Python for Programmers

Min Wu

mwu63@ucsc.edu

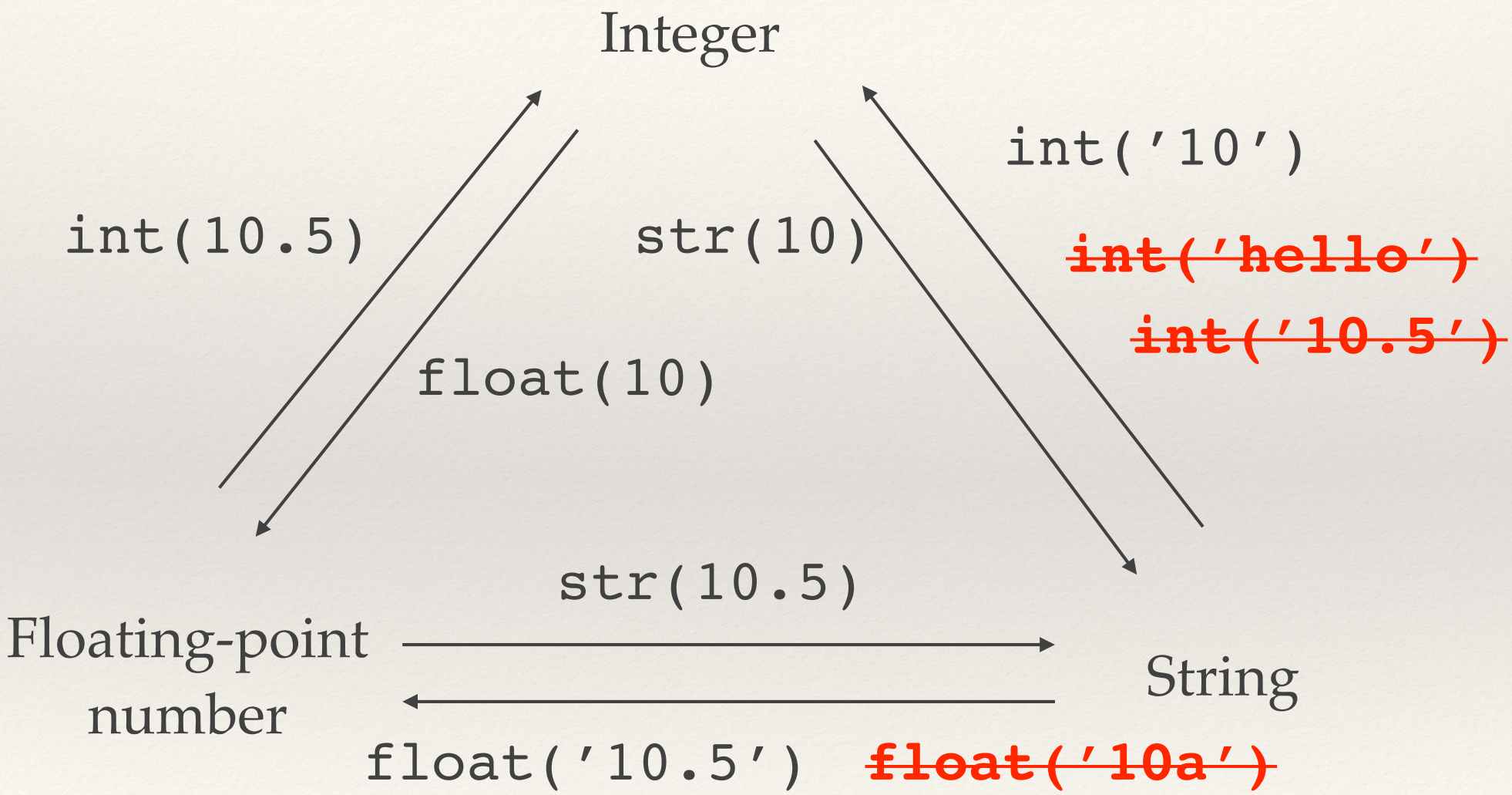
Preface

- Do I need to learn programming considering the rise of LLMs?

Outline

- Warmup: write Python code that can be understandable
- Basic data structures:
  - List
  - Dictionary
  - Tuple
  - Set
- Basic algorithms
  - Recursion
  - Flood fill
  - Dynamic programming
- Python OOP
  - Special methods, module and package
  - Inheritance, property, exceptions
  - Attribute access, descriptors
- Python thinking
  - Iterator
  - Generator and coroutine
  - Decorators
  - Context managers
- Popular libraries

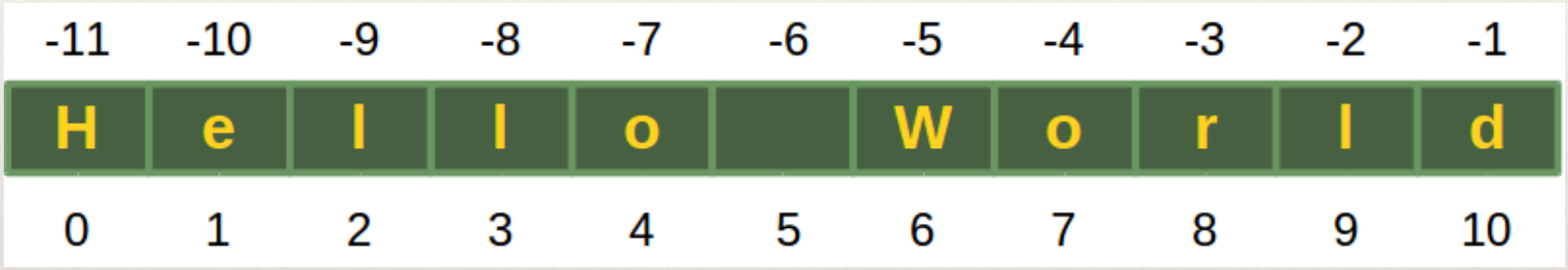
Data Type Conversion



Strings

- `len()`: number of characters in a string
- Get individual characters by index
- Get individual characters using for loop
  - ```
    for letter in sentence:
        print(letter)
    ```
    ```

String Index



String Slices

- Slice: a segment of a string
- `str[n:m]` returns part of the string `str` from the `n`-th character to the `m`-th character, including the `n`-th but excluding the `m`-th character.
-
```
>>> fruit[1:4]
'ana'
>>> fruit[3:4]
'a'
>>> fruit[3:3]
''
>>> fruit[3:2]
''
```

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
>>> fruit[:]
'banana'
```

Search for Letter/Substring

- ```
  >>> 'hello'.index('e')
  1
  >>> 'hello world'.find('world')
  6
  ```

- ```
  >>> 'hello'.find('a')
  -1
  >>> 'hello'.index('a')
  Traceback (most recent call last):
  …
  ValueError: substring not found
  ```

File Basics

- Permanent storage (files and databases)
- Read text file
  - ```
    fin = open('filename.txt')
    line = fin.readline()
    ```
  - ```
    # remove whitespace characters from
    beginning or end
    word = line.strip()
    ```

File Basics

- Write to a text file
  - `fout = open('filename.txt', 'w')`
- If the file exists, open it in the write mode, **clear the old data, and start fresh**;
- If the file does not exist, create a new file with the file name.
- On the other hand, trying to open a non-existent file in the read mode throws a `FileNotFoundError`.

File Basics

- Write to a text file using the `write()` method
- The argument has to be a string.
  - `fout.write('hello')`
- If you want a new line, you have to include the new line character `'\n'` in the string.
  - `fout.write('hello\n')`
- When you have done writing, you have to close the file.
  - `fout.close()`

Class 2: Basic Data Structures

Lists

- A list is a sequence of values
  - `[10, 20, 30, 40]`
  - `['apple', 'orange', 'grape']`
  - `['hello', 2.4, 10, [3, 7]]`
- The elements of a list do not have to be the same type.
- A list that contains no elements is an empty list `[]`.

List – Element Access

- Access the elements by index (both the positive and the negative indexes)
  - ```
    >>> num = [10, 20, 30, 40]
    >>> num
    [10, 20, 30, 40]
    >>> num[1]
    20
    >>> num[2] = 50
    >>> num
    [10, 20, 50, 40]
    >>> 50 in num
    True
    >>> 100 in num
    False
    ```
    ```

Sorting

- `sort` is mutator.
- ```
  arr = [47, 26, 7, 10]
  arr.sort()
  ```
- ```
  arr = ['hello', 'name', 'age', 'math']
  ```
- ```
  arr = [‘hello’, 47]
  arr.sort()
  …
  TypeError: unorderable types: int() <
  str()
  ```
  ```

Sorting

- `sorted` always returns a new sorted list from the given **iterable**.
- `sorted` is not a mutator.
- ```
  arr = [47, 26, 7, 10]
  b = sorted(arr)
  ```
  ```

List Operations

- Concatenation
  - ```
    >>> a = [1, 2, 3]
    >>> b = [4, 5, 6]
    >>> c = a + b
    >>> c
    [1, 2, 3, 4, 5, 6]
    ```
- Repetition
  - ```
    >>> d = a * 4
    >>> d
    [1, 2, 3, 1, 2, 3,
    1, 2, 3, 1, 2, 3]
    ```
- Slice
  - ```
    >>> d[1:3]
    [2, 3]
    >>> d[:4]
    [1, 2, 3, 1]
    >>> d[3:]
    [1, 2, 3, 1, 2, 3,
    1, 2, 3]
    >>> d[-3:]
    [1, 2, 3]
    ```
    ```

More About Slice

- **List is mutable.**
- ```
  >>> d
  [1, 2, 3, 1, 2, 3, 1, 2, 3,
  1, 2, 3]
  ```
- Deletion
  ```
  >>> d[1:4]=[]
  >>> d
  [1, 2, 3, 1, 2, 3, 1, 2, 3]
  ```
- Insertion
  ```
  >>> d[5:5]=[10, 20, 30]
  >>> d
  [1, 2, 3, 1, 2, 10, 20, 30,
  3, 1, 2, 3]
  ```
- Replacement
  ```
  >>> d[3:8]=[100, 200]
  >>> d
  [1, 2, 3, 100, 200, 3, 1,
  2, 3]
  ```
- ```
  >>> d[2:4] = 1
  …
  TypeError: can only assign
  an iterable
  ```

List Methods

- `append`: adds **a** new **element** to the end of a list
  - ```
    t = ['a', 'b', 'c']
    t.append('d')
    ```
  - What about `t.append(['e', 'f'])`?
- `extend`: takes a list as **an** argument and appends all of its element
  - ```
    t = ['a', 'b', 'c']
    t.extend(['d', 'e'])
    ```

Delete Elements – Pop (by Index)

```
>>> arr = [1, 2, 3, 'a', 4, 'e']
>>> x = arr.pop()      # by default, delete and return the last element
>>> x
'e'
>>> arr
[1, 2, 3, 'a', 4]
>>> y = arr.pop(2)     # delete and return certain element using index
>>> y
3
>>> arr
[1, 2, 'a', 4]
```

Delete Element – Remove (by Value)

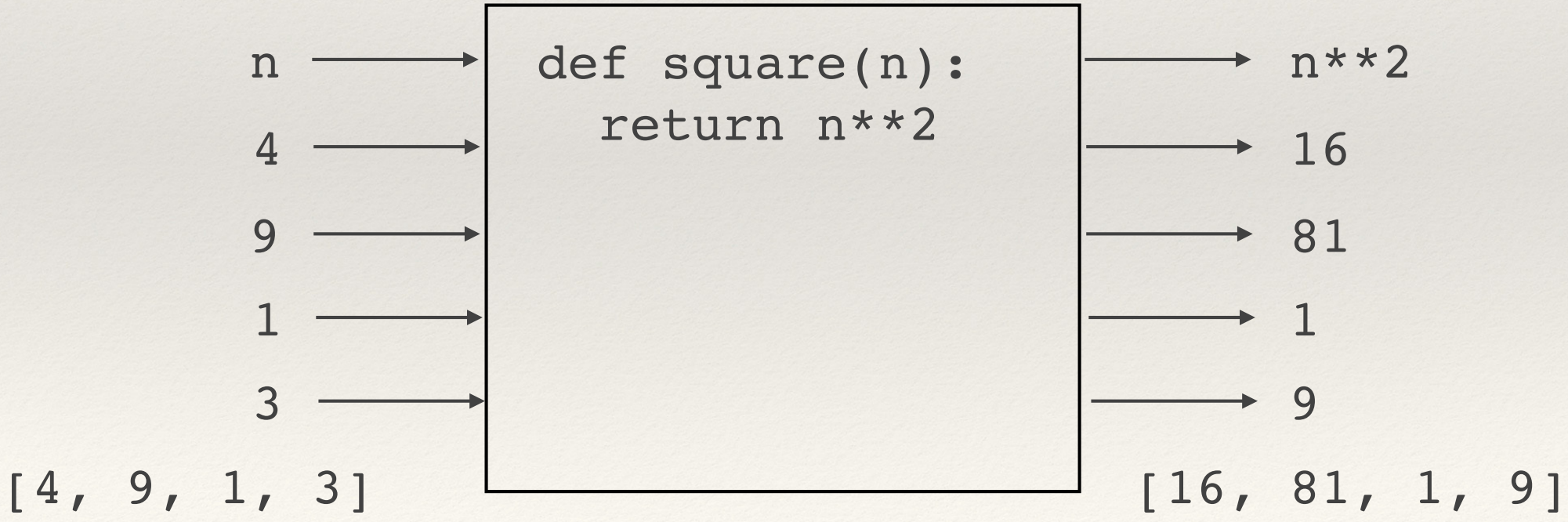
```
>>> arr
[1, 2, 'a', 4]
>>> arr.remove(4) # remove returns None
>>> arr
[1, 2, 'a']
>>> arr.remove(1)
>>> arr
[2, 'a']
```

Delete Elements – Del (multiple deletion)

```
>>> arr = [1, 2, 3, 'a', 4, 'e']
>>> arr
[1, 2, 3, 'a', 4, 'e']
>>> del arr[1:3] # delete using slice
>>> arr
[1, 'a', 4, 'e']
>>> del arr[1] # delete single by index
>>> arr
[1, 4, 'e']
```

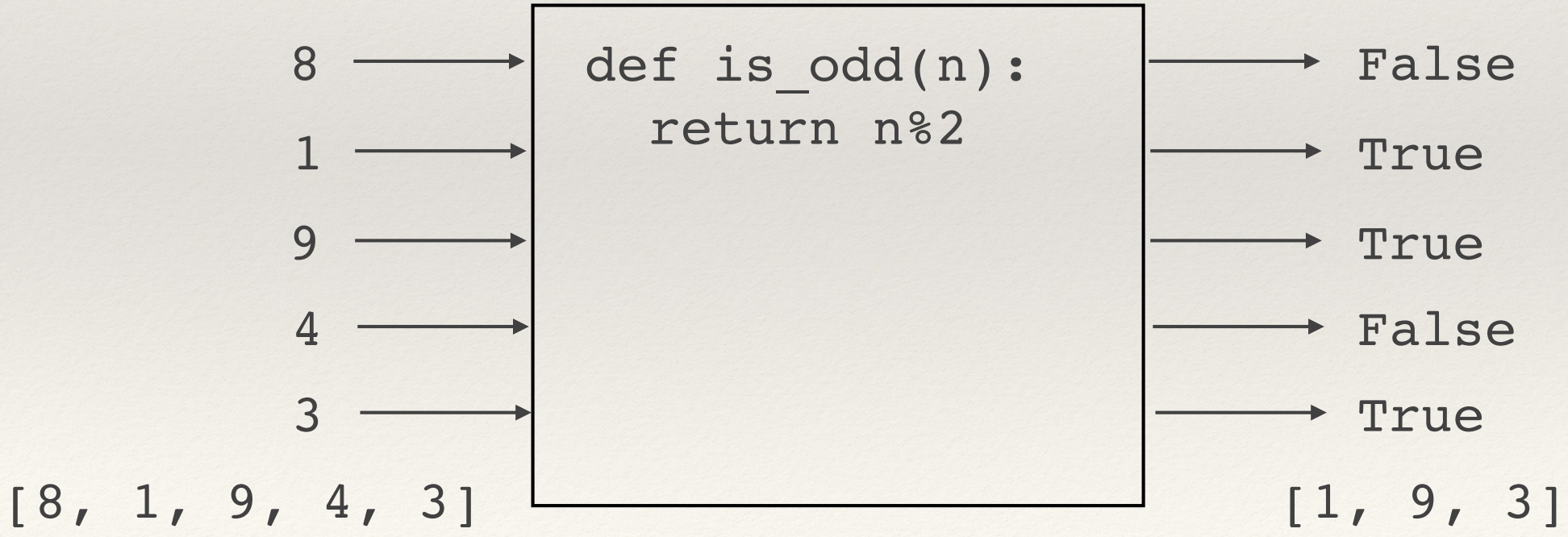
Map

- `map()` applies the given function to all elements in a list. The result of the function applied to each element forms a new list.



Filter

- `filter()` applies the given function to each element in a list. The returned new list includes all elements whose result from the function call is `True`.



Lambda

- Lambda represents anonymous function, which is a simple function that can be used by `map()` or `filter()`.
-
```
>>> lambda n: n**2
<function <lambda> at 0x101a08950>
>>> _
<function <lambda> at 0x101a08950>
>>> _(9)
81
```

Dictionary

- Key-value pairs
- Keys must be unique.

`in` and `not in` operators

- Test for membership
- list, string, dictionary
  - `>>> 'a' in 'apple'`
    `True`
  - `>>> 4 in [1, 2, 3, 4, 5]`
    `True`
  - `>>> 'a' in {'a': 1, 'b': 3}`
    `True`
- The `in` operator tests whether the dictionary has a given key.

Dictionary is orderless

- In Python, the key-value pairs stored in the dictionary are orderless.

Tuple

- A tuple is a sequence of values, very similar to list.
- But tuples are immutable.
- ```
  >>> t = (1, 2, 3, 'a', 'b', 'c')
  >>> t
  (1, 2, 3, 'a', 'b', 'c')
  >>> type(t)
  <class ‘tuple'>
  ```
- ```
  >>> l = [1, 2, 3, 'a', 'b']
  >>> l
  [1, 2, 3, 'a', 'b']
  >>> type(l)
  <class 'list'>
  ```

Tuple as Return Value

- ```
  def sum_ave(numbers):
      return sum(numbers), sum(numbers)/
  len(numbers)
  ```
- ```
  >>> sum_ave([1, 2, 3, 4, 5, 6])
  (21, 3.5)
  ```
- ```
  def min_max(numbers):
      return min(numbers), max(numbers)
  ```
- ```
  >>> min_max([3, 2, 8, 1, 4, 9])
  (1, 9)
  ```

Tuple as Function Parameter

- A parameter name that begins with * **gathers arguments into a tuple**.
- Can you rewrite `sum_ave(*numbers)`?
- How about `average(*numbers)`?

Keyword Arguments

- Standard positional arguments
- Keyword arguments: we use the name (keyword) instead of position
  - Using the function is easier since we do not have to worry about the order of the arguments.
- Default arguments: Python allows function arguments to have default values; if the function is called without the argument, the argument gets its default value.

Custom Sort

- `sort(key, reverse)`
  - `sort` is mutator
  - `key` specifies a function that is used to extract a comparison key from each list element
  - `reverse` is a boolean value. If set to `True`, then the sorted order is reversed.

Set

- A set contains an **unordered** collection of **unique** and **immutable** objects.
- A set can be created from a sequence or iterable object.
- ```
  >>> s = set('hello world')
  >>> s
  {'r', 'd', ' ', 'l', 'h', 'w', 'e', 'o'}
  >>> type(s)
  <class 'set'>
  ```
- ```
  >>> nums = set([1, 2, 2, 4, 1, 5])
  >>> nums
  {1, 2, 4, 5}
  ```

`Set` with Immutable Objects

- ```
  >>> nums = set([[1, 2, 3], [1, 2]])
  …
  TypeError: unhashable type: ‘list'
  ```
- ```
  >>> nums = set([(1, 2, 3), (1, 2)])
  >>> nums
  {(1, 2), (1, 2, 3)}
  ```
- Why?
- ```
  >>> l = [1, 2, 3]
  >>> s = set((l))
  >>> s
  {1, 2, 3}
  >>> s = set((l,))
  TypeError: unhashable type: 'list'
  ```
  ```

hash()

```
>>> hash(1)
1
>>> hash('hello')
6718554301251688771
>>> hash([1, 2, 3])
TypeError: unhashable type: 'list'
>>> hash((1, 2, 3))
2528502973977326415
```

Set Operations

- `add(element)`
- `remove(element) / discard(element) / pop()`
- `clear()`
- `set1.difference(set2)`
- `set1.intersection(set2)`
- `set1.issubset(set2), set1.issuperset(set2)`

Comprehensions

- A comprehension is an expression that uses the same keywords as loop and conditional blocks, but inverts their order to focus on the data rather than on procedures.
  - Make Python code more compact and shift focus from the “how” to the “what.”
- List, set, dictionary comprehensions
- Multi-dimensional comprehensions

Random Numbers

- The `random` module
- `import random`
- `random.random()` returns a random float between 0.0 and 1.0 `[0.0, 1.0)`
- `random.randint(low, high)` returns an integer between low and high `[low, high]`
- `random.choice(seq)` returns an element from `seq` at random
- Roll an unbiased dice 10 times. How to simulate it using Python?

Recursion

- A recursive method is a method that calls itself.
  - A base case or termination condition that causes the method to end.
  - A non-base **simpler** case whose actions move the algorithm toward the base case and termination.
- The basic idea of recursion is to make a complicate problem simpler one step at a time.

Recursion

- Get help solving the problem from **co-workers (clones)** who work and act like you do
  - Delegate similar, smaller problem to **clone**
  - Clones follow **the same instruction**
  - Combine result from **clones** to solve the original problem
  - Work toward trivial version that is directly solvable
- Problems exhibit "self-similarity"
  - Solving larger problem means solving smaller problems within
- Can be expressed as direct, clear, and simple code
- Can intuitively model a task that is recursive in nature

Flood-Fill

- Determine the connected regions from a given graph or matrix
- Recursive solution
  - Iterate through all the vertices
  - If the current vertex is not visited, start a new region.
  - Recursively visits the non-visited neighbors of the current vertex
  - **Potential issue: Stack Overflow error**

Flood-Fill (Cont.)

- Non-recursive implementation
  - Uses Queue (FIFO)
  - Iterate through all the vertices
  - If the current vertex is not visited, push it into a queue
  - While the queue is not empty
    - Pop the first pushed vertex
    - Push the non-visited neighbors into the queue

Dynamic Programming

- A problem can be solved by dynamic programming
  - It has optimal sub-structures. (Recursive format)
  - It has overlapping sub-problems. And the number of distinct sub-problems is small.
- Uses a DP memo table to store the result of each distinct sub-problem.
- At the start of recursive function, check if the current state has been computed before and stored in the DP memo table.
- Top-down vs. bottom-up DP

Python OOP

- How represents a point in two-dimensional space?
- Instance attributes: dot notation to assign a value to an attribute on an object
- Methods vs. normal functions

Special Methods

- `__init__` method: initializes the attributes immediately after an object is created.
- `__str__` returns a string representation of an object.
- `def __eq__ (self, other):`

Modules

- How to organize different classes?
- It is recommended to have one file per class so that each file has a single purpose.
- But how to use a class from a different file?
- Module — a single Python file
- You can load a class from one module (file) for use in another module (file).
  - `import module`
  - `from module import class`
  - `from module import class as short_name`
  - Note that the module name is the file name.

Import Module

- A module (Python file) that defines a Python class may have the test code at the bottom.
- When you import a module, you want to use the defined class without executing its test code.
- Using `__name__` to check whether a module is imported or directly executed with `python` *`module.py`*
- ```
  if __name__ == '__main__':
      # test code here
  ```
  ```

Package

- A package is a collection of modules (i.e., files) in a folder.

More Special Methods

- `__add__, __mul__`
- `NotImplemented`: Special value which should be returned by the binary special methods to indicate that the operation is not implemented with respect to the other type

Persistent Storage

- File storage
  - Load the notes from a file.
  - Save the notes back to a file.
- `Pickle` module
  - The `pickle` module serializes an object into a stream of bytes that can be written to a file and later restored.
  - `dump()` and `load()` methods
  - Different from text file, `pickle` stores data in binary mode.

Inheritance

- The **is-a** relationship between the subclass and the superclass
  - A subclass is created from an existing superclass.
  - "If the superclass has the functionality, then the subclasses automatically get the same functionality."
- A subclass is bigger and more specific than a superclass by containing more data and more methods.
- For code reuse
- ```
  class SubClass(SuperClass):
      pass
  ```
  ```

Method Overriding

- Inheritance can add or change the behavior of the super class.
- Any inherited method in a superclass can be overridden in a subclass.
  - That is, a subclass redefines an inherited methods when it needs to change or extend the behavior of that method.
- Partial overriding: code for overriding a method includes a call to the superclass method `super().method()`
  - `super()` can be used to execute the method in the super class.
  - The subclass method wants to do what the superclass does, plus something extra.

Class Variables

- Class variable is part of the class definition and shared by all instances of the class.
  - Outside the method definitions
- Class variables can be accessed either using the class name or `self` (instance).
  - If a field cannot be found on the object, then it will be searched at the class level.
- On the other hand, instance variables are generated by assignments to `self` attributes in methods.

Class Method

- Instance Method
  - `def method(self, …) {…}`
  - `instance.method(…)`
  - `class.method(instance, …)`
- Class Method
  - `def method(…) {…}`
  - `class.method(…)`
  - ~~`instance.method(…)`~~

Multiple Inheritance

- A subclass that inherits from more than one super class is able to access functionality from all of super classes.

MRO – Method Resolution Order

- Diamond problem
- `>>> print(Pet.__mro__)`
- `(<class '__main__.Pet'>,`
  `<class '__main__.Dog'>,`
  `<class '__main__.Domestic'>,`
  `<class '__main__.Animal'>,`
  `<class 'object'>)`
- How to call `__init__` in both super classes?
- `**kwargs` **keyword** arguments

Encapsulation?

- Python does not support encapsulation.
- All methods and attributes on a class are publicly available.
- By convention, an internal attribute or method starts with underscore _.
- Name mangling happens when an attribute or method starts with a double underscore __.
  - `_classname__name`

Properties

- Getters and setters
  - Ensure data encapsulation
  - Enforce data validation
- Property - make methods look like attributes
  - Write code to use direct member access
  - Keep data validation

Exceptions

- Special error object
- Use exception to inform invalid operations
- `try: except:` block

Get User Input

- ```
  >>> a = input()
  10
  >>> a
  '10'
  ```
- `input()` function returns user input as a string
  ```

Deal with Invalid Input

- ```
  >>> a = int(input())
  hello
  ValueError: invalid literal for int()
  with base 10: ‘hello'
  ```
  ```

Python “Interface”

- In OOP, an interface is a set of public method on an object which can be called by other parts of a program.
- Interfaces are not necessary in Python because Python is duck-typing and already focused on how an object behaves.
- Python Abstract Base Classes `abc.ABC`

Attribute Processing

- Create a new attribute by setting its value
- Set the value of an existing attribute
- Get the value of an attribute
- Delete an attribute
- `AttributeError` exception

Inspection

- `instance.__class__`: which class the instance was created from
  - `instance.__class__.__name__`
  - `instance.__class__.__bases__`
- `instance.__dict__`: a dictionary for instance attributes only
- `dir(instance)`
- Inspection tool

Attribute Access

- `__getattr__()`:
  - If an attribute already has a value, `__getattr__()` is not used and the attribute value is returned.
  - If the attribute does not have a value, then `__getattr__()` is given a chance to return a meaningful value.
  - If an attribute is not supported, it should raise an `AttributeError` exception.
- `__setattr__()`: create and set attributes
  - Intercept all attribute assignments

Descriptors

- Descriptors provide an alternative way to intercept attribute access.
- Descriptor protocol routes a specific attribute's get and set operations to methods of a separate class object.
  - Descriptor can have its own state.
- Descriptors are created as independent classes.
- A descriptor manages a single, specific attribute and provides control over both fetch and assignment access

Descriptor Basics

- Descriptor's specific methods:
  - `__get__`, `__set__`, `__delete__`
- Instance of the descriptor class assigned to another class's **class attribute**
- When the attribute is accessed, those descriptor's special methods are automatically invoked.
  - If any method is missing, it means that the corresponding type of access is not supported.

Descriptor Methods

- ```
  class Descriptor:
      def __get__(self, instance, owner):
      def __set__(self, instance, value):
      def __delete__(self, instance):
  ```
- `self`: descriptor class instance
- `instance`: instance of the client class to which the descriptor instance is attached `instance.attr`
  - If `instance` is `None`, then `class.attr`
- `owner`: the client class to which the descriptor instance is attached
  ```

`__call__`

- `instance.__call__(self[, args…])`
  - Called when `instance` is "called" as a function
  - `instance(arg1, arg2, …)` is a shorthand for `instance.__call__(arg1, args2, …)`

Closure

- A closure is a **nested** function object that remember values in enclosing scopes even if the enclosing scopes are not in memory.
- A nested function is a function defined inside another function and can access the local variables of the enclosing function (enclosing scope).
  - The nested function **cannot modify** the variables in the enclosing function.
  - Closure behaves as a pure function.

Iterator

- `iter(iterable)`: get an `iterator` from an `iterable`.
- `next(iterator)`: get a new element from the `iterable` until the `iterable` is exhausted and returns a `StopIteration` exception.
- Manual iteration
- Iteration simulation
  - __iter__, __next__

How does Python for-loop work?

- Any iterable object can be used in a for loop.
  - `next()` —> `__next__`
  - `iter()` —> `__iter__`
- Iterator is also iterable.

Generator

- If a function uses the `yield` statement, it defines a generator, which produces a sequence of values for use in iteration.
- When the function is called, none of its code starts executing. Only the generator is returned.
- The generator object executes the function whenever the `next(generator)` is called.

Generator (cont.)

- When `next()` is invoked, the generator function executes statements until it reaches a `yield` statement.
- The `yield` statement returns a result, which is the value following the `yield` keyword in the same statement.
- The execution resumes with the statement following the `yield` statement when another `next()` is invoked.
- If there is a `return` statement in the code of a generator or the generator function finishes, the execution will stop with a `StopIteration` exception.

Generator vs. Iterator

- A generator is also an iterator.
- Generators simplify the creation of iterators. A generator is a function that produces a sequence of values instead of a single value.

Generator Expressions

- Generator expressions are the generator version of list comprehension.
- It returns a generator instead of a list.
- Generator expression vs. list comprehension
  - They both do iteration.
  - List comprehension creates the entire list in memory while generator expression will create items on the fly.
  - Generator expression does not support indexing, slicing, addition, etc.

Coroutine

- Inside a generator, the `yield` statement can be used as an expression on the right side of an assignment operator.
- Call `next()` so that the coroutine executes statements leading to the first `yield` expression.
- Then, the coroutine suspends, waiting for a value to be sent to it using the `send()` method of the associated generator object.
- The value passed to `send()` is returned by the `yield` expression.
- Upon receiving a value, the coroutine executes statements until the next `yield` statement is encountered.
- Use `close()` to shut down a coroutine.

Decorator

- Aspect-oriented programming (AOP)
- OOP means is-a relationship, while AOP means act-as relationship
- In Python, we can apply a decorator (`@d`) to a function (`F`) to create a new function (`F'`)
  - `F' = @d (F)`

Context Manager

- To properly manage resources, e.g., opened files
  - Close each opened file after it has been used