

cheat_sheet

November 10, 2022

1 Cheat Sheet

1.1 Week 0

1.1.1 Print function:

```
[1]: variable_to_print = 7

print(variable_to_print)

print("string to print")

print("First line \nSecond line")
```

```
7
string to print
First line
Second line
```

1.1.2 Arithmetic operators:

```
[4]: 1+2
      1-2
      1/2
      13*2
```

```
[4]: 26
```

1.1.3 Assigning variables:

```
[3]: close_to_pi = 355/113
      print(close_to_pi)

      x = 2
      y = 3
      z = x * y
      print(z)
```

```
3.1415929203539825
6
```

1.1.4 Data Type: Strings

A string is a series of characters bounded by ‘single’ or “double” quotations, used interchangeably. If you are using an apostrophe or single quote inside a string, you can bound it by double quotations to still identify it as a string.

```
[5]: my_string = 'several characters'
     my_other_string = "example"
     error_string = 'grandma's recipes'
     fixed_string = "grandma's recipes"
```

```
Input In [5]
      error_string = 'grandma's recipes'
                        ^
SyntaxError: invalid syntax
```

You can use some arithmetic operators on strings:

```
[10]: oligo = "ATG"
      print(oligo*3)
      print(oligo + oligo)
```

```
ATGATGATG
ATGATG
A
```

1.2 Week 1

1.2.1 Useful UNIX commands:

- pwd = print working directory (where am I now?)
- ls = list (what’s in this directory?)
 - ls -lah = list all files in human-readable form
- cd (destination) = change directory to an absolute or relative destination
 - cd ~ = go to user’s home directory
 - cd .. = move up one directory
 - ./ = ‘here’
- cp (file to be copied) (destination) = copy file to a destination
 - cp -r (directory to be copied) (destination) = copy directory to a destination
- mkdir (name of new directory) = make a new directory inside your current directory
- mv (file to be moved) (destination) = move file to new destination
 - mv -r (directory to be moved) (destination) = move directory to a destination
 - Renaming a file or directory: mv the file or directory to its current destination

- `rm` (file to be removed) = delete a file
 - `rm -r` (directory to be removed) = delete a directory
 - THIS IS FOREVER, BE SURE YOU WANT TO `rm` WHATEVER YOU ARE `rm`-ING!!
- `nano` (filename) - opens nano text editor on the file specified. If file doesn't exist, nano will create it
 - Use shortcut commands listed at bottom of nano screen to save file or exit
- `*` = wildcard, represents any number of any characters
 - * `*.txt` = any file ending with `.txt`
 - * `abc*` = any file that starts with `abc`
 - * `*7` = any file that has a seven somewhere in the middle
- `?` = represents any one character
 - `file?.txt` = any file that starts with "file", has one character of any type, and ends with ".txt"
- `wc` = word count
 - `wc -l` = count number of lines
 - check out `wc -h` for all available options
- `grep` = search a file for a pattern
 - `grep` is very detailed, check out `grep -h` for usage
- `|` = pipe, take the output of the last operation and use it as the input for the next operation
 - `grep "lily" flowers.txt | wc -l` = searches for occurrences of "lily" in the `flowers.txt` file and returns the number of lines

1.3 Week 2

1.3.1 Data types:

```
[12]: my_string = 'string'
      my_int = 7
      my_float = 7.0
      my_list = [7, "petal", "sepal", 5.8]
```

```
print(type(my_string))
print(type(my_int))
print(type(my_float))
print(type(my_list))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'list'>
```

1.3.2 Numpy arrays:

```
[14]: import numpy as np
      array = np.array([1,1,2,3,5,8])
      print(array)
      print(type(array))
```

```
[1 1 2 3 5 8]
```

```
<class 'numpy.ndarray'>
```

You can apply operations across an array:

```
[15]: print(array + 1)
```

```
[2 2 3 4 6 9]
```

1.3.3 Indexing:

Use square brackets [] after a list or array name to indicate which value in the list or array you want to indicate:

```
[53]: array[0]
```

```
[53]: 1
```

```
[55]: array_2D = np.array([[1,1,2,3,5,8],[4,6,6,1,9,8]])
      array_2D[1,4]
```

```
[55]: 9
```

```
[56]: my_list[2]
```

```
[56]: 'sepal'
```

Use negatives to index off the end of the list or array:

```
[57]: array[-1]
```

```
[57]: 8
```

1.3.4 Slicing:

Use a colon [x:y] to indicate “from x up to but not including y” Use two colons [x:y:z] to indicate “from x up to but not including y, counting by z” Omit the x and/or y values to indicate the beginning or end: - [:y] = “from the beginning up to but not including y” - [x:] = “from x to the end” - [:] = “the whole thing” - [::-1] = “the whole thing in reverse (counting by -1)”

```
[15]: import numpy as np
      array = np.array([1,1,2,3,5,8])
      array_slice = array[1:5]
      print(array_slice)
```

```
print(array[0:6:2]) # prints every other value
print(array[::-1]) # prints the array in reverse
```

```
[1 2 3 5]
[1 2 5]
[8 5 3 2 1 1]
```

2D arrays are sliced just as you'd indicate a Cartesian position with (x,y)

```
[24]: import numpy as np
array_2D = np.array([[1,1,2,3,5,8],[4,6,6,1,9,8]])
print("ex1:", array_2D[1,2:5]) # Row [1], Columns [2:5]
print("ex2:", array_2D[:, :2]) # All rows, the first two columns
```

```
ex1: [6 1 9]
ex2: [[1 1]
      [4 6]]
```

1.4 Week 3

- Read in data from a .csv file (comma separated variables)

```
import numpy as np data = np.loadtxt('path_to_csv/csv_name.csv', delimiter = ',')
```

1.4.1 Generate heatmap:

```
import matplotlib.pyplot as plt plt.imshow(data) plt.colorbar() plt.ylabel('y
label') plt.xlabel('x label')
```

1.4.2 np.arange(x,y,z)

Produces a range from value x up to but not including y, counting by z

```
[8]: import numpy as np
print(np.arange(2,15))
print(np.arange(0,10,2))
print(np.arange(5,1,-1))
```

```
[ 2  3  4  5  6  7  8  9 10 11 12 13 14]
[0 2 4 6 8]
[5 4 3 2]
```

1.4.3 np.mean()

Takes the mean of an array or across the rows or column `np.mean(array)` returns the mean of all values in the array `np.mean(array, axis = 0)` takes the mean of each column `np.mean(array, axis = 1)` takes the mean of each row

Note: don't forget to check the shape of your output array to be sure you took the mean of the dimension you wanted.

`np.max()`, `np.min()`, and `np.sum()` function in the same way

1.4.4 np.zeros(n), np.ones(n)

Create an array of zeros or ones of length n

```
[25]: import numpy as np

print(np.zeros(7))
print(np.ones(5))
print(np.ones(5)+3)
```

```
[0. 0. 0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1.]
[4. 4. 4. 4. 4.]
```

1.4.5 Plotting: Scatter plots, etc.

- Create a scatter plot: plt.scatter(x, y, color = , marker = , etc.)

1.4.6 For-loops

Structure: for <variable> in <iterable object>: do action

Anything indented will be contained in the loop

```
[27]: odds=[1,3,5,7]    # Define list (iterable)
for num in odds:        # For each variable (called num) in odds, do:
    print(num)          # Action (print the variable)
```

```
1
3
5
7
```

```
[28]: odds=[1,3,5,7]
for anything in odds:   # the iterable variable can be called anything you want
    print(anything)
```

```
1
3
5
7
```

```
[29]: odds=[1,3,5,7]
for num in odds:
    print('loop')       # The action doesn't need to necessarily use the variable.
                        # in this case, the action (print('loop')) is performed as
                        ↪ many times as there are variables in the list
```

```
loop
loop
loop
loop
```

1.4.7 Appending to lists:

```
[39]: odds=[1,3,5,7]
      more_odds = odds + [9]
      print(more_odds)
```

```
[1, 3, 5, 7, 9]
```

```
[40]: more_odds.append(11) # Note you don't have to assign more_odds.append() to a
      ↪new variable, more_odds is automatically updated with .append()
      print(more_odds)      # In fact, if you run this cell a few more times, it will
      ↪continue to append
```

```
[1, 3, 5, 7, 9, 11]
```

1.4.8 Creating a counter for a for-loop:

```
[42]: counter = 0
      for i in range(0,5):
          print('loop ', counter)
          counter += 1          # This notation += is shorthand for "counter =
      ↪counter + 1"
```

```
loop 0
loop 1
loop 2
loop 3
loop 4
```

1.4.9 enumerate():

enumerate() automatically creates a counter by generating an 'enumerate object' of the format [(0, item[0]), etc.]

```
[45]: my_list = ['alpha', 'bravo', 'charlie', 'delta', 'echo']
      print(list(enumerate(my_list)))
```

```
[(0, 'alpha'), (1, 'bravo'), (2, 'charlie'), (3, 'delta'), (4, 'echo')]
```

You can assign the counter value and the list value to two variables so that they are both callable in the for-loop:

```
[47]: for counter, list_value in enumerate(my_list):
      print(counter)
      print(list_value)
```

```
0
alpha
1
bravo
```

```
2
charlie
3
delta
4
echo
```

1.5 Week 4

1.5.1 Defining Functions

```
[4]: def my_function(input_string):
      print('This is the function printing:', input_string)
      return input_string + ': this is the returned value'

      print(my_function('A string'))
```

```
This is the function printing: A string
A string: this is the returned value
```

The function above has these parts: - definition statement: `def` + the name of the function + any number of arguments/parameters taken in by the function - The arguments/parameters are the variables that will be used by the function, and make it so that you can run the function with different input values - the action: in this case it is a simple `print` call, but you can make the action as complex as you need - the return statement: this identifies the value that will be output by the function - any variables defined within a function will not be available outside the function - only the returned variable will be output.

Here's a more complex example:

```
[5]: def my_other_function(x, y, list_of_ints):
      num_list = []
      for i in list_of_ints:
          num = x + y*i
          num_list.append(num)
      return num_list

      output_value = my_other_function(2, 3, [4,5,6])
      print(output_value)
```

```
[14, 17, 20]
```

- This function takes in three arguments: `x`, a number, `y`, another number, and `list_of_ints`, a list of integers.
- an empty list, `num_list`, is defined, which will be a place to store the values the function generates
- a for-loop takes each value in the `list_of_ints` and includes them in the equation `x + y*(current_variable_in_list_of_ints)`, and assigns that value to the variable `num`

- num is added to the list `num_list`
- the function returns `num_list` so that when the function is called and assigned to `output_value`, the variable `output_value` now contains the returned `num_list` from `my_other_function`

1.5.2 Nested for-loops

If you write a for-loop within another for-loop, the interior for-loop will run fully each time the exterior for-loop iterates:

```
[19]: my_list = ['alpha', 'bravo', 'charlie']
      for list_val in my_list:
          print(list_val)
```

```
alpha
bravo
charlie
```

```
[20]: my_list = ['alpha', 'bravo', 'charlie']
      for list_val in my_list:
          print(list_val)
          for letter in list_val:
              print(letter)
```

```
alpha
a
l
p
h
a
bravo
b
r
a
v
o
charlie
c
h
a
r
l
i
e
```

1.5.3 Logical Operators

- `==` (is equal), `!=` (is not equal), `<`, `>=`, `<=`, `>`
- These operators can be used to compare various objects

- The output of a logical operation is a Boolean value (True or False)

```
[22]: print(5>3)
      print(5<3)
```

```
True
False
```

```
[30]: string1 = 'alpha'
      string2 = 'kilo'
      string3 = 'alpha'
      string4 = 'alpaca'

      print(string1 < string2) # when > or < are used to compare strings, they
                               ↪ evaluate alphabetical order

      print(string3 < string4)

      print(string1 == string3)

      print(string1 != string2)
```

```
True
False
True
True
```

- Using a logical operator to compare two arrays will return another array of the same shape containing Boolean values.
 - Each element of the first array will be compared to the corresponding element of the second array

```
[32]: import numpy as np
      array1 = np.array([1,1,2,3,5,8])
      array2 = np.array([4,6,6,1,9,8])

      comparison_array = array1 > array2
      comparison_array
```

```
[32]: array([False, False, False,  True, False, False])
```

```
[34]: array_2D_1 = np.array([[1,1,2,3,5,8],[4,6,6,1,9,8]])
      array_2D_2 = np.array([[5,4,2,7,3,4],[5,3,1,2,9,8]])

      comparison_array_2D = array_2D_1 == array_2D_2
      comparison_array_2D
```

```
[34]: array([[False, False,  True, False, False, False],
             [False, False, False, False,  True,  True]])
```

By applying a Boolean array to another array, you select for True values:

```
[40]: array_2D_1 = np.array([[1,1,2,3,5,8],[4,6,6,1,9,8]])
      array_2D_2 = np.array([[5,4,2,7,3,4],[5,3,1,2,9,8]])

      comparison_array_2D = array_2D_1 == array_2D_2

      output_array = array_2D_1[comparison_array_2D]

      print("Original:\n",array_2D_1)
      print("Comparison:\n",comparison_array_2D)
      print("Output:\n",output_array) # This 'applies' the Boolean array to the
      ↪original array, returning only the
      ↪values that correspond to "True" in
      ↪comparison_array_2D
```

```
Original:
[[1 1 2 3 5 8]
 [4 6 6 1 9 8]]
Comparison:
[[False False  True False False False]
 [False False False False  True  True]]
Output:
[2 9 8]
```

1.5.4 If/else-statements:

Use logical operators inside an if-statement to perform an action if a condition is met:

```
[41]: x = 7
      if x > 5:
          print("X is greater than five")
```

X is greater than five

Add an else condition to specify what happens if the condition is not met:

```
[42]: x = 3
      if x > 5:
          print("X is greater than five")
      else:
          print("X is not greater than five")
```

X is not greater than five

1.5.5 in statements:

- Evaluate whether an element is in an object

```
[46]: letter = 'a'
string_list = ['alpha', 'kilo']

for string in string_list:
    if letter in string:
        print(string, "contains an", letter)
    else:
        print(string, "has no", letter)
```

alpha contains an a
kilo has no a

```
[48]: num_list = [0,4,6,1,2]
print(4 in num_list)
print(3 in num_list)
```

True
False

1.6 Week 5

- Define a figure and axes variables: `my_fig = plt.figure()` `ax = plt.axes()`
`ax.plot(x,y)` `ax.set_xlabel('horizontal label')` `ax.set_ylabel('vertical label')`
`ax.set_title('my title')` `ax.set_xlim(0, 100)` `ax.axvline(2)`
`ax.axhline(6,color='r')`

Defining a figure with `plt.figure()` allows you to work on multiple figures at one time, each assigned to a different variable, i.e.:

```
[21]: import matplotlib.pyplot as plt

fig1 = plt.figure()
fig2 = plt.figure()
fig3 = plt.figure()
```

<Figure size 640x480 with 0 Axes>

<Figure size 640x480 with 0 Axes>

<Figure size 640x480 with 0 Axes>

1.6.1 np.diff()

`np.diff` calculates the differences between adjacent values in an array:

```
[18]: output = np.diff(np.array([1,1,2,3,5,8,13]))
output
```

```
[18]: array([0, 1, 1, 2, 3, 5])
```

(1-1 = 0, 2-1 = 1, 3-2 = 1, etc.)

Notice how many values are in each array above.

1.6.2 Plotting labels & legends:

When you call a plotting function, if you define a label for your line, bar, etc., and call the `plt.legend()` function, the legend will automatically populate with the labels for each line:

```
plt.plot(x,y, label = "my_line") plt.legend()
```

This is particularly useful when plotting many lines. If you have a list of your line names, you can iterate through them with a for-loop:

```
line_names = ["line1", "line2", "line3"]
for i in lines:
    plt.plot(lines[i], label = line_names[i])
```

1.6.3 While Loops:

Format: while <condition is met>: do function

```
[30]: i = 0
      while i < 7:
          i = i+1
          print(i)
```

```
1
2
3
4
5
6
7
```

1.7 Week 6:

Subsetting arrays with conditionals:

```
[38]: #Define an array
      arr = np.array([[2,4,1,7],[9,4,7,6],[1,2,1,5],[4,8,8,1]])
      print(arr, "\n")

      #Make a selection based on a conditional (values greater than 5)
      conditional_selection = arr > 5
      print(conditional_selection, "\n")

      #Subset the original array with the selection array
      subset_arr = arr[conditional_selection]
      print(subset_arr) # This returns an array of all the values from the original_
      ↳ array that are greater than 5
```

```
[[2 4 1 7]
 [9 4 7 6]]
```

```
[1 2 1 5]
[4 8 8 1]]
```

```
[[False False False  True]
 [ True False  True  True]
 [False False False False]
 [False  True  True False]]
```

```
[7 9 7 6 8 8]
```

1.7.1 np.argmax(), np.argmin(), np.argsort()

```
[47]: np.argmax(arr) # Returns the index of the max value in an array (reads a 2D
      ↪array like 1D),
      #i.e. the 4th index is the first value of the second row
```

```
[47]: 4
```

```
[58]: np.argmax(arr, axis = 1) # gives the index of the max value for each row
```

```
[58]: array([3, 0, 3, 1])
```

```
[42]: np.argmax(arr, axis = 0) # gives the index of the max value for each column
```

```
[42]: array([1, 3, 3, 0])
```

np.argmin() behaves the same, but finds the index of the min

np.argsort() returns the indexes of the array if it were to be sorted:

```
[50]: print(arr[0])
      print(np.argsort(arr[0]))
```

```
[2 4 1 7]
[2 0 1 3]
```

Read this as: the sorted order of the first row is the [2] value, then the [0] value, then the [1] value, then the [3] value.

A for-loop to demonstrate:

```
[52]: for i in np.argsort(arr[0]):
      print(arr[0,i])
```

```
1
2
4
7
```

1.7.2 Creating subplots:

- Defining figure and figure size `fig=plt.figure(figsize=(10,3))`
- Making subplots:

```
fig,ax=plt.subplots()
fig, (ax1,ax2)=plt.subplots(1,2)
fig.savefig('myfig.pdf')
```

- [Methods of the axis class:](#)

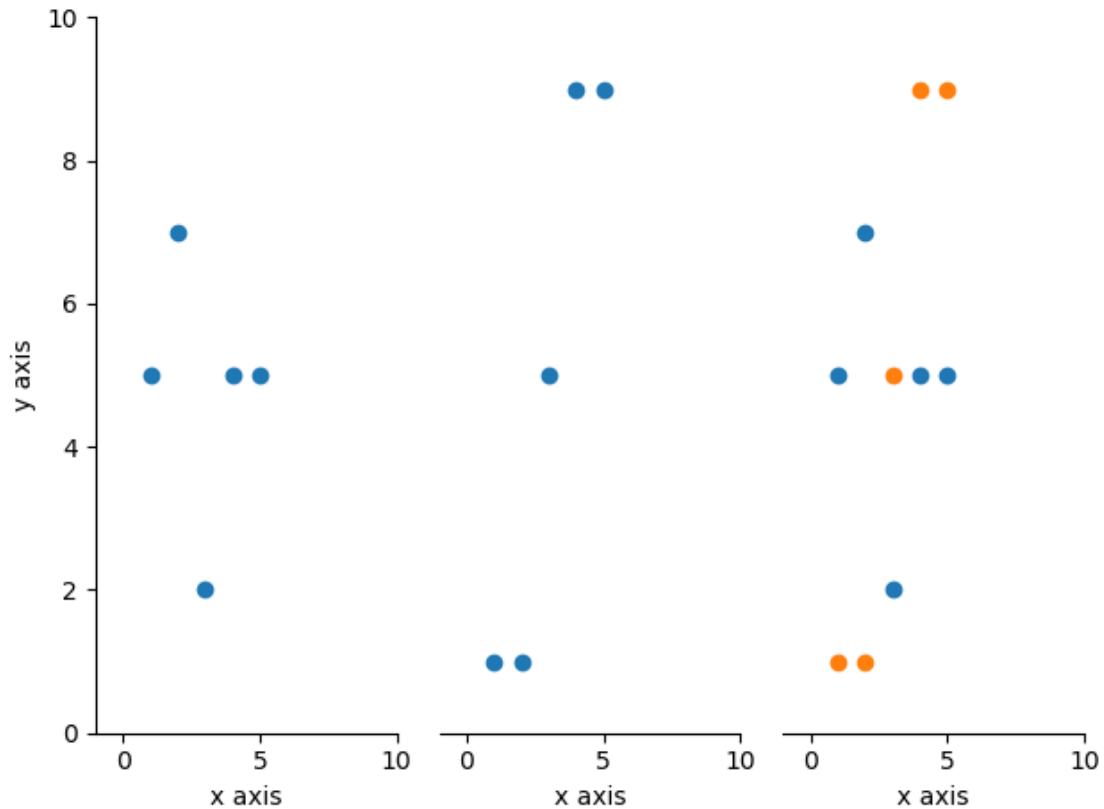
```
ax.xticks
ax.yticks
ax.set_ylim
ax.set_xlim
ax.xticklabels
ax.yticklabels
ax.text
ax.spines
ax.set_aspect
```

- Useful reference: [Matplotlib Gallery examples](#) For instance, [subplots example](#)

```
[53]: import numpy as np
import matplotlib.pyplot as plt

arr1 = np.array([[1,2,3,4,5],[5,7,2,5,5]])
arr2 = np.array([[1,2,3,4,5],[1,1,5,9,9]])

fig, ax = plt.subplots(1,3) # Defines a new figure and axes, gives dimensions
    ↳for subplot grid: 1 row, 2 columns
ax[0].scatter(arr1[0],arr1[1])
ax[1].scatter(arr2[0],arr2[1])
ax[2].scatter(arr1[0],arr1[1])
ax[2].scatter(arr2[0],arr2[1])
ax[0].set_ylabel('y axis')
for axis in ax:
    axis.set_xlim(-1,10)
    axis.set_ylim(0,10)
    axis.set_xlabel('x axis')
    axis.spines['top'].set_visible(False)
    axis.spines['right'].set_visible(False)
for i in range(1,3):
    ax[i].spines['top'].set_visible(False)
    ax[i].spines['right'].set_visible(False)
    ax[i].spines['left'].set_visible(False)
    ax[i].set_yticks([])
fig.tight_layout() # Generic statement for keeping labels and axes from
    ↳overlapping
```



1.7.3 Dictionaries:

Creating a dictionary

```
[8]: new_dict = {} # Start with an empty dict and add key-value pairs to it
      new_dict['key1'] = 'value1' # Call a value by 'indexing' the dictionary by its
      ↪key
      new_dict
```

```
[8]: {'key1': 'value1'}
```

```
[10]: # Read in a dict from lists
      key_list = ['key1', 'key2']
      value_list = ['value1', 'value2']

      new_dict = {}
      for i in range(0, len(key_list)):
          new_dict[key_list[i]] = value_list[i]

      new_dict
```



```
[10]: {'key1': 'value1', 'key2': 'value2'}
```

```
[13]: # Zip two lists into a dict:
key_list = ['key1', 'key2']
value_list = ['value1', 'value2']

new_dict = dict(zip(key_list, value_list))
print(new_dict)

# Return a dictionary's keys
print(new_dict.keys())

# Return a dictionary's values
print(new_dict.values())
```

```
{'key1': 'value1', 'key2': 'value2'}
dict_keys(['key1', 'key2'])
dict_values(['value1', 'value2'])
```

1.7.4 Complex dictionaries:

Dictionary values can be complex objects:

```
[51]: # Dictionary of lists
my_dict = {'plants': ['maple', 'pine', 'snowberry'], 'fungi': ['amanita',
↳ 'morel']}
print(my_dict)
print(my_dict['plants'][2]) # Index a complex dictionary with a series of [ ]
↳ brackets
```

```
{'plants': ['maple', 'pine', 'snowberry'], 'fungi': ['amanita', 'morel']}
snowberry
```

```
[30]: # Dictionary of dictionaries
my_dict = {'plants': {'angiosperm': 'maple', 'confier': 'pine'}, 'fungi':
↳ {'basidio': 'amanita', 'asco': 'morchella'}}
print(my_dict)
print(my_dict['fungi']['asco'])
```

```
{'plants': {'angiosperm': 'maple', 'confier': 'pine'}, 'fungi': {'basidio':
'amanita', 'asco': 'morchella'}}
morchella
```

```
[52]: # Iterate through a dictionary
my_dict = {'plants': {'angiosperm': 'maple', 'confier': 'pine'}, 'fungi':
↳ {'basidio': 'amanita', 'asco': 'morchella'}}

for key in my_dict:
```

```
print("Level 1:", key)
for x in my_dict[key]:
    print("Level 2:", x)
    print("Level 3:", my_dict[key][x])
```

```
Level 1: plants
Level 2: angiosperm
Level 3: maple
Level 2: confier
Level 3: pine
Level 1: fungi
Level 2: basidio
Level 3: amanita
Level 2: asco
Level 3: morchella
```

```
[ ]:
```