Tuesday Class:

I will have to end today's tutorial at **10.40am** as I need to go to the MST venue to help with setups, sorry!

Week 7 Tutorial

COMP10001 – Foundations of Computing

Semester 1, 2025

Clement Chau

- Code structure
- Variable names
- Test cases
- Commenting

Pre-MST Checklist:

- I know **when** to sit the MST (11am **or** 12pm)
- I know **where** to go to sit the MST (We have **5+ locations** in total!)
- I have obtained my **student ID** from **Stop 1**.
- I will bring something **to write with** in the MST **(pen/pencil)**

program.py

```python
GREEN = "green"
YELLO = "yellow"
GREYY = "grey"

def set_colors(secret, guess):
    '''
    Compares the latest `guess` equation against the unknown `secret` one.
    Returns a list of three-item tuples, one tuple for each character position
    in the two equations:
        -- a position number within the `guess`, counting from zero;
        -- the character at that position of `guess`;
        -- one of "green", "yellow", or "grey", to indicate the status of
           the `guess` at that position, relative to `secret`.
    The return list is sorted by position.
    '''
    mylist = []
    verified = []
    secret = list(secret)
    for i in range(len(guess)):
        if guess[i] == secret[i]:
            # Status of "guess" at that position is "green"
            mylist.append((i, guess[i], GREEN))
            verified.append(i)
            secret[i] = ""

    for i in range(len(guess)):
        # For all position in "guess" that is not green
        if i not in verified:
            if guess[i] in secret:
```

What kind of "commenting" are we expecting in this subject?

Docstrings are needed to describe what your **function** does at a high-level.

Describes **the input**, **what the function does with the input**, and the **output returned**.

Sufficient **Comments** are needed throughout your program to describe **not-as-obvious parts** of the program.

Project 1 Marking Guidelines

- Manual marking by tutors for a total of 1-2 points per task on the following criteria:
  - **Approach (0.5-1.5 points)**
    - Your code structure is intuitive, easy to understand and not overly complicated. You should use helper functions when needed.
  - **Style (0.2 points)**
    - Your code is formatted well with no PEP8 errors, no “magic” numbers or strings, and good variable names.
  - **Comments (0.3 points)**
    - Your docstring's and comments have the correct format with the appropriate level of detail.

Task	Title	Total marks	Manual marking marks			Automated marking marks
			Approach	Style	Commenting	
Task 1	Points available on board	3	0.5	0.2	0.3	2
Task 2	Check word existence (simplified)	3	0.5	0.2	0.3	2
Task 3	Check word existence	4	1.5	0.2	0.3	2
Task 4	Making a move	5	1.5	0.2	0.3	3
Bonus	Optimal gameplay	2 (BONUS)				2

Total: 6 marks

Past Project Investigation

1. For this part of the tutorial you'll be putting yourself into your tutor's shoes and "marking" some solutions. The solutions are to the problem:

   > Write a function `get_species_richness()` that calculates the species richness of a habitat, based on a series of observations of various bird species. The function takes one argument: `observed_list`, a list of independent observations of bird species. The function should return a tuple consisting of:
   >
   > - the species richness, calculated as the number of different species observed
   > - an alphabetically sorted list of the species that were observed

   (a) Is the author's approach sensible? Did they over-complicate it with unnecessary or convoluted parts? Did they over-simplify it by taking shortcuts that don't work or forgetting requirements?

   (b) Has the author followed PEP8 guidelines? Did they use descriptive variable names? Are there any "magic numbers" or other unexplained literals?

   (c) How is the commenting in the solution? Did the author include a docstring?

(a) Approach **(b) Style** **(c) Commenting**

(a) Is the author's approach sensible? Did they over-complicate it with unnecessary or convoluted parts? Did they over-simplify it by taking shortcuts that don't work or forgetting requirements?

(b) Has the author followed PEP8 guidelines? Did they use descriptive variable names? Are there any "magic numbers" or other unexplained literals?

(c) How is the commenting in the solution? Did the author include a docstring?

Solution 1:

```
def get_species_richness(observed_list):
    # easy approach is to convert to a set
    species_observed = set(observed_list)
    return len(species_observed), sorted(species_observed)
```

Answer:

Solution 1

(a) Approach-wise, this solution is excellent. The author has used suitable data structures and built-in functions to simplify their approach and write elegant code. Since this solution does everything that is asked for in the instructions, even though it looks very short they hasn't over-simplified anything.

(b) This solution follows the PEP8 guidelines and has sensibly named variables. There are no magic numbers or unexplained literals.

(c) The commenting is what lets this solution down. The single comment here doesn't add anything. A better comment would be saying why *the* `observed_list` *was being converted to a set. The solution is missing a docstring.*

(a) Is the author's approach sensible? Did they over-complicate it with unnecessary or convoluted parts? Did they over-simplify it by taking shortcuts that don't work or forgetting requirements?

(b) Has the author followed PEP8 guidelines? Did they use descriptive variable names? Are there any "magic numbers" or other unexplained literals?

(c) How is the commenting in the solution? Did the author include a docstring?

Solution 2:

```
def get_species_richness(observed_list):
    """ Takes a list of strings representing species observed. Returns a
    tuple containing the species richness (an int) and a sorted list
    containing names of each species. """
    observed_birds = []

    # constructs unique list of observed bird species
    for bird in observed_list:
        if bird not in observed_birds:
            observed_birds.append(bird)

    # counts number of species and sorts by unicode sort order
    return (len(observed_birds), sorted(observed_birds))
```

Answer to Solution 2:

Solution 2

(a) *This solution also has an excellent approach, although it is not as short as the first. It does everything required in the instructions and is well-structured.*

(b) *This solution follows the PEP8 guidelines and has sensibly named variables. There are no magic numbers or unexplained literals.*

(c) *The commenting is excellent - they have included useful # comments that explains `why` each block of code exists in their solution. The function has a docstring which tells us the inputs, outputs and brief summary of what the function does.*

(a) Is the author's approach sensible? Did they over-complicate it with unnecessary or convoluted parts? Did they over-simplify it by taking shortcuts that don't work or forgetting requirements?

(b) Has the author followed PEP8 guidelines? Did they use descriptive variable names? Are there any "magic numbers" or other unexplained literals?

(c) How is the commenting in the solution? Did the author include a docstring?

Solution 3:

```
#returns b and sorted dictionary
def get_species_richness(l):
    #create a dictionary
    dict1 ={}
    b = 0
    # loop
    for i in range(1,len(l)+5):
        #loop
        for c in l:
            if not c in dict1:
                # increment by i
                b+=i
                dict1[c] =   0
            # increment by 1
            dict1[c]     += 1
        """return"""
        return (b,sorted(dict1.keys()))
```

Answer to Solution 3:

Solution 3

(a) *This function is over-complicated and difficult to read. The outer loop is not doing anything useful because it only runs once due to the indentation of the* `return`. *Despite this code being correct in terms of test-case correctness, the author should work on simplifying their code and making it more readable.*

(b) *This code has several PEP8 violations. The spacing around operators and commas is inconsistent (should be one space on either side of =, + and +=; and a space after commas). The variable names leave a lot to be desired... Names like* `b`, `c` *and* `dict1` *aren't descriptive enough of what they represent. The name* `l` *looks very similar to a* `1` *(integer one). There is a magic number* `5` *where its meaning is unexplained.*

(c) *The code contains several comments, however many of them are unhelpful. Comments like* `# loop`, `# create a dictionary` *and* `# increment by 1` *are basically just repeating the code and don't help the reader understand why coding decisions were made and what that piece of the code achieves. Aside from the docstring, comments in the function should use # rather than* `"""..."""`. *There should also be a space after the # (PEP8 test would warn you about this saying "E265 block comment should start with* `'# '`*"). The author has not properly included a docstring, which should be on the line immediately after the* `def` *line, use* `"""..."""` *(i.e. be a documentation* string*) and describe the inputs, outputs and a brief overview of what the function does.*

Writing your own test cases!

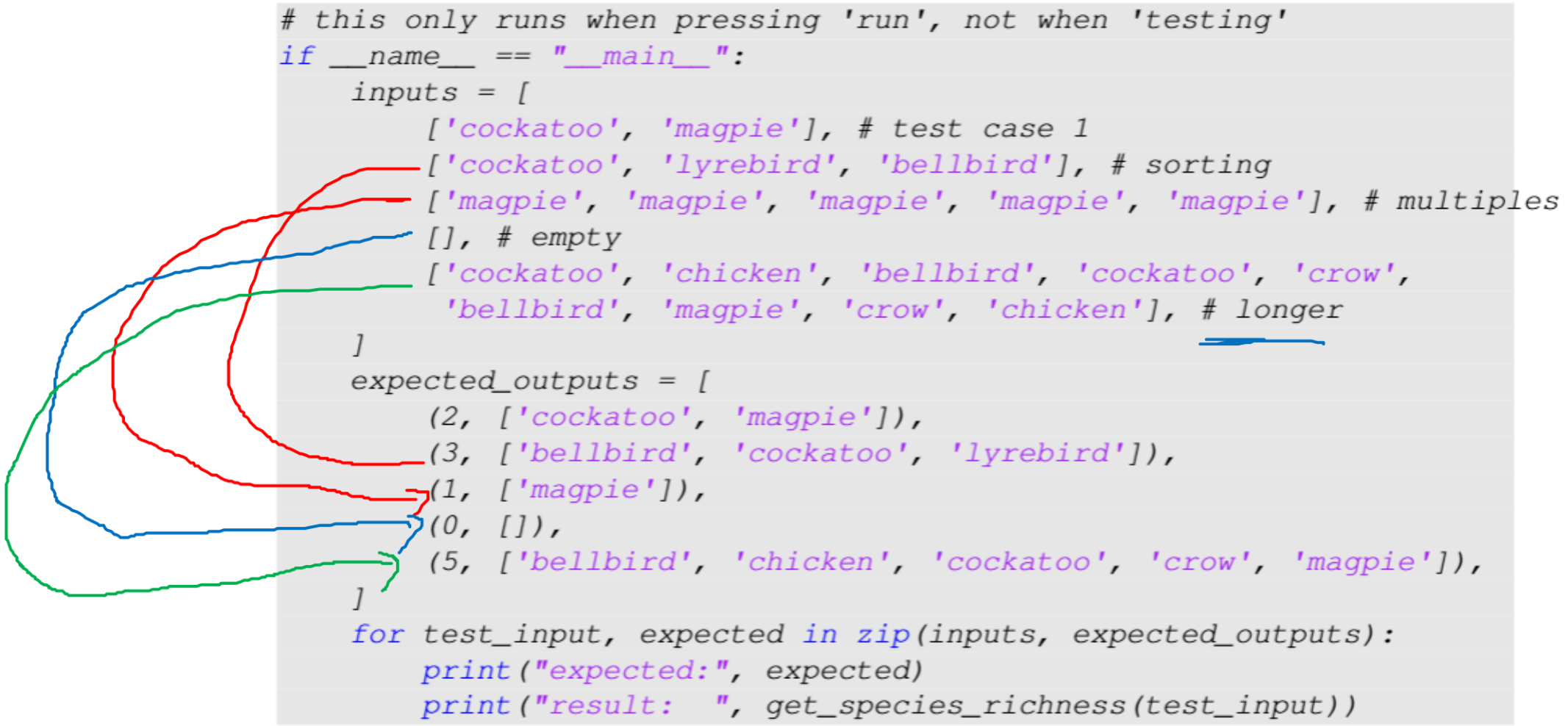
Write a function `get_species_richness()` that calculates the species richness of a habitat, based on a series of observations of various bird species. The function takes one argument: `observed_list`, a list of independent observations of bird species. The function should return a tuple consisting of:

- the species richness, calculated as the number of different species observed
- an alphabetically sorted list of the species that were observed

2. Construct three more test cases for the `get_species_richness()` problem in the test harness below. To write test cases, we should think about different possible inputs our code could receive and cover as many of them as possible. This does not mean writing a test case for every possible input, rather a test case for each category of input, especially testing any "corner cases" which are at the limits of the code's specification. We use test cases for marking correctness but testing also needs to be done by the author of the code!

Possible test cases:

A: *There are many possible tests - here are some examples:*



```
# this only runs when pressing 'run', not when 'testing'
if __name__ == "__main__":
    inputs = [
        ['cockatoo', 'magpie'], # test case 1
        ['cockatoo', 'lyrebird', 'bellbird'], # sorting
        ['magpie', 'magpie', 'magpie', 'magpie', 'magpie'], # multiples
        [], # empty
        ['cockatoo', 'chicken', 'bellbird', 'cockatoo', 'crow',
         'bellbird', 'magpie', 'crow', 'chicken'], # longer
    ]
    expected_outputs = [
        (2, ['cockatoo', 'magpie']),
        (3, ['bellbird', 'cockatoo', 'lyrebird']),
        (1, ['magpie']),
        (0, []),
        (5, ['bellbird', 'chicken', 'cockatoo', 'crow', 'magpie']),
    ]
    for test_input, expected in zip(inputs, expected_outputs):
        print("expected:", expected)
        print("result:  ", get_species_richness(test_input))
```

3. Now you're back in the student's shoes. Suppose you are working on another problem in the project and the function you are writing is becoming too long. What could you do to improve readability?

Answer:

A: *Define helper functions! A helper function is a function that performs some part of the computation of another function. Helper functions can make programs more readable by giving descriptive names to computations. By taking computations out of a function and placing them in helper functions, we can then reuse those helper functions if we ever need that computation again. This is much easier than copy-pasting parts of a larger function.*
You might also check to see if there are any redundant parts of your code or if any part is more complex than it needs to be.

4. What problems might you face if you define functions inside of other functions? Where should helper functions be defined?

Answer:

A: *Nesting function definitions makes it harder to reuse them, as the inner function only exists while the outer function is running. This also means you won't be able to unit test the inner function. It can also make your code confusing and likely to use variables that you haven't explicitly passed in to your function as an argument (non-local). Non-local variables are not great to use because they can make your code harder to understand and debug, since it's not always clear where the variable came from or when it might change.*
Define helper functions outside of other functions, usually at the top level of your file (alongside other functions). This way they can be used by multiple functions, are easier to test and your code stays clean and readable.

Exercise 1 / 3

1. Fill in the blanks with comments and a docstring for the following function, which finds the most popular animals by counting ballots. An example for `ballots` is `['dog', 'pig', 'cat', 'pig', 'dog']`, in which case the function returns `['dog', 'pig']`.

```
def favourite_animal(ballots):
    """ ..."""
    tally = {}

    # ...
    for animal in ballots:
        if animal in tally:
            tally[animal] += 1
        else:
            tally[animal] = 1

    # ...
    most_votes = max(tally.values())
    favourites = []
    for animal, votes in tally.items():
        if votes == most_votes:
            favourites.append(animal)

    return favourites
```

"""Takes a list ‘ballots‘ as input. Counts the frequency of each animal in ‘ballots‘, and returns a list of the most frequently voted animals."""

Counts frequencies of each animal in the ballots.

Find and store the animals that received the highest number of votes.

Exercise 2a / 3

2. Consider the following programs. What are the problematic aspects of their variable names and use of magic numbers? What improvements would you make to improve readability?

(a)
```
a = float(input("Enter days: "))
b = a * 24
c = b * 60
d = c * 60
print("There are", b, "hours", c, "minutes,", d, "seconds in", a, "days")
```

magic numbers

Answer:

A:

```
HOUR_DAY = 24
MINUTE_HOUR = 60
SECOND_MINUTE = 60

days = float(input("Enter days: "))
hours = days * HOUR_DAY
minutes = hours * MINUTE_HOUR
seconds = minutes * SECOND_MINUTE
print("There are", hours, "hours", minutes,
      "minutes", seconds, "seconds in", days, "days")
```

Using constants for the conversion multipliers and appropriate variable names will make this code much more easy to read.

Exercise 2b / 3

2. Consider the following programs. What are the problematic aspects of their variable names and use of magic numbers? What improvements would you make to improve readability?

(b)
```
word = input("Enter text: ")
x = 0
vowels = 0
word_2 = word.split()
for word_3 in word_2:
    x += 1
    for word_4 in word_3:
        if word_4.lower() in "aeiou":
            vowels += 1
if vowels/x > 0.4:
    print("Above threshold")
```

Answer to Exercise 2b:

A:

```
THRESHOLD = 0.4

text = input("Enter text: ")
n_words = 0
n_vowels = 0
words = text.split()
for word in words:
    n_words += 1
    for letter in word:
        if letter.lower() in "aeiou":
            n_vowels += 1
if n_vowels/n_words > THRESHOLD:
    print("Above threshold")
```

Rather than a series of numbered variable names with `word` *in them, we've named the variables more accurately according to what they store, from* `text` *to* `word` *and* `letter`*. Matching the plurality of nouns is a good idea, such as naming a list of words* `words` *while referring to a single word (in the for loop) as* `word`*. Reassigning to* `letter` *when converting case is better in this situation than creating a new variable, because we never use the original casing after that. The prefix of* `n_` *to variables which count the number of something is useful as it indicates the difference between, for example, a collection of words and a number representing an amount of words. A constant* `THRESHOLD` *has been used in place of a magic number at the end of the code.*

Exercise 3 / 3

input :

nums = [-1, -2, -3]

output : [-2]

3. The function below is supposed to take a list of integers and remove the negative integers from the list, however, it is not working as intended.

   - Write down three test cases that could be useful for function verification or finding bugs.
   - Debug the associated code snippet to solve the problem.

```
def remove_negative(nums):
    for num in nums:
        if num < 0:
            nums.remove(num)
```

return nums

Sample Answer to Exercise 3:

A: *Test cases to consider include: The empty list* `[]`*, a list with no negative numbers* `[0, 1, 2]` *and a list with only negative numbers* `[-1, -2, -3]`.
The debugging process will be different for everyone, but here is an example:
Begin by observing the function's failure to process the following test case.

```
lst = [-1, -2, 3]
remove_negative(lst)
print(lst)
```

Include a print statement to observe the values of the variables within the for loop.

```
def remove_negative(nums):
    for num in nums:
        print(num, nums)
        if num < 0:
            nums.remove(num)
```

Repeating the above test case, one will find that `num` *takes the value* -1 *and* 3*, but skips* -2 *entirely. With any luck, this will lead to the recollection/realisation that it is dangerous to remove elements from list whilst iterating over them, as Python can skip elements. Instead, the following solution may be attained:*

```
def remove_negative(nums):
    to_remove = []
    for num in nums:
        if num < 0:
            to_remove.append(num)

    for num in to_remove:
        nums.remove(num)
```

Programming on Paper

Question 1 / 2

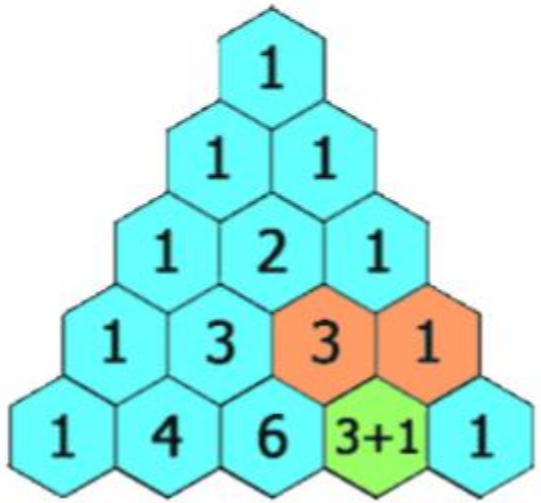
1. Write a function `check_parens()` that takes a string `text` and checks that the parentheses are valid (i.e. after opening, at some point in the text the parenthesis is closed). For example, `check_parens("(()()")` should return `True` and `check_parens("())")` should return `False`.

Answer:

```
def check_parens(text):
    """ Takes a string `text` and returns a bool indicating whether
    or not the parentheses in `text` are balanced, i.e. any opened
    parens are closed, no closing parens without being opened. """

    n_open = 0
    for char in text:

        # opening parens
        if char == '(':
            n_open += 1

        # closing parens
        elif char == ')':
            n_open -= 1

            # invalid if a parens is closed before being opened
            if n_open < 0:
                return False

    # any open parens has been closed <=> number open at end is 0
    return n_open == 0
```

Question 2 / 2

2. Challenge: Write a function `gen_pascal()` that takes an integer `num_rows` (≥ 1) and returns the first `num_rows` rows of Pascal's triangle as a list of lists of ints. In Pascal's triangle, each number is the sum of the two numbers directly above it. For example, `gen_pascal(5)` should output
`[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]`

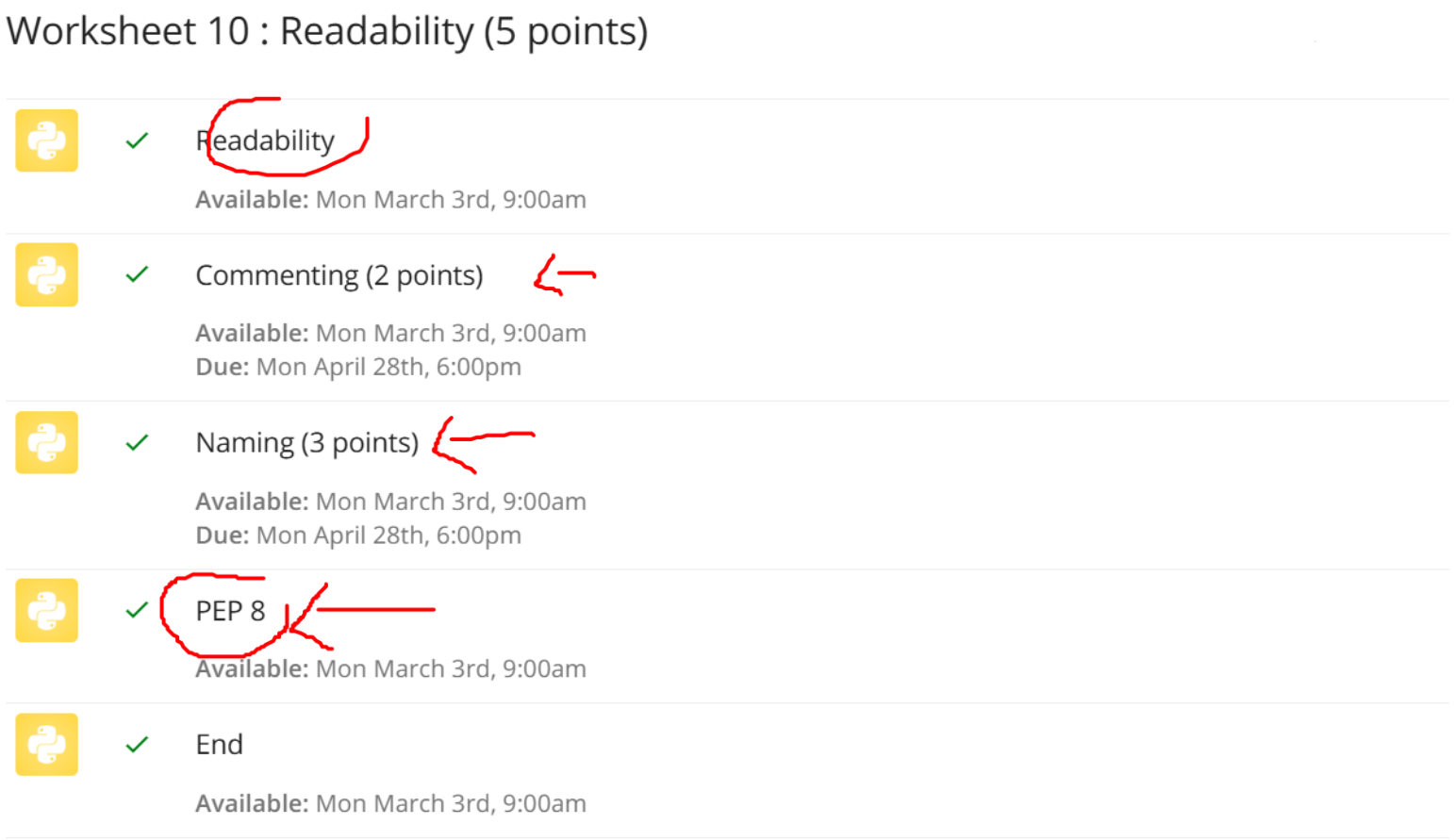


Answer to Question 2:

A:

```
def gen_pascal(num_rows):
    """ Takes an int `num_rows` and generates the pascal's triangle
    up to the given number of rows. Returns the triangle as a list of
    list of ints. """

    # initialise the first row of pascals triangle
    triangle = [[1]]

    # construct additional rows
    for i in range(1, num_rows):
        prev_line = triangle[i - 1]
        new_line = [1]

        # add numbers from above line
        for j in range(1, i):
            new_line.append(prev_line[j - 1] + prev_line[j])

        # final rightmost 1 and finished this line of triangle
        new_line.append(1)
        triangle.append(new_line)

    return triangle
```

Today's tutorial covers worksheets:



And also:

Worksheet 11a: Debugging exercises (FOR DEBUGGING PRACTICE, NOT FOR MARKS)

✓ Bugs

Available: Mon March 3rd, 9:00am

✓ Testing & Debugging

Available: Mon March 3rd, 9:00am

✓ Problems

Available: Mon March 3rd, 9:00am

Independent Work

- **MST is on Tuesday (15 April), with 2 sessions:**
  - **11am to 12pm**
  - **12pm to 1pm**
- **Next due dates:**
  - Ed Worksheets **10** and **11** is **due Monday, April 28th, 6pm**.
  - Your **Project 1** is **due Friday, May 2nd, 6pm**.
- **Raise your hand** if you have any questions!

Scan here for annotated slides