Checklist02

Python Scripting / Data Processing

1. How is **Python** different from the **Bourne** shell? How is it similar?
   - Similar:
     - both interpreted
     - both used for scripting and automation
     - both can read from stdin, write to stdout
     - both can use command line args
     - both can run external programs
   - Different:
     - Python:
       - python is general purpose programming language with rich data structures (lists, dictionaries, etc.)
       - python has strong **types**
     - Bourne Shell:
       - mainly a command interpreter
       - used to run programs and automate Unix commands
       - mostly treats data as strings
2. How do we manage control flow in **Python?** How do we utilize these constructs?
   - **Conditionals**:
     - used to make decision in a program
     - `if` , `elif` , and `else`
     - example:
       - ```
         if x > 0:
             print("positive")
         ```
   - **Loops**:
     - used to repeat code
     - `for` and `while`
     - example:
       - ```
         for line in file:
           print(line)
         ```
   - **Exceptions**:
     - used to handle errors gracefully

- `try` and `except`
- example:
- ```
  try:
      x = int(input())
  except ValueError:
      print("Invalid number")
  ```

- **Functions**:
  - used to organize and reuse code
  - defined with `def`
  - example:

    ```
    def square(x):
        return x*x
    ```

3. What **data structures** do we have in **Python?** What are their basic operations? When would we want to use each type of data structure?
   - **Lists**:
     - an ordered **mutable** sequence
     - Basic operations:
       - `append()`
       - `pop()`
       - indexing: `myList[i]`
       - slicing
       - iteration
     - **When to use:**
       - when order matters
       - when you need to modify elements
       - when duplicates are allowed
   - **Tuples**:
     - an ordered, **immutable** sequence
     - Basic operations:
       - indexing
       - unpacking
       - iteration
     - When to use:
       - when data should not change
       - when representing fixed records
       - when you want something hashable (can be a dictionary key)
    ```

- **Dictionaries**:
  - an unordered (in concept), mutable mapping of keys to values
  - Basic operations:
    - `d[key]`
    - `d[key] = value`
    - `.get()`
    - `.keys()`, `.values()`, `.items()`
  - When to use:
    - when you need fast lookup by key
    - when counting things
    - when mapping one value to another
- **Sets**:
  - an unordered collection of unique elements
  - Basic operations:
    - `add()`
    - `remove()`
    - membership test (`in`)
    - set operations (`union`, `intersection`)
  - When to use:
    - When you only care about unique values
    - When checking membership quickly
    - When performing mathematical set operations

4. How do we do the following in Python?
   - process command-line arguments:
     - ```
       import sys

       sys.agrv #contains the command line args as STRINGS

       #Example:
       arguments = sys.argv[1:]
       while arguments:
         arg = arguments.pop(0)
         match arg:
              case 'd': print("D!")
              case _: print("not d!")
       ```
   - read from standard input (line by line)
     - ```
       import sys
       ```

```
#example:
for line in sys.stdin: #most common
  line = line.strip()
  print(line)

#example:
while True:
  line = sys.stdin.readline()
  if not line:
       break
  print(line.strip())

#interactively prompting user:
name = input('What is your name?')
print(f'Hello {name}')
```

- read input from files:
  - ```
    #example: best practice
    with open("filename.txt", "r") as f:
      for line in f:
           print(line.strip())

    #example:
    f = open("filename.txt", "r")
    for line in f:
      print(line.strip())
    f.close()
    ```
- access environmental variables
  - ```
    import os

    os.environ #dictionary containing all env variables

    #accessing a variable
    # !!! raises a KeyError if the variable does not exist !!!
    home = os.environ["HOME"]

    #safer way
    home = os.environ.get("HOME")
    ```
- change the case of a string

-
```
s = "Hello World"

s.lower() # "hello world"
s.upper() # "HELLO WORLD"
s.title() # "Hello World"
s.capitalize() # "Hello world"

# !!! Remember strings are IMMUTABLE !!!
# !!! so these methods return a new sting !!!
```

- split a string, combine a list of strings
  -
```
#split()

#example:
s = "one two three"
words = s.split() # ['one', 'two', 'three']
#example:
s = "a,b,c"
parts = s.split(",") #['a','b','c']

#join()

#example:
words = ['one', 'two', 'three']
s = " ".join(words) # "one two three"

# !!! join() is called on a seperator string !!!
# !!! all elements in the list must be strings !!!

#example:
",".join(['a','b','c']) # "a,b,c"
```

- slice a list
  -
```
myList = [0,1,2,3,4,5]

myList[1:4] #[1,2,3]
myList[:3] # [0,1,2]
myList[3:] # [3,4,5]
myList[::2] #[0,2,4]
myList[::-1] #[5,4,3,2,1,0]

myList[1:5:2] # [1,3]
```

```
myList[-1] # 5
myList[:-1] # [0,1,2,3,4]
```

- execute an external command
  - ```
    import os
    os.system("ls -l")
    ```
Unix Tool	Function
cd	os.chdir(path)
chmod	os.chmod(path, mode)
ln	os.link(src, dst)
mkdir	os.mkdir(path)
pwd	os.getcwd()
rm	os.unlink(path)
mv	os.rename(src, dst)
ln -s	os.symlink(src, dst)
- process JSON data from the web
  - ```
    import request

    url = https://example.com/data.json

    response = requests.get(url)
    data = response.json() #data matches the top level data structure
    ```
- count with dictionaries
  - ```
    # pythonic example
    counts = {}

    for item in data:
      counts[item] = counts.get(item, 0) + 1
      # saying if counts[item] exists return its value
      # if it does not then return 0

    #example:
    counts = {}

    for item in data:
      if item in counts:
           counts[item] += 1
      else:
           counts[item] = 1
    ```
    ```

```
#example:
words = ["a", "b", "a", "c", "b", "a"]
counts = {}

for w in words:
  counts[w] = counts.get(w,0) + 1
# {'a' : 3, 'b' : 2, 'c' : 1}
```

- sort lists and dictionaries
  - ```
    myList = [3,1,4,2]

    sorted_list = sorted(myList) # retunrs [1,2,3,4], NOT in place
    myList.sort() #sorts list in place -> [1,2,3,4]

    words = ["apple", "pear", "banana"]
    sorted(words, key=len) # ['pear', 'apple', 'banana']

    counts = {'apple': 3, 'banana': 2, 'pear': 1}

    sorted_items = sorted(counts.items(), key = lambda item: item[1],
    reverse = True)

    print(sorted_items)
    #returns [('apple', 3), ('banana', 2), ('pear', 1)]
    ```

5. How do we test a Python script?
   - Using `doctest`
     - ```
       def add(a,b):
         """
         Returns the sum of a and b.
         >>> add(2,3)
         5
         >>> add (-1,1)
         0
         """
         return a + b

       if __name__ == "__main__":
         import doctest
         doctest.testmod()
       ```
     - ```
       python -m doctest -v script.py
       python doctestfn {scriptName} {fileName}
       ```
       ```

- Using `mypy`
  - ```
    def square(x: int) -> int:
      return x * x
    ```
  - ```
    mypy myScript.py
    ```
- Using `unittest`
  - ```
    import unittest

    def add(a,b):
      return a + b

    class TestAdd(unittest.TestCase):
      def test_positive(self):
           self.assertEqual(add(2,3),5)
      def test_negative(self):
           self.assertEqual(add(-1,1), 0)

    if __name__ == "__main__":
      unittest.main()
    ```
  - ```
    python script.py
    python -m unittest script.py
    #automatically discovers test ases in the script
    ```

Functional Programming:

1. What is the difference between **imperative** and **functional** programming?
   - **Imperative:**
     - An **imperative program** is written as a list of instructions, telling the computer, **step-by-step** what to do.
     - Example:
       - ```
         doubles = []
         for n in numbers:
           doubles.append(n*2)
         ```
   - **Functional:**
     - A **functional program** is a **composition** of **functions**.
     - stateless (no side-effects)
         ```

        - order of evaluation undefined
        - Example:
            - ```
              doubles = map(lambda n: 2*n, numbers)
              ```
    - "**Functional programming** is like describing your problem to a **mathematician**. **Imperative programming** is like giving instructions to an **idiot.**"
2. What are some benefits of **functional programming:**
    - No side effects → easier to reason about
    - Modular and composable → functions can be combined
    - Easier parallelization → no shared mutable state
    - Concise and expressive → less boilerplate
    - Often easier to test and debug
3. How is **functional** programming related to the **Unix Philosophy?**
    - "Do one thing and do it well" → pure functions are self-contained
    - "Chain programs together" → functional pipelines (`map`, `filter`, `reduce`) resemble Unix pipelines (`|`)
    - Encourages small, reusable, composable pieces of code
    - Example:
        - ```
          cat numbers | grep even | awk '{print $1^2}'
          ```
        - ```
          numbers = [1, 2, 3, 4, 5]
          squares = list(map(lambda x: x**2, filter(lambda x: x % 2 == 0, numbers)))
          ```
4. How do we use `map`, `filter` and `lambda` to do **functional** programming in **Python?**
    - `map`
        - Basic Synatax:
            - `map(function, iterable)`
        - Examples:
            - ```
              nums = [1,2,3,4]
              squared = map(lambda x: x*x, nums)
              print(list(squared)) #[1,4,9,16]
              ```
            - remember map returns an iterator, not a list
    - `filter`
        - Basic syntax:
            - `filter(function, iterable)`
        - Examples:

        - ```
          nums = [1,2,3,4,5,6]
          evens = filter(lambda x:x%2 == 0, nums)

          print(list(evens))
          ```
    - remember filter returns an iterator, not a list
- `lambda` functions
    - Basic syntax:
        - `lambda parameters: expression`
    - Examples:
        - ```
          lambda x : x * 2

          #this is equivalent to
          def double(x):
            return x*2

          # Common use : sorting
          #this example sorts by grades
          students = [("Aryan", 90), ("Bob",80)]
          students.sort(key=lambda x : x[1])
          ```

List Comprehensions

- Basic syntax:
    - `[expression for item in iterable if condition]`
- Examples:
    - ```
      nums = [1,2,3,4]
      #squaring numbers
      squares = [x*x for x in nums]
      #filter
      evens = [x for x in nums if x%2 == 0]

      #combnining map and filter
      results = [x*x for x in nums if x%2 == 0]
      # this replaces:
      list(map(lambda x: x*x, filter(lambda x: x%2==0, nums)))

      #dict comprehension
      square_dict = {x: x*x for x in nums if x%2 == 0}
      ```
      ```

```
# {1:1, 2:4, 3:9, 4:16}

#set comprehension
even_set = {x for x in nums if x%2 == 0}
# {2, 4}
```

Generators

1. What is an **iterator** and how is it different from a **list?**
   - **Definition**: An **iterator** is an object that produces elements one at a time
   - **Memory**: Generates elements on-the-fly : so its **memory efficient**
   - **Reusability**: Can usually be consumed only once, raises `StopIteration`
   - **Methods**: supports `next()`
     - DOES NOT support indexing
   - **Creation**: creating using generator expression or `iter([1,2,3])`
2. What is a **generator** and how is it different from a list?
   - **Definition**: A function or expression that **yields** elements one at a time
   - **Memory**: Generates elements on-the-fly; **very memory efficient**
   - **Reusability**: Usually consumed once; must recreate to reuse
   - **Creation**: Created using generator function with `yield` or generator expression `(x for x in range(3))`
   - **Evaluation**: **Lazy:** elements only computed only when needed
3. What does the **yield** command do:
   - produces a value without exiting the funciton
   - when the function is called, it returns a generator object
   - each call to `next()` resumes the function from where it **last yielded**, not from the beginning
     - after all values are yielded, the generator raises a `StopIteration`
4. Why would we use a **generator** over a **list** (and vice versa)?
   - **Generator over list**:
     - memory efficient
     - Lazy evaluation: only computes values when needed
     - Infinite Sequences: can represent streams or infinite sequences that a list cannot hold in memory
   - **List over Generator**
     - random access/ indexing: Can access any element quickly

        - Multiple iterations: Can iterate over the data multiple times
            - Immediate availability: All values are computed upfront, so no `StopIteration` surprises

5. How do we convert a **list comprehension** to a **generator** expression?
    - replace `[]` with `()`
    - this makes it more memory efficient, since elements computed on the fly
    - generators are lazy, list comprehension is eager
    - Example:
        - ```
          #list comprehension
          [x**2 for x in range(5)]

          #generator expression
          (x**2 for x in range(5))
          ```

6. How we use common **iterators** and **generators** such as:
    - `sorted`
        - basic syntax: `sorted (iterable, key=None, reverse=False)`
        - returns a new sorted list from any iterable
        - does not modify original
        - iterable $\rightarrow$ list output, not a generator
        - Example:
            - ```
              lst = [3,1,2]
              for x in sorted(lst):
                print(x)
              #1 2 3
              ```
    - `reversed`
        - basic syntax: `reverse(sequence)`
        - returns an **iterator** that yields elements in reverse order
        - does not create a new list (memory efficient for large sequences)
        - Example:
            - ```
              lst = [1,2,3]
              for x in reversed(lst):
                print(x)
              # 3 2 1
              ```
    - `range`
        - basic syntax: `range(start, stop, step)`
        - returns a **generator-like object** (range object)
        - Lazy; elements produced one at a time
              ```

    - Supports iteration but not random modification
    - Example:
        - ```
          for i in range(3):
            print(i)
          # 0 1 2
          ```
- `enumerate`
    - basic syntax: `enumerate(iterable, start = 0)`
    - returns an iterator of `(index, value)` pairs.
    - Useful for tracking positions while iterating
    - Example:
        - ```
          lst = ["a", "b"]
          for i, val in enumerate(lst, 1):
            print(i,val)
          # 1 a
          # 2 b
          ```

Concurrency and Parallelism

1. What is the difference between **concurrency** and **parallelism**?
    - **Concurrency**
        - **composition** of independent computations
        - concerned about **structure**
        - multiple tasks making progress at the same time, but not necessarily at the same instant
    - **Parallelism**
        - **Simultaneous** execution of (possibly related) computations
        - Concerned with **execution** (with hardware resources)
        - multiple tasks executing literally at the same time, usually on multiple cores
    - "**Concurrency** provides a way to structure a solution to solve a problem that may (but not necessarily) be **parallelizable**"
2. What are some problems with **concurrency?**
    - **Race Condition:** multiple tasks compete for the same resource in an unpredictable manner
    - **Deadlock:** multiple task are stuck waiting for each other (indefinetly)
3. How does **functional** programming help enable **concurrency** and avoid these problems?
    - **No shared mutable state**:

    - Pure functions do not modify external variables, so multiple tasks can run concurrently without interfering with each other
    - prevents **race conditions**
  - **Immutability:**
    - Data structures are immutable → tasks don't overwrite each other's data
    - reduces **resource contention**
  - **Composable, small functions**:
    - Functions are self-contained and stateless → easier to schedule concurrently
    - Makes debugging concurrent tasks simpler
4. What does it mean for a problem to be **task parallel, data parallel**, or **embarrassingly parallel**? Examples?
  - **Task Parallelism:**
    - Different tasks (functions) run in parallel, possibly on the same or different data
    - Focus: Different work at the same time
    - LOOK FOR GENERATORS
    - Example:
      - Downloading files while processing others
      - In Python: one thread compresses images while another uploads them
  - **Data Parallelism**:
    - The same operation is applied to different pieces of data in parallel
    - Focus: **Same work**, multiple data items
    - LOOK FOR `map()` and/or `filter()`
    - Example:
      - Doubling every number in a huge list using multiple cores
      - Python: `map(f, large_list)` executed with multiprocessing
  - **Embarrassingly Parallel:**
    - Tasks are completely **independent**, no communication or synchronization required
    - Focus: Easily splitting tasks across processors
    - Example:
      - Rendering frames of an animation
      - Running simulations with different random seeds
  - Data parallelism usually implies embarrassingly parallel

Same task 3 different ways:

```python
# Given a stream of numbers, return odd numbers

# Imperative
def odds(stream):
    results = []
    for line in stream:
        n = int(line)
        if n % 2:
            results.append(n)
    return results

# Functional (map + filter) → returns iterator
def odds_fp(stream):
    return filter(lambda x: x % 2,
                  map(int, stream.read().split()))


# List Comprehension → returns list
def odds_lc(stream):
    return [int(x) for x in stream.read().split()
            if int(x) % 2]


# Generator (yield) → returns generator
def odds_gen(stream):
    for x in stream.read().split():
        n = int(x)
        if n % 2:
            yield n
```