CSCI 145 · Exam 3 Reference Sheet (Lectures 10–14) — page 1 of 2

RECURSION

A **recursive** function calls itself on a smaller version of the problem

- **Base case:** answered directly, with no recursive call. A function can have more than one
- **Recursive case:** calls the function on a smaller input, closer to the base case
- Each call gets its own frame on the **call stack**, with its own variables

Input	Smaller input	Typical base case
`a number n`	`n - 1`	`n == 0` or `n <= 1`
`a string word`	`word[1:]`	`len(word) == 0`
`a list nums`	`nums[1:]`	`len(nums) == 0`

- **Writing one:** decide the base cases first, then trust the recursive call
- With no base case, or a step that skips past it, the calls never stop: after about 1,000 frames, Python stops with a **RecursionError**

PENDING OPERATIONS

A **pending operation** is work done after the call returns

- Work before the call happens on the way down
- Work after the call happens on the way back up, last call first

```
def list_sum(nums):
    if len(nums) == 0:
        return 0
    return nums[0] + list_sum(nums[1:])
```

```
list_sum([4, 7, 2]) = 4 + (7 + (2 + 0)) = 13
```

- In a function that computes and returns an answer (not a turtle drawing), the recursive case must **return** the combined value. Without that `return`, the function returns `None`

TAIL RECURSION OR PENDING OPERATION?

Tail	The call is the last thing done: `return f(nums[1:])`
Pending	Work is left after the call: `return nums[0] + f(nums[1:])`

TWO RECURSIVE CALLS

```
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

The + waits for two calls; smaller calls repeat

THE TURTLE (import turtle as t)

`t.forward(distance)`	Move forward
`t.backward(distance)`	Move backward
`t.left(angle)`	Turn left by angle degrees
`t.right(angle)`	Turn right by angle degrees
`t.up() / t.down()`	Lift the pen / put it down

TUPLES (IMMUTABLE, LIKE STRINGS)

`point = (3, 7)`	A tuple of a few related values
`point[0]`	Indexing and slicing, as for lists
`len(point)`	Also `len` and `in`, as for lists
`point[0] = 5`	**TypeError**: a tuple can't change
`(5,)`	A tuple with one value needs a comma
`point = 3, 7`	Commas make a tuple (packing)
`x, y = point`	Unpacking: one name per value, or a **ValueError**
`a, b = b, a`	Swaps a and b
`return x, y`	Returns the tuple `(x, y)`
`points = [(3, 7)]`	A list of tuples
`points.append((x, y))`	Adds one tuple to the list
`for x, y in points:`	Unpacks each tuple in the list

DICTIONARIES (MUTABLE, LIKE LISTS)

`{}`	An empty dictionary
`{"Ana": 3, "Ben": 5}`	Keys `"Ana"` and `"Ben"` map to 3 and 5
`d[key]`	The value for key; a missing key is a **KeyError**
`d[key] = value`	Changes the value for key (it keeps its position), or adds a new item at the end
`d[key] += 1`	Adds 1 to the value for an existing key
`key in d`	`True` if key is a key in d. Otherwise, `False` (checks keys, not values)
`d.get(key, default)`	The value for key if key is in d. Otherwise, default
`len(d)`	The number of items (key: value pairs)
`for key in d:`	Loops over the keys, in the order they were added
`d.items()`	`(key, value)` tuples: `for key, value in d.items():`
`d.values()`	Just the values: `sum(d.values())`

- Keys are unique and immutable: strings, numbers, tuples. A list key is a **TypeError**
- Values can be anything and can repeat

COUNTING WITH A DICTIONARY

```
counts = {}
for item in items:
    if item in counts:
        counts[item] += 1
    else:
        counts[item] = 1
```

The first time: start the count at 1. After that: add 1

CSCI 145 · Exam 3 Reference Sheet (Lectures 10–14) — page 2 of 2

SETS (NO DUPLICATES, NO ORDER)

`{3, 1, 2}`	A set: curly braces, no colons
`set(nums)`	A set from a list; duplicates disappear
`set(word)`	A set of the letters in a string
`set()`	An empty set; `{}` is an empty **dictionary**
`len(s)`	The number of values in s
`x in s`	`True` if x is in s. Otherwise, `False`
`s.add(x)`	Puts x in s; if already there, nothing happens
`s.remove(x)`	Takes x out; if not there, a **KeyError**
`s.discard(x)`	Takes x out if it's there; no error
`s[0]`	**TypeError**: no order, so no indexing
`sorted(s)`	A sorted list of the values in s

- A set may print its values in any order
- Values must be immutable, like dictionary keys

`a & b`	In both: the intersection	`a.intersection(b)`
`a | b`	In either: the union	`a.union(b)`
`a - b`	In a, not in b: the difference	`a.difference(b)`

WHICH WOULD YOU USE?

	List	Tuple	Dictionary	Set
Looks like	`[3, 7]`	`(3, 7)`	`{"Ana": 92}`	`{3, 7}`
Find an item by	Position	Position	Key	Only `in`
Can change?	Yes	No	Yes	Yes
Duplicates?	Yes	Yes	Not as keys	No

REFERENCES

- A variable holds a **reference** to an object

`b = a`	An **alias**: a second name for the same object
`b.append(5)`	Changes the object; every name for it sees it
`b = [5]`	b names a new object; a doesn't change
`a == b`	The same value?
`a is b`	The same object?
`id(a)`	A number that names the object

- A function gets a reference: if it changes a list, dictionary, or set passed in, the caller sees the change
- A function with no `return` returns `None`
- Numbers, strings, and tuples are immutable, so sharing them is harmless

COPYING A LIST OF LISTS

`b = grid`	**Alias**: the same list
`b = grid.copy()`	**Shallow copy**: rows shared
`b = copy.deepcopy(grid)`	**Deep copy**: nothing shared

- Shallow copy: a new outer list holding the same rows (also `grid[:]`, `list(grid)`). Replacing a row (`grid[0] = [5, 6]`) changes one list only; a change inside a row (`grid[1][0] = 7`) shows up through both
- `copy.deepcopy` needs `import copy`

WRITING

`f.write(text)`	Writes text; adds no newline
`f.write(name + "\n")`	Writes one line
`f.write(str(92))`	`f.write(92)` is a **TypeError**

TEXT FILES

- A text file is one long string, with `\n` (newline) at the end of each line
- Escapes: `\n` newline, `\t` tab, `\"` quote, `\\` backslash
- `print` shows `\n` as a line break, then adds its own newline

```
with open(filename, mode) as f:
    read from or write to f here
the file is closed here
```

Mode	Means	If the file exists	If it doesn't
`"r"`	Read (default)	Reads it	**FileNotFoundError**
`"w"`	Write	Erases it first	Creates it
`"a"`	Append	Adds to the end	Creates it

- Without `with`: `f = open(filename)` … `f.close()`
- A name alone: Python looks in the program's folder

READING

`f.read()`	The whole file as one string
`f.readlines()`	A list of lines; each keeps its `\n`
`for line in f:`	One line per pass; each keeps its `\n`
`line.strip()`	Removes spaces and `\n` at both ends
`line.split(",")`	A list of strings, split at each comma
`line.split()`	A list of the words, split at spaces
`int(text)`	A string to a number; also `float`
`if line.strip():`	Skips a blank line: `int("")` crashes
`lines[1:]`	Skips the header line

Everything read from a file is a **string**

CSV (import csv)

A header line names the columns; then one row per line

`rows = list(csv.reader(f))`	A list of lists of strings
`writer = csv.writer(f)`	A writer for the open file
`writer.writerow(row)`	Writes a list as one line

- Open with `open(filename, "w", newline="")` to write
- `writerow` adds the commas and the newline, and converts numbers
- A field with a comma inside goes in quotes: `csv.reader` handles them, `split(",")` doesn't
- CSV has no types: every field comes back a string

JSON (import json)

`data = json.load(f)`	File to Python data
`json.dump(data, f)`	Python data to file

- Like Python dictionaries and lists; can nest
- Keeps types; JSON's `true` is Python's `True`
- A table becomes a list of dictionaries

CSV OR JSON?

	CSV	JSON
Looks like	A table	Dictionaries and lists
Types	All strings	Kept
Shape	Same fields per row	Any nesting

Lines of words, like a song's: plain text