CS 231
Naomi Nishimura

Python session 1

Note: Although there may not be time in the session itself for full use of the design recipe, it is highly recommended that you write the purpose, effects, and contract for each function before checking the model solutions. In that way, you will have a chance to practice use of the design recipe before being marked on it in assignments.

1 Python basics and `grids.py`

Files to download: `check.py`, `grids.py`
Resource to use: Grids (linked off Python page)
To try:

- Before looking at the sample solution, try to write a function `same_value` that consumes a grid and four numbers representing the row and column numbers of two positions in the grid (first the row and column numbers of the first position and then the row and column numbers of the second position) and produces `True` if the data items stored in the two positions are the same and `False` otherwise.
- Import the module `check.py` so that you can use `check.expect` for a test.
- Import the module `grids.py` so that you can use the methods `access` and `enter` *without* having to put `grid.` before each use of a method.
- See what happens if you remove the return statement (you can do this by making it into a comment).

Sample solution: `sess1q1samevalue.py` with tests in `sess1q1samevalueuse.py`
Python syntax to notice:

- Use of `import` to import a module and the use of dot notation to use a function in the module
- Use `from grids import *` for module `grids.py` in order to avoid having to use the name of the module before the functions `Grid` and `make_grid`
- Use of `def` for a function definition
- Indentation of the body of the function
- Use of `return` to return a value
- Syntax to call the function
- Appearance of `None` to show that no value was returned when the `return` statement was commented out

Python style to notice:

- Identifiers are in lower case, with words connected by underscores
- Lines are indented four spaces
- Aspects of the design recipe appear in comments; examples (and thorough tests) are missing

2 Branching

Files to download: `check.py`
To try:

- Before looking at the sample solution, try to write a function `find_grade` that consumes an integer mark and produces a letter grade (a string), where a mark of 80 or above receives `"A"`, a mark of 70-79 receives a `"B"`, and a mark of below 70 receives a `"C"`.
- Import the module `check.py` and use `check.expect` to write tests for your function.
- Try experimenting with using either nested branching or `elif` for two different solutions, giving your second function the name `find_grade_alt`.
- For the sake of completeness, the solutions also contain `find_grad_extra`, which does not use `else` at all. You are welcome to try this option as well.

Sample solution: `sess1q2findgrade.py`
Python syntax to notice:

- Use of colons after branching statements
- Use of indentation

3 Looping and grids.py

Files to download: `check.py`, `grids.py`
Resource to use: Grids (linked off Python page)
To try:

- Before looking at the sample solution, try to write a function `subgrid` that consumes a grid and four integers (the first and last rows and first and last columns) and produces a grid containing all the data items in the given rows and columns (inclusive). For example, if the subgrid should go from row 10 to row 15 and from column 2 to column 4, the order of the four integers should be 10, 15, 2, 4.
- You can assume that the first row is no greater than the last row, that the first column is no greater than the last column, and all rows are columns are present in the input grid.

- Try writing two different solutions, one using `for` loops and the other using `while` loops, giving your second function the name `subgrid_alt`.

Sample solution: `sess1q3subgrid.py` with tests in `sess1q3subgriduse.py`
Python syntax to notice:

- Placement of initialization and update of values used in `while` loop conditions

4 List mutation

Files to download: `check.py`
To try:

- Before looking at the sample solution, try to write a function `change_list` that mutates a list of integers by multiplying the first half of the numbers by 10 (where for an odd list, the "first half" is smaller than the "second half").
- See what happens when you try to print the result of a function call.

Sample solution: `sess1q4changelist.py`
Python syntax to notice:

- Use of square brackets in a list to access a particular item

5 Looping, lists, and combinations

Files to download: `check.py`, `equiv.py`
To try:

- Before looking at the sample solution, try to write a function `all_triples` that consumes a list and produces a list of lists of length three. The list of lists that is produced will contain each possible group of three items found in distinct positions in the list. In a group, order does not matter.
- Each group consists of items in distinct positions in the list, so if the input list is of length less than three, there are no possible groups.
- Each group appears exactly once. Thus, if the input list is `[1, 2, 3, 4]`, the list of lists will contain exactly one of `[1, 2, 3]`, `[1, 3, 2]`, `[2, 1, 3]`, `[2, 3, 1]`, `[3, 1, 2]`, and `[3, 2, 1]`, as they all represent the group containing `1`, `2`, and `3`.
- Although each set contains items in distinct positions in the input list, if a particular value appears in more than one position, a group can contain that value more than once. For example, if the input list is `[1, 1, 5, 6, 7]`, the group consisting of the items in the first three positions contains two `1`'s and a `5`.
- Try using different ranges for the three nested loops to see what happens.

- Do not use `itertools` for this exercise.
- Use `equiv.py` in your tests.

Sample solution: `sess1q5alltriples.py`
Python syntax to notice:

- Creation of a new list using `[]`
- Use of `range` in the `for` loop
- Use of dot notation for the method `append`
- Use of `equiv.py`

6 Itertools and combinations

Files to download: `check.py`, `equiv.py`
To try:

- Before looking at the sample solution, try to write a function `all_triples_itertools` that consumes a list and produces a list of lists of length three, as described in the previous exercise.
- Use `itertools` to write your function.
- Use `equiv.py` in your tests.

Sample solution: `sess1q6alltriplesitertools.py`
Python syntax to notice:

- Use of `import` for the `itertools` module
- Use of `list` to convert the result to a list

7 Graphs

Files to download: `check.py`, `graphs.py`, `equiv.py`, `samplegraph4.txt`
Resources to use: Graphs (linked off Python page), Equivalent lists (linked off Python page)
To try:

- Before looking at the sample solution, try to write a function `tour_cost` that consumes a graph and a list of all the vertices of the graph in some order and produces the total cost of the cycle formed by the vertices in the given order. You can assume that there are edges between all pairs of vertices in the graph, and hence any ordering of the vertices forms a cycle. The cost of the cycle is the sum of the weights of the edges in the cycle.

- Make sure to remember the edge from the last vertex back to the first.
- You can try your function on a sample graph that appears in the information on the graph module, using `make_graph` to create the graph from a file.

Sample solution: `sess1q7tourcost.py` with tests in `sess1q7tourcostuse.py`
Python syntax to notice:

- Use of methods and functions in the module graph
- Use of `range`
- Use of `%` in the alternate solution `tourcost_alt`

8 Exhaustive search for TSP

Files to download: `check.py`, `graphs.py`, `equiv.py`, `sess1q7tourcost.py`, `samplegraph4.txt`
Resources to use: Graphs (linked off Python page), Equivalent lists (linked off Python page)
To try:

- Before looking at the sample solution, try to write a function `exhaustive_tsp` that consumes a graph and uses `tour_cost` to determine the cheapest tour through the vertices of the graph. You can assume that there are edges between all pairs of vertices in the graph.
- Use `itertools` as explained in lecture Module 1.

Sample solution: `sess1q8exhaustivetsp.py` with tests in `sess1q8exhaustivetspuse.py`
Python syntax to notice:

- Use of slice in a list
- Use of for loop to iterate in a list
- Use of `list` to convert a permutation into a list
- Use of `in` in testing and a list of all possible correct solutions