This constructor can also construct an integer upon an existing value of another type.
    - For example, if `f` represents a floating-point number, `int(f)` produces the truncated value of `f`. For example, `int(3.9)` produces the value 3, while `int(-3.9)` produces -3.
    - The constructor can also be used to produce an integer. For example, the expression `int('137')` produces the integer value 137.

# The float Class

- The **float** class is the floating-point class
  - The floating-point equivalent of an integer is expressed directly as 2.0.
  - One other form of literal for floating-point notation. For example, the literal 6.022e23 represents the mathematical value  $6.022 \times 10^{23}$ .
- The constructor `float( )` returns a floating-point number.
- When given a parameter, the constructor `float( )` returns the equivalent floating-point value.
  - `float(2)` returns the floating-point value 2.0.
  - `float('3.14')` returns 3.14.

# The list Class

- A **list** instance stores a sequence of references (or pointers) to objects.
- Elements of a list may be arbitrary objects (not just other objects).
- Lists are array-based sequences and elements indexed from 0 to n–1 inclusive.
- Lists have the ability to dynamically expand their capacities as needed.
- Python uses the characters [ ] as delimiters for lists.
  - [ ] is an empty list.
  - ['red', 'green', 'blue'] is a list containing three strings.
- The list( ) constructor produces an empty list.
- The list constructor will accept any iterable object.
  - list('hello') produces a list of individual characters.

# The tuple Class

- ❑ The **tuple** class provides an (unchangeable) version of a list. It allows instances to have an representation that may be more compact than that of a list. Parentheses are used.
  - The empty tuple is ()
- ❑ To express a tuple of length 1, a comma must be placed after the single element within the parentheses.
  - For example, (17,) is a one-element tuple.

# The str Class

- ❑ String literals can be enclosed in single quotes, as in 'hello', or in "hello".
- ❑ A string can also begin with three single or double quotes if it contains newlines in it.

```
print("""Welcome to the GPA  
calculator.  
Please enter all your letter grades  
on separate lines, ending with  
Enter a blank line to designate  
the end of the list.)
```

# The set Class

- ❑ Python's **set** class represents a set of elements, without duplicates, and no order to those elements.
- ❑ Only instances of immutable types can be added to a Python set. Therefore, objects such as point numbers, and character strings cannot be elements of a set.
  - The frozenset class is an immutable frozen set.
- ❑ Python uses curly braces { and } to represent sets.
  - For example, as {17} or {'red', 'green'}
  - The exception to this rule is that {} creates an empty set. Instead, the constructor set( ) returns an empty set.

# The dict Class

- Python's **dict** class represents from a set of distinct keys to a
- Python implements a dict using approach to that of a set, but w associated values.
  - The literal form { } produces an em
- A nonempty dictionary is expressed separated series of **key:value** dictionary {'ga' : 'Irish', 'de' : 'O 'Irish' and 'de' to 'German'.
- Alternatively, the constructor a key-value pairs as a parameter, pairs = [('ga', 'Irish'), ('de', 'Ge

# Expressions and Operators

- ❑ Existing values can be combined into new expressions using specific keywords known as operators
- ❑ The semantics of an operator depend upon the type of its operands
- ❑ For example, when a and b are numbers, the syntax **a + b** indicates addition, while if a and b are strings, the same operator **+** indicates concatenation

# Logical Operators

- ❑ Python supports the following operators for Boolean values:

<b>not</b>	unary negation operator
<b>and</b>	conditional AND operator
<b>or</b>	conditional OR operator

- ❑ The and and or operators are short-circuiting in that they do not evaluate the second operand if the result can be determined based on the value of the first operand.

# Equality Operators

- ❑ Python supports the following operators to test two notions of equality:

<b>is</b>	same identity
<b>is not</b>	different identity
<b>==</b>	equivalent
<b>!=</b>	not equivalent
- ❑ The expression, `a is b`, returns True, precisely when `a` and `b` are aliases for the same object.
- ❑ The expression `a == b`, tests the general notion of equivalence.

# Comparison Operators

- ❑ Data types may define  
via the following operators

<	less than
$\leq$	less than or equal
>	greater than
$\geq$	greater than or equal

- ❑ These operators have equivalent meanings for numeric types, and for strings.

# Arithmetic Operators

- Python supports the following arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	true division
//	integer division
%	the modulo operator
- For addition, subtraction, and multiplication, if both operands have type int, then the result is also int. If either operand has type float, then the result is float.
- True division is always of type float. Integer division is always int (with the result truncated to an integer).

# Bitwise Operators

- ❑ Python provides the following operators for integers:

<code>~</code>	bitwise complement (pronounced "not")
<code>&amp;</code>	bitwise and
<code> </code>	bitwise or
<code>^</code>	bitwise exclusive-or
<code>&lt;&lt;</code>	shift bits left, filling in zeros
<code>&gt;&gt;</code>	shift bits right, filling in zeros

# Sequence Operator

- Each of Python's built-in sequence types (str, tuple, and list) supports similar operator syntaxes:

<code>s[j]</code>	element at index $j$
<code>s[start:stop]</code>	slice including indices $\text{start}$ through $\text{stop} - 1$
<code>s[start:stop:step]</code>	slice including indices $\text{start}, \text{start} + \text{step}, \text{start} + 2*\text{step}, \dots$
<code>s + t</code>	concatenation of sequences $s$ and $t$
<code>k * s</code>	shorthand for $s + s + \dots + s$ ( $k$ times)
<code>val in s</code>	containment check
<code>val not in s</code>	non-containment check

# Sequence Comparison

- ❑ Sequences define comparison based on lexicographic order, performed by element comparison until a difference is found.
  - For example,  $[5, 6, 9] < [5, 7, 8]$  because  $6 < 7$  at index 1.

<code>s == t</code>	equivalent (element by element)
<code>s != t</code>	not equivalent
<code>s &lt; t</code>	lexicographically less than
<code>s &lt;= t</code>	lexicographically less than or equal to
<code>s &gt; t</code>	lexicographically greater than
<code>s &gt;= t</code>	lexicographically greater than or equal to

# Operators for Sets

- ❑ Sets and frozensets support the following operators:

<code>key in s</code>	containment check
<code>key not in s</code>	non-containment check
<code>s1 == s2</code>	<code>s1</code> is equivalent to <code>s2</code>
<code>s1 != s2</code>	<code>s1</code> is not equivalent to <code>s2</code>
<code>s1 &lt;= s2</code>	<code>s1</code> is subset of <code>s2</code>
<code>s1 &lt; s2</code>	<code>s1</code> is proper subset of <code>s2</code>
<code>s1 &gt;= s2</code>	<code>s1</code> is superset of <code>s2</code>
<code>s1 &gt; s2</code>	<code>s1</code> is proper superset of <code>s2</code>
<code>s1   s2</code>	the union of <code>s1</code> and <code>s2</code>
<code>s1 &amp; s2</code>	the intersection of <code>s1</code> and <code>s2</code>
<code>s1 - s2</code>	the set of elements in <code>s1</code> but not in <code>s2</code>
<code>s1 ^ s2</code>	the set of elements in either <code>s1</code> or <code>s2</code> , but not both

# Operators for Dictionaries

- ❑ The supported operators for type dict are as follows

<code>d[key]</code>	value associated with key
<code>d[key] = value</code>	set (or reset) the value for key
<code>del d[key]</code>	remove key and its associated value
<code>key in d</code>	containment check
<code>key not in d</code>	non-containment check
<code>d1 == d2</code>	<code>d1</code> is equivalent to <code>d2</code>
<code>d1 != d2</code>	<code>d1</code> is not equivalent to <code>d2</code>

# Operator Precedence

Operator Precedence

	Type	Symbols
1	member access	expr.member
2	function/method calls container subscripts/slices	expr(...) expr[...]
3	exponentiation	**
4	unary operators	+expr, -expr
5	multiplication, division	*, /, //, %
6	addition, subtraction	+, -
7	bitwise shifting	<<, >>
8	bitwise-and	&
9	bitwise-xor	^
10	bitwise-or	
11	comparisons containment	is, is not, = in, not in
12	logical-not	not expr
13	logical-and	and
14	logical-or	or
15	conditional	val1 if cond else val2
16	assignments	=, +=, -=, *=, /=, //=, %=

# Program Structure

- ❑ Common to all control structures is used to delimit the beginning acts as a body for a control structure.
- ❑ If the body can be stated as a single statement, it can technically place to the right of the colon.
- ❑ However, a body is more typically an indented block starting on the line following the colon.
- ❑ Python relies on the indentation extent of that block of code, or a code within.

# Conditionals

```
if first_condition  
    first_body  
elif second_condition  
    second_body  
elif third_condition  
    third_body  
else:  
    fourth_body
```

# Loops

- ❑ While loop:

**while condition:**  
*body*

- ❑ For loop:

**for element in iterable:**

*body*

# body may r

- ❑ Indexed For loop:

big  
for  
if

# Break and Continue

- Python supports a **break** statement to immediately terminate a while loop executed within its body.

```
found = False
for item in data:
    if item == target:
        found = True
        break
```

- Python also supports a **continue** statement which causes the current iteration of a loop to skip over the rest of the loop body but with subsequent passes continuing as expected.

# Functions

- Functions are defined using the keyword `def`.

```
def count(data, target):
    n = 0
    for item in data:
        if item == target:
            n += 1
    return n
```

- This establishes a new identifier as a function (count, in this example), and it establishes the parameters that it expects, which is called the **signature**.
- The **return** statement returns the value and terminates its processing.

# Information Passing

- ❑ Parameter passing in Python follows the standard assignment statement
- ❑ For example

```
prizes = count(grades)
```

is the same as

```
data = grades  
target = 10
```

and results in



# Simple Output

- ❑ The built-in function, **print**, generate standard output
- ❑ In its simplest form, it prints a sequence of arguments, separated by spaces, and followed by a trailing new-line character.
- ❑ For example, the command `print('maroon')` outputs the string 'maroon'.
- ❑ A nonstring argument `x` will be converted to a string by `str(x)`.

# Simple Input

- ❑ The primary means for acquiring input from the user console is a built-in function named `input`.
- ❑ This function displays a prompt message, takes an optional parameter, and then waits for the user to enter some sequence of characters and press the return key.
- ❑ The return value of the function is a string containing all the characters that were entered since the last time the return key was pressed.
  - Such a string can immediately be converted to an integer using the `int` function.

```
year = int(input('In what year were you born?'))
```

# A Simple Program

- Here is a simple program with some input and output

```
age = int(input('Enter your age in years: '))
max_heart_rate = 206.9 - (0.67 * age)
target = 0.65 * max_heart_rate
print('Your target fat-burning heart rate is', target)
```

# Week 1 Reading List

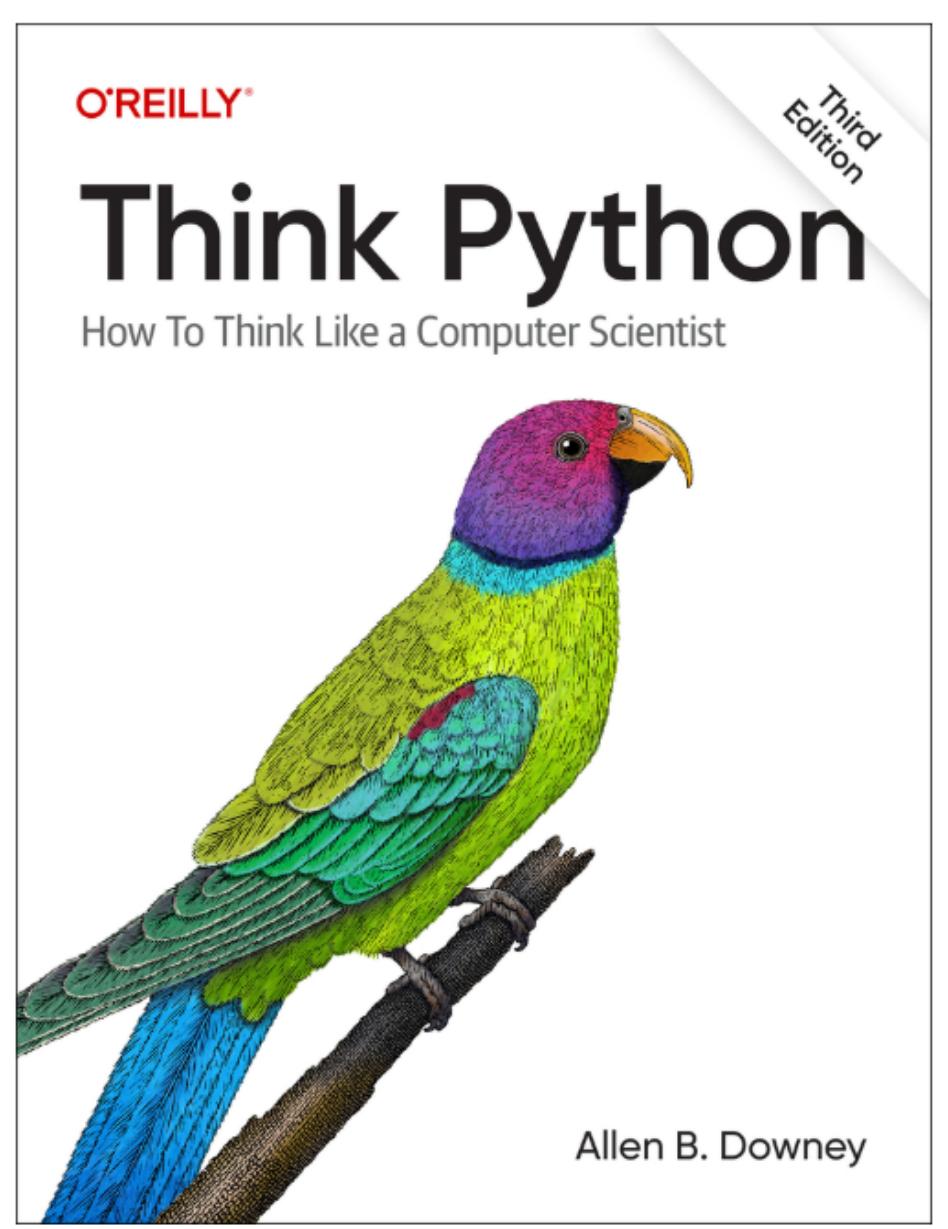
- [1] Chapter 1
- [2] Chapter 1

[1] Problem Solving with Algorithms and Data Structures  
Miller and David Ranum, Luther College, freely available online

[2] Data Structures and Algorithms in Python,  
by Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser  
SJSU library via [link \[2\]](#)

# Other resources

- <https://allendowney.github.io/ThinkPython/>



# Week 2 Reading List

- [1] Chapter 2, 3, 4
- [2] Chapter 2, 4

[1] Problem Solving with Algorithms and Data Structures  
Brad Miller and David Ranum, Luther College,  
via [link \[1\]](#)

[2] Data Structures and Algorithms in Python,  
by Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser  
via SJSU library via [link \[2\]](#)