

A decorative network graph pattern in the top-left corner, featuring a complex web of interconnected nodes and edges. Some nodes are highlighted with blue circles, and others with solid blue dots.

Lecture 4.1

Lists

A decorative network graph pattern in the bottom-right corner, featuring a complex web of interconnected nodes and edges. Some nodes are highlighted with blue circles, and others with solid blue dots.

Topics 4.1 and 4.2 Intended Learning Outcomes

- ◎ By the end of the week you should be able to:
 - Create and manipulate lists,
 - Understand the difference between sets and lists,
 - Create dictionaries for storing key/value pairs, and
 - Understand what kinds of problems can be solved using lists, sets, and dictionaries.

Introduction

- ◎ The **sequences** we've worked with so far have had very **specific** purposes.
 - A **string** is a **sequence** of **characters**.
 - `r = "CSE4IP"`
 - A **range** is a counting **sequence** of **integers**.
 - `r = range(5)`
- ◎ What if we want something more **customised**, like a **sequence** of *temperatures* or *customers*?

Introduction

Q1: Write a Python program to calculate the average of 4 numbers.

Answer:

Python Code: Q1 Calculate the average of 4 numbers

```
num1 = float(input("Enter number 1: "))
num2 = float(input("Enter number 2: "))
num3 = float(input("Enter number 3: "))
num4 = float(input("Enter number 4: "))
print("Numbers entered:", num1, num2, num3, num4)
average = (num1 + num2 + num3 + num4) / 4
print(f"The average of 4 numbers is {average}")
```

Q2: Write a Python program to calculate average of 25 number.

Answer: If we use the above code as a template, we need to add **21** additional variables. Obviously, the overall length of the program will grow. However, averaging **100 variables** or more is impractical. To solve this issue, we can use the **for-loop** statement as follows:

Python Code: Q2 Calculate the average of 25 numbers

```
sum=0
for i in range (25):
    num = float(input("Enter number " + str(i)+ ": "))
    sum = sum + num
print (f"The average of 25 numbers is {sum/25}")
```

As can be seen, **Q2** python code can be easily modified to calculate the average of 100 variables or even 1000 variables - just change the value of **range ()** function. **However**, unlike **Q1**, **we can not display the values of the entered numbers.**

Introduction

How about the following questions?

- **Q1:** Write a Python program to read the IDs and marks of 100 students. Assign grades to all students based on their marks. *How many variables do we need?*
- **Q2:** Write a Python program to read the temperature at morning, noon, and evening for August days. Display the highest and lowest temperatures. *How many variables do we need?*
- **Q3:** Write a Python program to take customer orders from 8:00 a.m. to 5:00 p.m. Send all orders into item sections for dispatching process. *How many variables do we need?*
- **Q4:** Write a Python program to read a number, string, or file. Perform the following operations: put them in order, re-display them, and use the entered values to test some conditions or different operators.

In all of these situations, we must KEEP the values of all variables for future manipulation. To save variable values for future use, we can use Python **data structures**.

Introduction

Data Structure

Data structures can be defined as containers that can be used to store and organise variable values to manipulate the available data.

Python **variable** stores only one value, whereas Python **data structure** stores more than **one values**.

- ◎ Python data structures are **lists**, **sets** and **dictionaries**.
- ◎ In this lecture we will learn how to **build** our own **sequences** using **lists**.

Lecture Overview

1. Basics of Lists
2. Working with Lists
3. Lists and References

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and edges. The nodes are represented by small circles, some of which are larger and have concentric circles, while others are smaller and solid. The edges are thin lines connecting these nodes, creating a dense, organic structure.

Basics of Lists

List

List

Lists are dynamic data structure that store sequence of items. Items data can be homogeneous or heterogeneous.

Lists are **mutable** so items can be added to or removed from them. We can also use indexing, slicing, and various methods to work with lists.

How can we create a list?

Lists can be created using square brackets `[]` or `list()` constructor. Items inside list are separated by commas.

```
l = list() # empty list using list()
print(l)
>>> []
l = list((1, 2, 3)) # Using list constructor
print(l)
>> [1, 2, 3]
l = [] # empty list using []
print(l)
```

```
l = [1, 2, 3] # homogeneous list using []
print(l)
>>> [1, 2, 3]

# heterogeneous items
l = [1.0, 'APG', 3]
print(l)
>>> [1.0, 'APG', 3]
```

List

List: Index

List index: List index starts at 0. We can also use negative index.

```
l = [1, 2, 3, 4, 5, 6]
```

```
>>> print (type(l))
```

```
>>> <class list>
```

L=	1	2	3	4	5
Index:	0	1	2	3	4
Negative index:	-5	-4	-3	-2	-1

We can also use `input()` function to create a list of items.

```
l = input ("Enter list : ")
>>> Enter list: [1, 2, 3]
print (l)
>>> [1, 2, 3]
print (type(l))
>>> <class 'str'>
```

Printing lists: We can use the `print()` function to print the entire contents of a list

```
l = [1, 2, 3, 4]
print (l)
>>> [1, 2, 3, 4]
```

List Literals

- ◎ In Python, a list literal is created using **square brackets**.
- ◎ The items within a list can be of ***any type***.
- ◎ The items within a list can be ***objects***

```
# A list of floats.  
l=[3.5, 7.8, 9.2, 1.7]
```

```
# A list of strings.  
l=['paper', 'scissors', 'rock']
```

```
# A list of booleans.  
l=[True, False, True, True, False]
```

```
# A list of objects.  
l=[object1, object2, object3]
```

Mixed Type Lists

- ◎ A list can contain items with **mixed data types**.
- ◎ This is usually a **bad** idea.
 - **Lists** are best used for sequences of **similar items**.
 - **Objects** are a **better** choice for grouping **different (but related) variables**.

```
# A list of mixed types.  
l=[370, True, 'rocket', False, 1.3]
```

Nested Lists

- ⦿ Lists can **contain other lists**.
- ⦿ A list inside another list is called a **nested list**.
- ⦿ This could be useful for **representing a collection of shopping lists**, for *example*.

```
# A list of lists of integers.  
l=[[1, 2], [5, 6], [12, 13]]
```

Empty Lists

- ◎ A list with no items is called an **empty list**.
- ◎ You might want to **start** with an **empty** list if you plan on **building** a list **one** item at a time.

```
# An empty list.  
l=[]
```

List Operations

- ◎ In many ways lists can be worked with in the **same way** as **strings**.
 - Just mentally replace "**string**" with "**list**" and "**characters**" with "**items**".

- ◎ This mental substitution works for:
 - **Concatenation**
 - **Indexing**
 - **Finding length**
 - **Slicing**
 - **Iterating with a for loop**

Concatenation

String Concatenation

Combine the *characters* from two *strings* to create a new *string*.

```
>>> 'abc' + 'de'  
'abcde'
```

List Concatenation

Combine the *items* from two *lists* to create a new *list*.

```
>>> [1, 2, 3] + [88, 0]  
[1, 2, 3, 88, 0]
```


Indexing

String Indexing

Get the *character* at the given index in the *string*.

```
>>> s = 'abc'
>>> s[1]
'b'
```

List Indexing

Get the *item* at the given index in the *list*.

```
>>> l = [7, 8, 9]
>>> l[1]
8
```

Length

String Length

Get the number of *characters* in the *string*.

```
s='abc'  
>>> len(s)  
3  
>>> len('')  
0
```

List Length

Get the number of *items* in the *list*.

```
>>> l= [7, 8, 9]  
>>> len(l)  
3  
>>> len([])  
0
```

Lists are Mutable

- ◎ Unlike **strings**, **lists are mutable**.
 - This means that a list's contents **can change**.
- ◎ Without creating a new list you can do things like:
 - **Replace** items,
 - **Append** new items, and
 - **Remove** existing items.

Replacing Items

- ◎ You can use **indexing** in combination with **assignment** to replace an item in a list.
- ◎ This will **mutate** (change) the list.

```
>>> names = ['alice', 'bob', 'charlie']  
>>> names  
['alice', 'bob', 'charlie']  
  
>>> names[-1] = 'craig'  
>>> names  
['alice', 'bob', 'craig']
```

Appending Items (adding items)

- ◎ **Appending** to a list will **insert** a new **item** at the **end**.
- ◎ **Appending** is a common way of **building** up lists.
- ◎ Use the **append** list method.

```
>>> days = []  
>>> days.append('Mon')  
>>> days.append('Tue')  
>>> days.append('Wed')  
  
>>> days  
['Mon', 'Tue', 'Wed']
```

Example: List of Square Numbers

```
# File: squares_list.py
squares = []
for n in range(1, 11):
    squares.append(n ** 2)
print(squares)
```

```
$ python squares_list.py
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- ◎ This program builds a list of square numbers.
- ◎ It works by using a for loop to iterate over the numbers from 1 to 10 and squaring them.
- ◎ The squared numbers are appended to the list using `append`.

Removing Items

- ◎ Items can be **removed** from a list using the **del** keyword.
- ◎ The part of the list to be removed can be **specified** using an **index** or **slice**.

```
>>> nums = [10, 11, 12, 13, 14, 15, 16]
>>> del nums[2]
>>> nums
[10, 11, 13, 14, 15, 16]

>>> del nums[-3:]
>>> nums
[10, 11, 13]
```

Sorting Items

- ◎ A list can be **sorted** using the **sort** list method.
- ◎ The items in the list will be arranged in **ascending order**.
- ◎ The **reverse** keyword argument can be used to reverse the sort order.

```
>>> items = [3, 4, 12, 1, 5]
>>> items.sort()
>>> items
[1, 3, 4, 5, 12]

>>> letters = ['x', 'z', 'y']
>>> letters.sort(reverse=True)
>>> letters
['z', 'y', 'x']
```


Check Your Understanding

Q. What is the result of this Python expression?

```
len([2, 3, 4] + [8] + [])
```

Check Your Understanding

Q. What is the result of this Python expression?

```
len([2, 3, 4] + [8] + [])
```

A. 4

- Ⓐ The result of the concatenation is [2, 3, 4, 8].
- Ⓑ Concatenating with an empty list does not produce a list with any more items in it.
- Ⓒ The list [2, 3, 4, 8] contains 4 items, so its length is 4.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid grey and others are hollow with a grey outline. The lines connecting them are thin and grey, creating a dense, organic structure.

Working with Lists

Filtering

- © A common thing that you might want a program to do is **filter** a list.
- © Filtering a list means **selecting** items based on a **certain condition** to **create a new list**.
- © Example: **selecting** all **odd** numbers from a **list**.

Example: Filtering

```
# File: filter_odd.py
numbers = [3, 4, 12, 1, 5]

odds = [] # new empty list

for num in numbers:
    if num % 2 == 1:
        odds.append(num)

print('Odd numbers:')
print(odds)
```

```
$ python filter_odd.py
Odd numbers:
[3, 1, 5]
```

- ◎ Loop over every item.
- ◎ Items which pass a certain test (in this case, oddness) are **appended** to a **new list**.

Mapping

- ◎ Another common thing that you might want a program to do is **map** a list.
- ◎ **Mapping** a list means **transforming** each item to **create a new list**.
- ◎ Example: **doubling** all **numbers** in a list.

Example: Mapping

```
# File: map_double.py
numbers = [3, 4, 12, 1, 5]

doubled = [] # new empty list

for num in numbers:
    doubled.append(num * 2)

print('Doubled numbers:')
print(doubled)
```

```
$ python map_double.py
Doubled numbers:
[6, 8, 24, 2, 10]
```

- ◎ Loop over every item.
- ◎ Items are **transformed** and **appended** to a new list.

Incorrect Example: Mapping


Beware!

- © This program will not print out doubled numbers.
- © Assigning to the variable in the for loop (num) will not affect the list (numbers).
- © So, in this program, the for loop is useless!

```
numbers = [3, 4, 12, 1, 5]

for num in numbers:
    num = num * 2

print('Doubled numbers:')
print(numbers)
```



From File to List

- ◎ A file can contain many items of data.
- ◎ In many cases, it is useful to represent this data using a **list** in a Python program.
 - This gives us access to all of the list handling techniques discussed earlier.
- ◎ One way of going from a **file to a list** is by reading **line-by-line** and **appending** to a list.

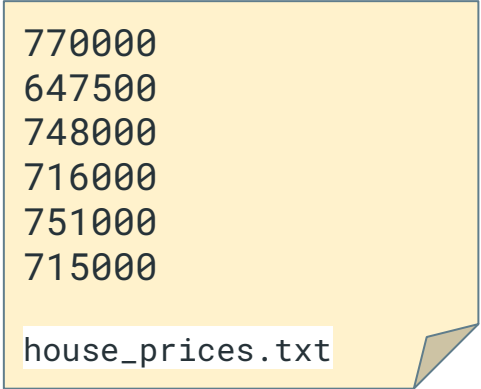
Example: House Prices

```
# File: houses.py

prices = [] # new empty list

for line in open('house_prices.txt'):
    price = float(line)
    prices.append(price)

print(prices)
```



```
770000
647500
748000
716000
751000
715000
```

house_prices.txt

```
$ python houses.py
[770000.0, 647500.0, 748000.0, 716000.0, 751000.0, 715000.0]
```

Example: Cheapest Houses

- Once we have read the file data as a list, we can process it however we want.
- For example, if we want to output the three **cheapest houses** we could:
 - Sort the list.
 - Print the **first three items** in the **sorted** list.

```
# File: cheap_houses.py
prices = [] # new empty list

for line in open('house_prices.txt'):
    price = float(line)
    prices.append(price)

prices.sort() // sort items in list
print(prices[:3])
```

```
$ python cheap_houses.py
[647500.0, 715000.0, 716000.0]
```

From List to File

- ◎ We've now covered a way of **reading** the contents of a **file** into a **list**.
- ◎ But what about the other direction (**list to file**)?
- ◎ One way of writing a list to a file is by placing each **item** on its **own line**.
 - For each item in a list, **convert** it into a **string** and **write** it to the file.

Example: Saving Sorted House Prices

```
# File: sort_houses.py
prices = []
for line in open('house_prices.txt'):
    price = float(line)
    prices.append(price)
prices.sort()

# open file for writing
out_file = open('sorted_prices.txt', 'w')
for price in prices:
    out_file.write(f'{price:.2f}\n')
out_file.close()
```

770000
647500
748000
716000
751000
715000

house_prices.txt

Output:

647500.00
715000.00
716000.00
748000.00
751000.00
770000.00

sorted_prices.txt

Check Your Understanding

Q. You want to create a list of all students whose surname begins with the letter 'N'. Should you be **mapping** and/or **filtering** the original list of students?

Check Your Understanding

Q. You want to create a list of all students whose surname begins with the letter 'N'. Should you be **mapping** and/or **filtering** the original list of students?

A. Filtering only.

- ◎ This problem can be solved by selecting items based on a condition (i.e. checking that the first letter of the surname is 'N').
- ◎ Mapping is not required as the students don't need to be transformed in any way.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting different levels of connectivity or importance. The lines are thin and gray, creating a subtle background pattern.

Lists and References

Lists are Mutable

- ◎ We've established that lists are **mutable**.
- ◎ In an earlier lecture we learnt about how variable assignment causes the **variable** to **reference** an object.
 - Variable assignment **does not copy** the object.
- ◎ Since lists are mutable, this may lead to **unexpected** behaviour if you don't understand **references** properly!

Incorrect Example: Printing the Original List

```
>>> wiggles = ['greg', 'anthony', 'murray', 'jeff']
>>> old_wiggles = wiggles

>>> wiggles[0] = 'sam'

>>> print(old_wiggles)
['sam', 'anthony', 'murray', 'jeff']
```

- ⊙ In this example we are attempting to store the original list in `old_wiggles`.
- ⊙ However, **adding** a new item to `wiggles` **changes** `old_wiggles` **too**.
- ⊙ This is **because** `wiggles` and `old_wiggles` are **both referencing** the **same object**.

Incorrect Example: Printing the Original List

```
1 wiggles = ['greg', 'anthony',  
             'murray', 'jeff']  
2 old_wiggles = wiggles  
3 wiggles[0] = 'sam'
```

Objects (in memory)

References

Incorrect Example: Printing the Original List

```
1 → wiggles = ['greg', 'anthony',  
2      'murray', 'jeff']  
3 old_wiggles = wiggles  
  wiggles[0] = 'sam'
```

Objects (in memory)

'greg'
'anthony'
'murray'
'jeff'

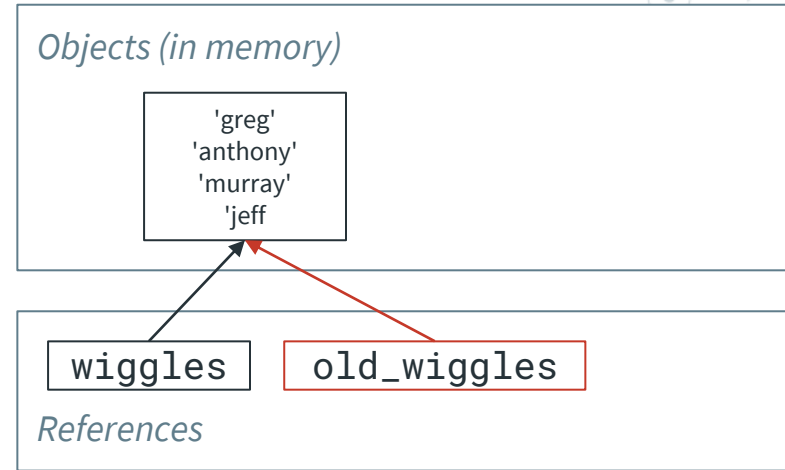
wiggles

References

- ◎ The list object is assigned to variable `wiggles`.

Incorrect Example: Printing the Original List

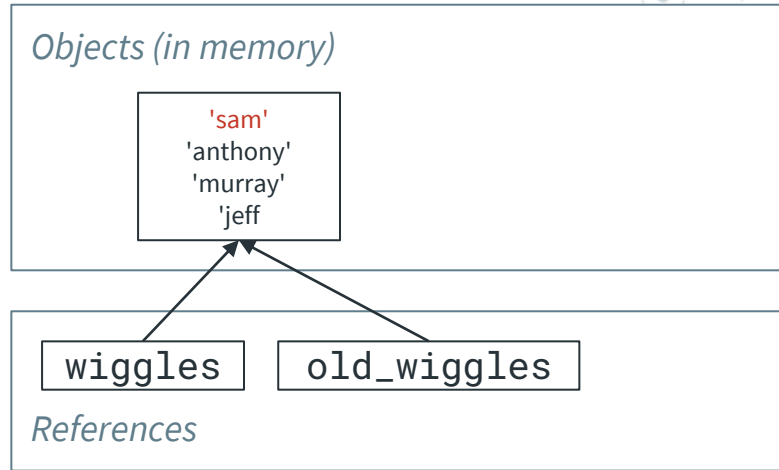
```
1 wiggles = ['greg', 'anthony',  
2           'murray', 'jeff']  
2 → old_wiggles = wiggles  
3 wiggles[0] = 'sam'
```



- ◎ `old_wiggles` is assigned the same list object as `wiggles`.
 - There is still only one list object, the only difference is that there are now two references to it.

Incorrect Example: Printing the Original List

```
1 wiggles = ['greg', 'anthony',  
2           'murray', 'jeff']  
3 → wiggles[0] = 'sam'
```



- ⦿ Changing the list object affects both variables.

Copying Lists

- ◎ If we want two **separate** list objects, we can use the **copy** list method.
- ◎ copy returns a **new** list **object** instance containing the same items as the original list.
 - `my_list.copy()` is `my_list` evaluates to False.
- ◎ **Changing** which items are in the **original** list **won't** affect the **copied** list, and **vice versa**.

Example: Printing the Original List

```
>>> wiggles = ['greg', 'anthony', 'murray', 'jeff']
>>> old_wiggles = wiggles.copy()

>>> wiggles[0] = 'sam'

>>> print(old_wiggles)
['greg', 'anthony', 'murray', 'jeff']
```

- ⦿ Here we assign a *copy* of the original list to `old_wiggles`.
- ⦿ When `wiggles` is mutated, only that list object is affected---the copy of the list referenced by `old_wiggles` is independent and therefore unaffected.

Example: Printing the Original List

```
1 wiggles = ['greg', 'anthony',  
             'murray', 'jeff']  
2 old_wiggles = wiggles.copy()  
3 wiggles[0] = 'sam'
```

Objects (in memory)

References

Example: Printing the Original List

```
1 → wiggles = ['greg', 'anthony',  
2      'murray', 'jeff']  
3 old_wiggles = wiggles.copy()  
   wiggles[0] = 'sam'
```

Objects (in memory)

'greg'
'anthony'
'murray'
'jeff'

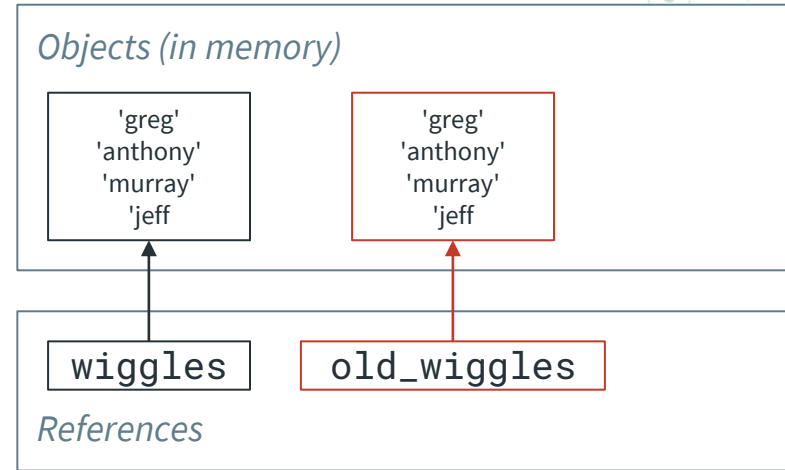
wiggles

References

- ◎ The list object is assigned to variable `wiggles`.

Example: Printing the Original List

```
1 wiggles = ['greg', 'anthony',  
2           'murray', 'jeff']  
2 → old_wiggles = wiggles.copy()  
3 wiggles[0] = 'sam'
```

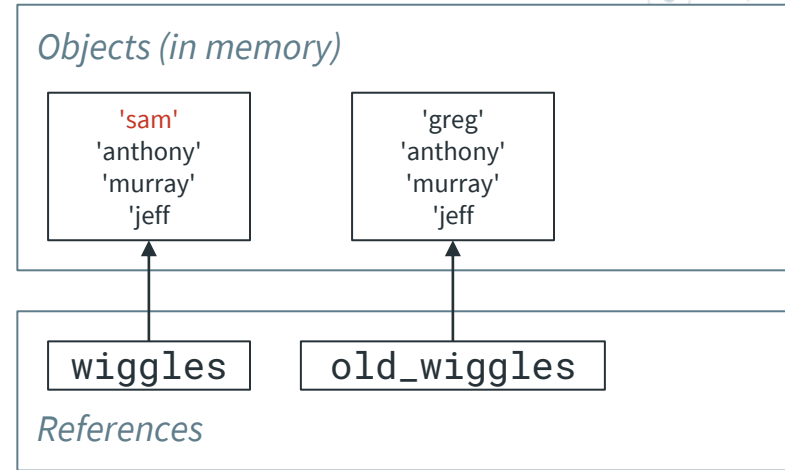


© `old_wiggles` is assigned a copy of the list object referenced by `wiggles`.

There are now **two** list objects.

Example: Printing the Original List

```
1 wiggles = ['greg', 'anthony',  
2           'murray', 'jeff']  
3 → wiggles[0] = 'sam'
```



- ◎ Changing the original list does not affect the copy.

Lists and Functions

- © Since lists are **mutable**, a function which takes a list as an argument can **modify** the list.
- © This means that functions can **affect** list variables without requiring a ***return statement and assignment.***
- © We have actually already seen something like this with the **sort** method.

Example: Lists and Functions

```
1 def delete_first(my_list):  
2     del my_list[0]  
3     l = [1, 2, 3, 4]  
4     print(l)  
5     delete_first(l)  
6     print(l)
```

```
[1, 2, 3, 4]  
[2, 3, 4]
```

- When the function is called on line 5, `my_list` will reference the same list object as `l`.
- This means that deleting the first item from `my_list` inside the function will affect `l` outside of the function.

Check Your Understanding

Q. How many list objects are created by the shown program?

```
1 a = [1, 2, 3]
2 b = [1, 2, 3]
3 c = a
4 d = c.copy()
```

Check Your Understanding

Q. How many list objects are created by the shown program?

A. 3.

- ⦿ **Line 1:** List object created.
- ⦿ **Line 2:** List object created (having the same items doesn't matter).
- ⦿ **Line 3:** No list object created (c references the same list object as a).
- ⦿ **Line 4:** List object created (copying creates a new list).

```
1 a = [1, 2, 3]
2 b = [1, 2, 3]
3 c = a
4 d = c.copy()
```


Examples

Example: Write a python program to replace each element in a list `l=[1, 2, 3, 4, 5, 6]` with its square.

```
>>> l=[1,2,3,4,5,6]
>>> for i in range(len(l)):
...     l[i] = l[i]**2
...
>>> print (l)
[1, 4, 9, 16, 25, 36]
```

Example: Write a python program to count how many items in a list `l` are greater than 2.

```
>>> l=[1,2,3,4,5,6]
>>> count=0
>>> for i in range(len(l)):
...     if l[i] > 2:
...         count +=1
...
>>> print ("Number of item > 2 is {}".format(count))
Number of item > 2 is 4
```

Example: Given a list `l` that contains numbers between 1 and 10, write a python program to create a new list `x` whose first element is how many ones are in `l`, whose second element is how many twos are in `l`, etc.

```
>>> l=[1, 2, 1, 4, 2, 6, 3, 8, 9, 10]
>>> x = []
>>> count=0
>>> for i in range(len(l)):
...     item= l[i]
...     x.append(l.count(item))
...
>>> print (x)
[2, 2, 2, 1, 2, 1, 1, 1, 1, 1]
```

Examples

***, + operators:** +, * operators can be applied to lists.

```
>>> l1 = [1, 2, 3, 4]
>>> l2 = [3, 4, 5, 6]
>>> print (l1+l2)
[1, 2, 3, 4, 3, 4, 5, 6]

>>> print ([1,3]*3)
[1, 3, 1, 3, 1, 3]
>>> print ([0]*4)
[0, 0, 0, 0]
```

List commons functions: len(), sum(), min(), and max(),

```
>>> l = [1, 2, 3, 4]
>>> print (len(l))
4
>>> print (sum(l))
10
>>> print (min(l))
1
>>> print (max(l))
4
```

Looping: We can loop through list using for loop or while loop.

```
>>> l=[1, 2, 6, 8]
>>> for i in l:
...     print (i, " ", end="")
...
1 2 6 8
>>> for i in range(len(l)):
...     print (l[i], " ", end="")
...
1 2 6 8
```

Examples

```
>>> l = [1, 3, 4, 6, 7]
>>> print (l)
[1, 3, 4, 6, 7]

>>> l.append(20) # append to list
>>> print (l)
[1, 3, 4, 6, 7, 20]

>>> for i in range(5):
...     l.append(i + 6)
...
>>> print (l)
[1, 3, 4, 6, 7, 20, 6, 7, 8, 9, 10]
```

```
>>> l.sort() # sort
>>> print (l) # sorted
[1, 3, 4, 6, 6, 7, 7, 8, 9, 10, 20]

>>> l.sort(reverse=True) # sort Descending order
>>> print (l) # sorted Descending order
[20, 10, 9, 8, 7, 7, 6, 6, 4, 3, 1]

>>> l.reverse() # reverse
>>> print (l) # reversed
[1, 3, 4, 6, 6, 7, 7, 8, 9, 10, 20]
```

```
>>> l.remove(8) # remove
>>> print (l) # removed 8
[1, 3, 4, 6, 6, 7, 7, 9, 10, 20]

>>> l.pop(7) # pop
9
>>> print (l) # pop item at location 7
[1, 3, 4, 6, 6, 7, 7, 10, 20]
```

```
>>> l.insert(8,100) # insert
>>> print (l) # insert 100 at location 8
[1, 3, 4, 6, 6, 7, 7, 10, 100, 20]

>>> del l[0] # delete item from list
>>> print (l)
[3, 4, 6, 6, 7, 7, 10, 100, 20]

>>> l.clear() # clear list
>>> print (l)
[]
```

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting different levels of connectivity or importance. The lines are thin and gray, creating a mesh-like structure.

Summary

In This Lecture We...

- ⦿ Created **lists** to store arbitrary sequences of items.
- ⦿ Used methods to manipulate lists.
- ⦿ Explored some of the repercussions of lists being mutable.

A decorative network graph pattern in the top-left corner, featuring a complex web of interconnected nodes and edges. Some nodes are highlighted with blue circles, and others with solid blue dots.

Lecture 4.2

Sets and Dictionaries



Introduction

- ◎ In the previous lecture we looked at storing items in a list.
- ◎ A **list** is an example of a **data structure**---a collection of items that enables efficient access and modification.
- ◎ In this lecture we will learn about two more data structures:
 - **Sets.**
 - **Dictionaries.**

Lecture Overview

1. Sets
2. Dictionaries
3. Example Program: Most Frequent Word

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid grey and others are hollow with a grey outline. The lines connecting them are thin and grey, creating a dense, organic structure.

Sets

What is a Set?

- © A **set** is a data structure for storing an **unordered** collection of items with **unique** values.

What is a Set?

◎ Like a **list**:

- A set is a data structure which can hold multiple items.
- A set is **mutable**.

◎ Unlike a **list**:

- The items within a set are **unordered**.
- Two items with the same value **cannot** be stored in the same set.

Why Use Sets?

- ◎ The purpose of sets might **not be obvious at first**.
- ◎ However, they do have a few good uses:
 - Checking whether an item value is in a set is **very, very, fast**. This makes them useful for **implementing** things like **blacklists**.
 - If you want to remove **duplicate values**, sets can take care of that for you.
 - **Mathematical** sets **operations**

Set Literals

- ◎ In Python, a set can be created using **curly brackets**.

```
# A set of strings.  
s={'north', 'south', 'east', 'west'}  
  
# A set of integers.  
s={2, 3, 5, 7, 11}
```

Empty Sets

- ⦿ A set with no items is called an **empty set**.
- ⦿ You **can't use curly brackets** to create an **empty** set (we'll see why soon).
- ⦿ Instead, the `set` constructor must be used.

```
# An empty set.  
set()
```

```
>>> s = set()
```

Sets are Unordered

- ◎ Items in a set **do not** have an **order**.
 - To use a physical analogy, sets are like **bags**.
- ◎ There is no "**start**" or "**end**" of a set.
- ◎ You **can't index** or **slice** a set.

```
>>> s = {'cat', 'duck', 'dog'}
>>> s
{'dog', 'duck', 'cat'}
>>> s[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

Set Items are Unique

- ◎ The items within a set are always **unique**.
 - That is, no **two** items in a set can have the **same value**.
- ◎ **Duplicate** items specified in a set literal will be **discarded**.

```
>>> my_set = {'nap', 'sleep', 'eat', 'sleep'}  
>>> my_set  
{'sleep', 'eat', 'nap'}
```


Converting Between Sets and Lists

- ◎ A **list** can be converted into a set using the **set** function.
- ◎ Similarly, a **set** can be converted into a **list** using **list** function.
 - The initial ordering of items in the set is arbitrary.
- ◎ Converting a **list** to a **set** and then back again will **remove duplicate** items.

```
>>> l = [3, 2, 1, 2, 3]
>>> s = set(l)

>>> s
{1, 2, 3}

>>> l2 = list(s)

>>> l2
[1, 2, 3]
```

Converting Between Sets and Lists

Beware!

The code below removes an arbitrary item from the set (not 'bam').

```
>>> s = {'bam', 'bang', 'bing'}
>>> l = list(s)
>>> del l[0]
>>> s = set(l)
>>> s
{'bam', 'bang'}
```

- Remember: **sets are unordered.**
- Converting a set to a list will **not** somehow "restore the ordering" of the set literal.

Set Operations

- ◎ The three most important set operations are:
 - **Adding** an item to the set,
 - **Removing** an item from the set, and
 - **Checking** whether an item value is in the set.

Adding Items

- ◎ The **add** set method adds a new item to a set.
 - Can you guess why the method is **not called "append"**?
- ◎ If the item value is already in the set, nothing happens.

```
>>> colours = {'red', 'green'}
>>> colours
{'green', 'red'}

>>> colours.add('blue')
>>> colours
{'green', 'blue', 'red'}

>>> colours.add('blue')
>>> colours
{'green', 'blue', 'red'}
```

Removing Items

- ◎ The **remove** set method removes an item from a set.
- ◎ If the item value is not in the set, an error occurs.

```
>>> nums = {1.1, 1.3, 1.7}
>>> nums
{1.7, 1.1, 1.3}
```

```
>>> nums.remove(1.3)
>>> nums
{1.7, 1.1}
```

```
>>> nums.remove(2.8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2.8
```

Checking Membership

- ◎ The **in** keyword **checks** whether an item value is in a set.
 - We call this **checking membership**.
- ◎ Use **not in** to check whether an item value is **not** in a set.

```
>>> dirs = {'up', 'down', 'left', 'right'}
>>> dirs
{'up', 'right', 'left', 'down'}

>>> 'left' in dirs
True

>>> 'forwards' in dirs
False

>>> 'forwards' not in dirs
True
```

Examples

Copying a set: A set can be copied using `=` operator or `copy()` function. If we use `=` operator, any changes we made in one set it will be reflected in others.

```
>>> l = {1,2,3,4}
>>> l2 = l
>>> print (l2)
{1, 2, 3, 4}
>>> l2.add (20)
>>> print (l2)
{1, 2, 3, 4, 20}
>>> print (l)
{1, 2, 3, 4, 20}
>>> l3 = l.copy()
>>> l3.add (50)
>>> print (l3)
{1, 2, 3, 4, 20, 50}
>>> print (l)
{1, 2, 3, 4, 20}
```

Set commons functions: `len()`, `sum()`, `min()`, and `max()`, `sort()`

```
>>> l = {1,2,3,4}
>>> print (len(l))
4
>>> print (sum(l))
10
>>> print (min(l))
1
>>> print (max(l))
4
>>> l = {10,2,6,1,8}
>>> l_sorted = sorted(l)
>>> print (l_sorted)
[1, 2, 6, 8, 10]
```

Set Methods

Set method

Set methods: We can write `dir(set)` in Python shell (`>>>`) to get set methods and functions. You could also get the description of all methods using `help(set)`.

```
>>> print (dir(set))
>>>
['_and__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
 '__iand__', '__init__', '__init_subclass__', '__ior__', '__isub__', '__iter__',
 '__ixor__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__',
 '__rand__', '__reduce__', '__reduce_ex__', '__repr__', '__ror__', '__rsub__',
 '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
 '__subclasshook__', '__xor__', 'add', 'clear', 'copy', 'difference',
 'difference_update', 'discard', 'intersection', 'intersection_update', '
 isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference',
 'symmetric_difference_update', 'union', 'update']
```

Method	Description
Union()	Union of sets
Intersection()	Intersection of two sets
Difference()	Difference between two or more sets
Symmetric_Difference()	Symmetric differences of two sets
issuperset()	Check if this set contains another set or not
issubset()	Check whether another set contains this set or not

Check Your Understanding

Q. If the user inputs `fred`, will they be welcomed into the party?

```
1 guests = {'tim', 'betty', 'rod'}
2 guests.add('fred')
3 name = input('Enter name: ')
4 if name in guests:
5     print('Welcome to the party!')
6 else:
7     print('Not invited!')
```

Check Your Understanding

Q. If the user inputs `fred`, will they be welcomed into the party?

A. Yes.

- ⦿ After line 2, `guests` will contain the items `'betty'`, `'fred'`, `'rod'`, and `'tim'`.
- ⦿ After user input, `name` will contain `'fred'`.
Since `'fred'` is in `guests`, the condition on line 4 is true.

```
1 guests = {'tim', 'betty', 'rod'}
2 guests.add('fred')
3 name = input('Enter name: ')
4 if name in guests:
5     print('Welcome to the party!')
6 else:
7     print('Not invited!')
```



Dictionaries

What is a Dictionary?

Dictionary

Dictionaries are dynamic data structure that store sequence of items presented in **key:value** pairs. Items keys are **unique** but values can be homogeneous or heterogeneous.

Dictionaries are **mutable** so items can be added to or removed from them. We can use indexing, slicing, and various methods to work with dictionaries.

How can we create a dictionary?

Dictionaries can be created using curly brackets **{ }** and colon **(:)** to separate the key and value pairs. We can also use **dict()** constructor. Items inside dictionary are separated by commas.

```
>>> d = {} # empty dict
>>> print(type(d))
<class 'dict'>
>>> d = dict() # empty dict
>>> print(type(d))
<class 'dict'>
>>> d = {1: "one", 2: "two"}
>>> print(d)
{1: 'one', 2: 'two'}
```

```
>>> d1 = {"one": 1, "two": 2}
>>> print(d1)
{'one': 1, 'two': 2}
>>> # create a dictionary named subject
>>> subject = {"Title": "CSE4IP", "Sem": 1, "Mode": "Online", "No": 100}
>>> print(subject)
{'Title': 'CSE4IP', 'Sem': 1, 'Mode': 'Online', 'No': 100}
```

What is a Dictionary?

- ◎ **Values** can be looked up by their associated **keys**.
 - This is like looking up a definition (value) of a word (key) in a physical dictionary.
 - The **keys** in a dictionary are always **unique**.
- ◎ Like **sets** and **lists**, dictionaries are **mutable**.
 - Items can be **added** and **removed** without creating a new dictionary.

What is a Dictionary?

- ◎ Another way of thinking about a dictionary is as a more general list.
 - In a list, the **indices** are sequential integers starting at 0.
 - In a dictionary, the **indices** (**keys**) can be almost anything.
- ◎ This allows us to write code like

```
properties['colour'] = 'blue'
```

Dictionary Literals

- ◎ In Python, a dictionary can be created using **curly brackets** and **colons**.
- ◎ The **colon** is used to **separate keys** from associated **values**.

```
# A dictionary with string keys and values.  
{ 'Bob': '0491577644', 'Jim': '0491570156' }
```

```
# A dictionary with integer keys and float values.  
{ 1: 10.0, 2: 18.5, 3: 25.0 }
```

Empty Dictionaries

- ◎ Empty dictionaries are created using curly brackets with nothing between them.
 - This is why empty sets must be created using `set()`.

```
# An empty dictionary.  
{}
```


Indexing Dictionaries

- ◎ **Indexing** a dictionary with **keys** can be used for getting and setting values.
- ◎ In this way, indexing works similarly to lists.
 - One **difference** is that indexing can be used to **add** a **new item** to a dictionary.

Dictionary Literals

Dictionaries are known to be more general version of **lists**. For example, create a list that contains the number of days of each month of the year.

```
>>> month_days = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

If we want to print the number of days for **January** we can use `month_days[0]`, `month_days[1]` for **February** and so on.

```
>>> print (month_days[0])
```

```
31
```

```
>>> print (month_days[1])
```

```
28
```

Lets re-implement the above example using **dictionary** instead of **list**.

```
>>> month_days = {'January':31, 'February':28, 'March':31, 'April':30, 'May':31, 'June':30, 'July':31,  
...              'August':31, 'September':30, 'October':31, 'November':30, 'December':31}
```

If we want to print the number of days for **January** we can use `month_days['January']`, `month_days['February']` for **February** and so on.

```
>>> print (month_days['January'])
```

```
31
```

```
>>> print (month_days['February'])
```

```
28
```

As can be seen, using a dictionary, the code is more readable, and we don't need to know the index of each month. We can use the month names to print the number of days.

Indexing Dictionaries

The item values in dictionary can be of any data type such as String, int, boolean and **list**.

```
>>> subject = {"Title": "CSE4IP", "Sem": 1, "Mode": False, "Labs": [1,2]}
>>> print (subject)
{'Title': 'CSE4IP', 'Sem': 1, 'Mode': False, 'Labs': [1, 2]}
```

There are two different methods to access the items of a dictionary.

- Using item **key** inside square brackets **[key]** to get item **value**.
- Using item **key** as a parameter of **get()** method to get item **value**.

```
>>> subject = {"Title": "CSE4IP", "Sem": 1, "Mode": False, "Labs": [1,2]}
>>> print (subject["Title"]) # using square brackets [key]
CSE4IP
>>> print (subject["Labs"])
[1, 2]
>>> print (subject["Labs"][1])
2
```

```
>>> subject = {"Title": "CSE5APG", "Sem": 1, "Mode": False, "Labs": [1,2]}
>>> print (subject.get("Title")) # Using get() method
CSE5APG
```

Setting Values

- ◎ If a **key does not exist** in a dictionary, setting its value will **add** a new item to the dictionary.
- ◎ If a **key does exist** in a dictionary, setting its value will **replace** the **old** value.

```
>>> pet = {'name': 'fido', 'legs': 4}
>>> pet
{'name': 'fido', 'legs': 4}

>>> pet['colour'] = 'brown'
>>> pet
{'name': 'fido', 'legs': 4, 'colour': 'brown'}

>>> pet['legs'] = 3
>>> pet
{'name': 'fido', 'legs': 3, 'colour': 'brown'}
```

Getting Values

The following methods can be used to retrieve all key or values of a given dictionary.

Method	Method
keys()	Returns the list of all keys .
values()	Returns the list of all values.
items()	Returns all the items as tuple of a key-value pair.

```
>>> subject = {"Title": "CSE4IP", "Sem": 1, "Mode": False, "Labs": [1,2]}
>>> # Get all keys
>>> print(subject.keys())
dict_keys(['Title', 'Sem', 'Mode', 'Labs'])
>>> # Get all values
>>> print(subject.values())
dict_values(['CSE4IP', 1, False, [1, 2]])
>>> # Get all items key-value
>>> print(subject.items())
dict_items([('Title', 'CSE4IP'), ('Sem', 1), ('Mode', False), ('Labs', [1, 2])])
```

Getting Values

- ⦿ Indexing by an existing **key** will return its associated value.
- ⦿ Indexing by a **key** which **does not exist** in a dictionary will result in an error.

```
>>> nums = {1: 'one', 2: 'two', 3: 'three'}  
>>> nums[2]  
'two'
```

```
>>> nums[0]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 0
```

Removing Items

- © The `del` keyword can be used to remove an item from a dictionary (just like lists).

```
>>> nums = {1: 'one', 2: 'two', 3: 'three'}  
>>> del nums[1]  
  
>>> nums  
{2: 'two', 3: 'three'}
```

Iterating Over Dictionaries

- ◎ You can iterate over the items in a dictionary using a for loop and the `items` dictionary method.

```
# File: contacts.py
contacts = {'Bob': '0491577644', 'Jim': '0491570156'}
for name, phone in contacts.items():
    print(f'{name}\''s phone number is {phone}')
```

```
$ python contacts.py
Bob's phone number is 0491577644
Jim's phone number is 0491570156
```




Example Program: Most Frequent Word

Task Definition

Create a program which reads a text file called "words.txt" that contains one word per line. Display the word that appears most often and the number of times that it appears. If there is a tie, display any one of the tied words.

Example

```
play  
duck  
duck  
goose  
duck
```

words.txt

```
$ python word_count.py  
duck  
3
```

Identifying Inputs and Outputs

Input

- ◎ The words contained in the text file "words.txt".

Output

- ◎ The most frequent word and its count.

Identifying Processing Steps

- ◎ By breaking up the problem we end up with smaller, **easier sub-problems** to solve.
- ◎ This particular problem can be naturally broken into **two halves**.
- ◎ The two halves are connected using what I call "**intermediate**" inputs and outputs.
 - Not "real" inputs and outputs (remain internal).
 - Indicate values which are passed from one sub-problem to another.

Identifying Processing Steps

1. Read through the file and **count how many times each word is seen**.

- ⦿ Input: the words in "words.txt".
- ⦿ Intermediate output: words with counts.

2. Aggregate the word counts to **find the most frequent word**.

- ⦿ Intermediate input: words with counts.
- ⦿ Output: the **most frequent** word and its **count**.

Sub-problem 1: Counting Words

- ◎ The goal of the first half of the program is to get *all* words and their associated counts.
- ◎ We will now go through this process manually to help us identify the processing steps.
- ◎ At this stage, we want to pay close attention to the steps that we are taking as we work through the example input.

Identifying Processing Steps

```
play  
duck  
duck  
goose  
duck
```

```
words.txt
```

WORD	COUNT

Identifying Processing Steps



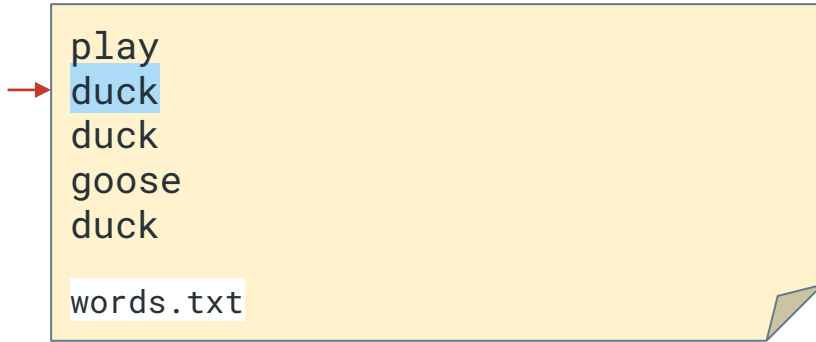
```
play  
duck  
duck  
goose  
duck
```

words.txt

WORD	COUNT
play	1

- ◎ This is the first time we've seen the word "play".
- ◎ Write down "play" with a count of 1.

Identifying Processing Steps



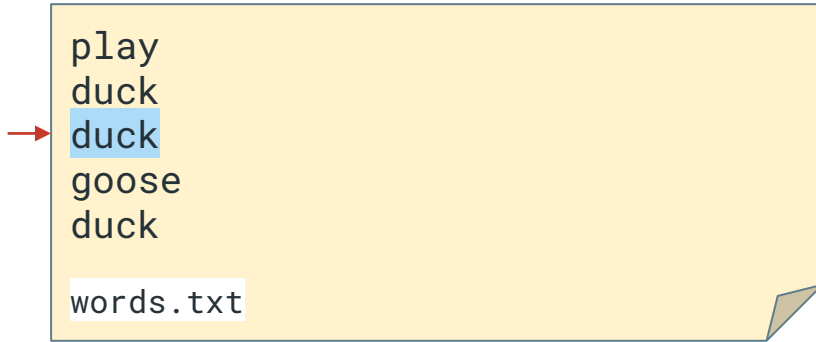
play
→ duck
duck
goose
duck

words.txt

WORD	COUNT
play	1
duck	1

- ◎ This is the first time we've seen the word "duck".
- ◎ Write down "duck" with a count of 1.

Identifying Processing Steps



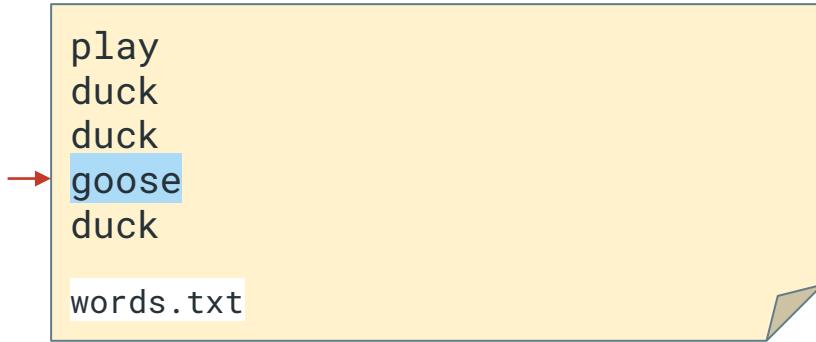
play
duck
→ duck
goose
duck

words.txt

WORD	COUNT
play	1
duck	2

- ◎ We've already seen the word "duck" (it's in the table).
 - ◎ Calculate the new count for "duck" by adding one to the previous count (which was 1).
 - The new count is $1 + 1 = 2$.
- Replace the count for "duck" with 2.

Identifying Processing Steps



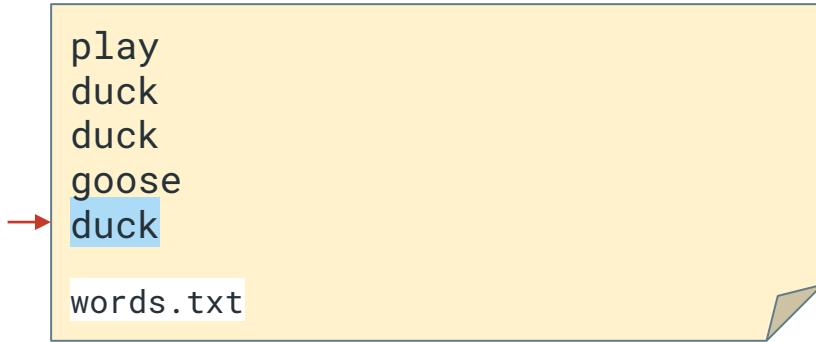
play
duck
duck
goose
duck

words.txt

WORD	COUNT
play	1
duck	2
goose	1

- ◎ This is the first time we've seen the word "goose".
- ◎ Write down "goose" with a count of 1.

Identifying Processing Steps



play
duck
duck
goose
→ duck

words.txt

WORD	COUNT
play	1
duck	3
goose	1

- ◎ We've already seen the word "duck" (it's in the table).
 - ◎ Calculate the new count for "duck" by adding one to the previous count (which was 2).
 - The new count is $2 + 1 = 3$.
- Replace the count for "duck" with 3.

Analyse Processing Steps

- ◎ We consider one word at a time.
- ◎ We make a decision every time we encounter a word (*have we not seen this word before?*).
 - If the word has not been seen, associate a count of 1 with the word.
 - If the word has been seen, add one to the existing count for the word.

Analyse Processing Steps

- ◎ We consider one word at a time. ← for loop
- ◎ We make a decision every time we encounter a word (*have we not seen this word before?*). ← if condition
 - If the word has not been seen, **associate** a ← if block
count of 1 with the word.
 - If the word has been seen, **add** one to the ← else block
existing count for the word.

Selecting a Data Structure

- ◎ During our manual processing we were tracking **word/count pairs**.
 - Each **count** was associated with a **word**.
- ◎ We know that **dictionaries** allow us to associate values with keys.
- ◎ Therefore a dictionary would be an appropriate data structure to use for word counts.
 - The **keys** are the **words** (strings).
 - The **values** are the **counts** (integers).

Coding from Processing Steps

- ◎ We consider one word at a time.
- ◎ We make a decision every time we encounter a word (*have we not seen this word before?*).
 - If the word has not been seen, associate a count of 1 with the word.
 - If the word has been seen, add one to the existing count for the word.

Coding from Processing Steps

- ◎ We consider one word at a time.
- ◎ We make a decision every time we encounter a word (*have we not seen this word before?*).
 - If the word has not been seen, associate a count of 1 with the word.
 - If the word has been seen, add one to the existing count for the word.

```
word_file = open('words.txt')
counts = {}
for line in word_file:
    word = line[:-1]
    if word not in counts:
        counts[word] = 1
    else:
        old_count = counts[word]
        counts[word] = old_count + 1
```

Coding from Processing Steps

- ◎ We consider one word at a time.
- ◎ We make a decision every time we encounter a word (*have we not seen this word before?*).
 - If the word has not been seen, associate a count of 1 with the word.
 - If the word has been seen, add one to the existing count for the word.

```
word_file = open('words.txt')
counts = {}
for line in word_file:
    word = line[:-1]
    if word not in counts:
        counts[word] = 1
    else:
        old_count = counts[word]
        counts[word] = old_count + 1
```

Coding from Processing Steps

- © We consider one word at a time.
- © We make a decision every time we encounter a word (*have we not seen this word before?*).
 - If the word has not been seen, associate a count of 1 with the word.
 - If the word has been seen, add one to the existing count for the word.

```
word_file = open('words.txt')
counts = {}
for line in word_file:
    word = line[:-1]
    if word not in counts:
        counts[word] = 1
    else:
        old_count = counts[word]
        counts[word] = old_count + 1
```

Coding from Processing Steps

- © We consider one word at a time.
- © We make a decision every time we encounter a word (*have we not seen this word before?*).
 - If the word has not been seen, associate a count of 1 with the word.
 - If the word has been seen, add one to the existing count for the word.

```
word_file = open('words.txt')
counts = {}
for line in word_file:
    word = line[:-1]
    if word not in counts:
        counts[word] = 1
    else:
        old_count = counts[word]
        counts[word] = old_count + 1
```

Coding from Processing Steps

- ◎ We consider one word at a time.
- ◎ We make a decision every time we encounter a word (*have we not seen this word before?*).
 - If the word has not been seen, associate a count of 1 with the word.
 - If the word has been seen, add one to the existing count for the word.

```
word_file = open('words.txt')
counts = {}
for line in word_file:
    word = line[:-1]
    if word not in counts:
        counts[word] = 1
    else:
        old_count = counts[word]
        counts[word] = old_count + 1
```

Sub-problem 1: Completed

- ◎ This code will get the counts of all words in "words.txt".
- ◎ Our "intermediate output" is the variable `counts`, which contains the count for each word.
- ◎ `counts` will be used by the second sub-problem.

```
word_file = open('words.txt')
counts = {}
for line in word_file:
    word = line[:-1]
    if word not in counts:
        counts[word] = 1
    else:
        old_count = counts[word]
        counts[word] = old_count + 1
```

Sub-problem 2: Finding the Most Frequent Word

- ◎ The goal of the second half of the program is to find the most frequent word.
 - This is a form of **aggregation** (we are looking for the **maximum value** word count).
- ◎ We have already seen how to find the maximum value using a for loop in an earlier lecture.
- ◎ The twist here is that we also want to know the word associated with the highest count.


Identifying Processing Steps

WORD	COUNT
play	1
duck	3
goose	1

MAX. COUNT	MAX. WORD
0	

- ◎ Start with a **maximum** word count of **0** and a **blank maximum** word.

Identifying Processing Steps



WORD	COUNT
play	1
duck	3
goose	1

MAX. COUNT	MAX. WORD
0	play

- ⦿ The word count for "play" (1) is higher than our current maximum (0).
 - Set our maximum count to 1.
 - Set our maximum word to "play".

Identifying Processing Steps

WORD	COUNT
play	1
→ duck	3
goose	1

MAX. COUNT	MAX. WORD
± 3	duck

- ◎ The word count for "duck" (3) is higher than our current maximum (1).
 - Set our maximum count to 3.
 - Set our maximum word to "duck".

Identifying Processing Steps

WORD	COUNT
play	1
duck	3
→ goose	1

MAX. COUNT	MAX. WORD
3	duck

- ◎ The word count for "goose" (1) is **not** higher than our current maximum (3).
 - Do nothing.

Analysing Processing Steps

- ◎ Start with a maximum word count of 0 (and a blank maximum word).
- ◎ We consider one **word/count pair** at a time.
- ◎ If the count is higher than our current maximum, update our current maximum.

Coding from Processing Steps

- ◎ Start with a maximum word count of 0 (and a blank maximum word).
- ◎ We consider one word/count pair at a time.
- ◎ If the count is higher than our current maximum, update our current maximum.

Coding from Processing Steps

- ◎ Start with a maximum word count of 0 (and a blank maximum word).
- ◎ We consider one word/count pair at a time.
- ◎ If the count is higher than our current maximum, update our current maximum.

```
max_count = 0
max_word = ''
for word, count in counts.items():
    if count > max_count:
        max_count = count
        max_word = word
```

Coding from Processing Steps

- ◎ Start with a maximum word count of 0 (and a blank maximum word).
- ◎ We consider one word/count pair at a time.
- ◎ If the count is higher than our current maximum, update our current maximum.

```
max_count = 0
max_word = ''
for word, count in counts.items():
    if count > max_count:
        max_count = count
        max_word = word
```

Coding from Processing Steps

- ◎ Start with a maximum word count of 0 (and a blank maximum word).
- ◎ We consider one word/count pair at a time.
- ◎ If the count is higher than our current maximum, update our current maximum.

```
max_count = 0
max_word = ''
for word, count in counts.items():
    if count > max_count:
        max_count = count
    max_word = word
```


Coding from Processing Steps

- ◎ Start with a maximum word count of 0 (and a blank maximum word).
- ◎ We consider one word/count pair at a time.
- ◎ If the count is higher than our current maximum, update our current maximum.

```
max_count = 0
max_word = ''
for word, count in counts.items():
    if count > max_count:
        max_count = count
        max_word = word
```

Putting Everything Together

- ◎ Now that we've solved both sub-problems, all that's left is to put everything together in the one script file.
- ◎ We also add two print statements at the end to display the most frequent word and its count.

```
1 word_file = open('words.txt')
2 counts = {}
3 for line in word_file:
4     word = line[:-1]
5     if word not in counts:
6         counts[word] = 1
7     else:
8         old_count = counts[word]
9         counts[word] = old_count + 1
10
11 max_count = 0
12 max_word = ''
13 for word, count in counts.items():
14     if count > max_count:
15         max_count = count
16         max_word = word
17
18 print(max_word)
19 print(max_count)
```

Open the words file.
Create an empty dictionary.
Loop over lines in the file.
Remove the newline character.
If we haven't seen the word...
...give it a count of 1.
Otherwise...
...get the old count...
...and add 1 to it.

Start with a max count of 0.
Start with a blank max word.
Loop over word/count pairs.
Whenever we find a higher count...
...update max count...
...and update max word.

Display max word.
Display max count.

Check Your Understanding

Q. If the initial `max_count` was set to 3 instead of 0, what would the program output be?

WORD	COUNT
play	1
duck	3
goose	1

```
1 max_count = 3
2 max_word = ''
3 for word, count in counts.items():
4     if count > max_count:
5         max_count = count
6         max_word = word
7
8 print(max_word)
9 print(max_count)
```

Check Your Understanding

Q. If the initial `max_count` was set to 3 instead of 0, what would the program output be?

A. Word: (blank), count: 3.

- ⦿ No word in the dictionary has a count higher than 3.
- ⦿ This means that `max_count` and `max_word` never change.

WORD	COUNT
play	1
duck	3
goose	1

```
1 max_count = 3
2 max_word = ''
3 for word, count in counts.items():
4     if count > max_count:
5         max_count = count
6         max_word = word
7
8 print(max_word)
9 print(max_count)
```

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting a hierarchical or central structure. The lines are thin and gray, connecting the nodes in a non-linear fashion.

Summary

In This Lecture We...

- ◎ Discovered the **set** and **dictionary** data structures.
- ◎ Learnt about how these data structures differ from lists, and how to use them.
- ◎ Solved a programming task using a dictionary as the main data structure.

Next Lecture We Will...

- ◎ Learn about the different kinds of software errors.

A decorative network diagram in the top-left corner of the slide. It features a complex web of interconnected nodes and edges. The nodes are represented by small circles, some of which are solid blue, some are solid grey, and some are hollow with a blue outline. The edges are thin grey lines connecting the nodes. The overall shape of the network is roughly triangular, pointing towards the top-left corner.

Lecture 4.2

Software Errors

A decorative network diagram in the bottom-right corner of the slide. It features a complex web of interconnected nodes and edges. The nodes are represented by small circles, some of which are solid blue, some are solid grey, and some are hollow with a blue outline. The edges are thin grey lines connecting the nodes. The overall shape of the network is roughly triangular, pointing towards the bottom-right corner.

Topics 4.2 and 4.3 Intended Learning Outcomes

- ◎ By the end of the **week** you should be able to:
 - Understand the difference between **syntax, runtime, and logic errors**,
 - Recognise and fix common syntax errors in Python,
 - Uncover logic errors using **assert statements** and **test cases**, and
 - Handle and raise runtime errors using **exceptions**.

Introduction

- ◎ Even the most skilled programmers must deal with errors regularly.
- ◎ In this lecture we will learn about the three main kinds of errors: **syntax errors**, **runtime errors**, and **logic errors**.
- ◎ We will also discuss strategies for **debugging** (finding and fixing errors).

Lecture Overview

1. Kinds of Errors
2. Asserts
3. Testing Code

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by circles of varying sizes, some with concentric rings, and the lines are thin and grey. The overall structure is organic and sprawling, resembling a molecular or neural network.

Kinds of Errors

1.1 Syntax Errors

1.2 Runtime Errors

1.3 Logic Errors

1.1 Syntax Errors

- ◎ A **syntax error** is an error in the way that the source code of a program is written.
 - It is a **violation** of the strict set of **rules** that a programming language lays out.
 - Syntax errors are commonly caused by **accidentally typing** the **wrong character**.

1.1 Syntax Errors

- ◎ Syntax errors often **prevent** a program from being **run** at all.
- ◎ Syntax errors are more common amongst **beginner** programmers.
 - You will make **fewer** syntax errors as your familiarity with a programming language develops (*just like natural languages*).
- ◎ Usually the **easiest** kind of error to **find** and **fix**.

Python's `SyntaxError`

- ◎ In Python, a `SyntaxError` is raised when there is a **syntax** error in code.
 - `IndentationError` and `TabError` are more specific types of `SyntaxError`.
- ◎ Python will try to explain **where** the error is.
 - The error **message** includes a **line number**.

bad_bracket.py

```
1 my_dict = {'A': 65, 'B': 66}
2 my_dict['C') = 67
3 my_dict['D'] = 68
4 print(my_dict)
```

```
File "bad_bracket.py", line 2
    my_dict['C') = 67
                ^
```

```
SyntaxError: invalid syntax
```


Common Syntax Errors

- ◎ Common causes of Python syntax errors include:
 - **Incorrect** use of **assignment**,
 - **Misspelled/misused/missing keywords**,
 - **Misused/missing symbols**,
 - **Mismatched** parentheses, brackets, and quotes,
 - **Incorrect indentation**, and
 - **Empty blocks**.

We will now look at examples of these.

Incorrect use of Assignment

```
# Incorrect
```

```
'Wilfred' = name
```

```
SyntaxError: can't assign to literal
```

```
# Fixed
```

```
name = 'Wilfred'
```

- ⦿ The variable receiving the reference must always be on the **left side** of the assignment statement.
 - It might help to read the "=" as "**is assigned**" or "**now refers to**".

Incorrect use of Assignment

```
# Incorrect
if age = 16:
    print('Sweet sixteen!')
```

```
SyntaxError: invalid syntax
```

```
# Fixed
if age == 16:
    print('Sweet sixteen!')
```

- ◎ To **test** whether two objects have the same value, use **double equals** (==).
 - In Python, **single equals** (=) is used for **assignment** only.

Misspelled Keywords

```
# Incorrect
if t < 10:
    print('Cold')
elif t > 30:
    print('Hot')
```

SyntaxError: invalid syntax

```
# Fixed
if t < 10:
    print('Cold')
elif t > 30:
    print('Hot')
```

- ⦿ Ensure that keywords are spelled correctly.
- ⦿ Syntax highlighting in your text editor helps, since keywords are shown in a different colour.

Misused Keywords

```
# Incorrect
```

```
return = 5
```

```
SyntaxError: invalid syntax
```

```
# Fixed (if defining a variable)
```

```
return_val = 5
```

```
# Fixed (if returning a value)
```

```
return 5
```

- ⦿ Don't use **reserved keywords** as **variable** or **function names**.
- ⦿ Ensure that keywords are used correctly.
 - e.g. **break** and **continue** only make sense in **loops**.

Mismatched Quotes

Incorrect

```
store = 'Bunning's'
```

```
SyntaxError: invalid syntax
```

Fixed

```
store = 'Bunning\'s'
```

- ◎ Ensure that **escape** characters are used when necessary.
- ◎ Don't forget to end strings with **quotation** marks.
- ◎ Ensure that you start and end string literals with the same kind of **quotation** mark (i.e. double or single).

Incorrect Indentation

```
# Incorrect
if cost > 10:
    print('Too expensive')
else:
    print('Buy!')
```

IndentationError: unindent does not
↳ match any outer indentation level

```
# Fixed
if cost > 10:
    print('Too expensive')
else:
    print('Buy!')
```

- ◎ Always **indent/de-indent** by **4 spaces** at a time (don't mix different types of indentation).
- ◎ **Indent** statements after **if, elif, else, while, for**, etc.

Empty Blocks

Incorrect

```
def do_nothing():
```

```
IndentationError: expected an  
  ↪ indented block
```

Fixed

```
def do_nothing():  
    pass
```

- ◎ Use the `pass` keyword if you want to have a block that doesn't do anything.

1.2 Runtime Errors

- ◎ A **runtime error** is an error which arises while a program is running.
 - The program has **correct syntax**.
- ◎ **Runtime errors** can **cause** a program to **crash** (exit **unexpectedly** and display an error message).

Common Runtime Errors

- ◎ Common causes of Python runtime error include:
 - **Opening** a file for reading that **doesn't exist**,
 - **Dividing** a number by **zero**,
 - **Attempting** to use an **undefined variable**, and
 - **Using** a **non-existent index** with a **list/dictionary**.

Handling Runtime Errors

- ◎ Sometimes runtime errors can be **avoided** by performing a **check beforehand**.
 - e.g. Does the file **exist**?
- ◎ Alternatively, runtime errors can be **handled** by the **program** to recover from the error.
- ◎ We will learn about **handling runtime** errors in the **next** lecture.

1.3 Logic Errors

- ◎ A **logic error** is an error that causes **incorrect behaviour** of a **program**.
 - The program has **correct syntax** and perhaps even runs without crashing, but **does not** behave as **expected**.

1.3 Logic Errors

- ◎ Logic errors are **caused** by **poorly** designed **algorithms**.
- ◎ Logic errors are typically the **most difficult** kind of error to **find** and **fix**, since they **don't result** in **informative** error messages being displayed.
- ◎ The rest of the **lecture** **cover** techniques for **revealing logic errors**.

Avoiding Logic Errors

- ◎ The best way of avoiding logic errors is to think carefully about your **algorithm design**.
 - A **sloppy** or **incomplete** flowchart is much more likely to result in logic errors.
- ◎ Additionally, try to **ensure** that the code you write **matches your intent**.
 - e.g. If a person must be "over 18", you probably want to use `age >= 18`, not `age > 18`.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid grey and others are hollow with a grey outline. The lines are thin and grey, connecting the nodes in a non-linear fashion.

Asserts

Assert Statements

- ◎ Logic errors are often difficult to **identify** because they don't **produce** obvious **error messages**.
- ◎ Using **assert statements** it is possible to **expose** some **logic errors** as **runtime** errors.

Assert Statements

- ◎ An **assert** statement contains a **condition** (boolean expression) which is expected to always be true.
 - If the condition **evaluates** to **false**, a runtime error will **occur**.
- ◎ By **adding assert** statements to your code based on what you expect to be **true**, you will be **alerted** to flaws in your program's **logic**.

Assert Statements

- ◎ In Python, an **assert** statement is written using the **assert keyword** and a **condition**.
 - If the condition is **true**, **nothing** happens.
 - If the condition is **false**, an **AssertionError** is raised.

```
>>> msg = 'Hello'
>>> assert len(msg) > 1
>>> assert len(msg) == 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Example: Assert Debugging

- ◎ Let's say that we want to write a program which reverses a string, then replaces the last character of the reversed string with an asterisk.
- ◎ We have a go at this, but find that our code does not produce the results that we expect.

Example: Assert Debugging

```
s = 'heart'  
# Reverse the string.  
r = ''  
for i in range(-1, -len(s), -1):  
    r = r + s[i]  
# Replace the last letter.  
f = r[:-1] + '*'  
print(f)
```

Output:

Expected:

tra*

trae*

Example: Assert Debugging

- ◎ One too many characters has been removed!
 - Something is wrong with the logic of our program.
- ◎ But which part of the program contains the problem?
 - Is it in the reversal part?
 - Is it in the replacement part?
- ◎ Adding assert statements can help us narrow in on the error.

Example: Assert Debugging

```
1 s = 'heart'
2 # Reverse the string.
3 r = ''
4 for i in range(-1, -len(s), -1):
5     r = r + s[i]
6     assert len(r) == len(s)
7     # Replace the last letter.
8     f = r[:-1] + '*'
9     assert len(f) == len(r)
10 print(f)
```

- © Here we have added two assert statements.
1. The **length** of the string should **not** change after reversing.
 2. The length of the string should not change after replacing the last character.

Example: Assert Debugging

asserts.py

```
1 s = 'heart'
2 # Reverse the string.
3 r = ''
4 for i in range(-1, -len(s), -1):
5     r = r + s[i]
6 assert len(r) == len(s)
7 # Replace the last letter.
8 f = r[:-1] + '*'
9 assert len(f) == len(r)
10 print(f)
```

```
$ python asserts.py
Traceback (most recent call last):
  File "asserts.py", line 6, in <module>
    assert len(r) == len(s)
AssertionError
```

- ◎ It was the **first assertion** (line 6) which failed!
- ◎ So there is an issue with the string reversal part of our program.

Example: Assert Debugging

- ◎ We can further investigate the string reversal code using a temporary print statement.
- ◎ Here's what we find:
 - The final expected repetition of the for loop is missing!
 - This means that the range sequence is too short.

```
1 s = 'heart'
2 # Reverse the string.
3 r = ''
4 for i in range(-1, -len(s), -1):
5     r = r + s[i]
6     print(r)
7 assert len(r) == len(s)
...

```

```
$ python asserts.py
```

```
t
tr
tra
trae
```

We never get to
traeh

```
Traceback (most recent call last):
  File "asserts.py", line 7, in <module>
    assert len(r) == len(s)
AssertionError
```


Example: Assert Debugging

- ◎ The fix is to adjust the stopping value of the range.
- ◎ We then remove the temporary print statements, since it affects the program's output.
- ◎ However, we can keep the assert statements.
 - They have no effect on program output when satisfied.

```
s = 'heart'
# Reverse the string.
r = ''
for i in range(-1, -(len(s) + 1), -1):
    r = r + s[i]
assert len(r) == len(s)
# Replace the last letter.
f = r[:-1] + '*'
assert len(f) == len(r)
print(f)
```

Assert Messages

- With **many asserts** it might become difficult to immediately **spot** what a failed assert means.
- For clarity you can **include** a **custom message** in an **assert** statement.
- The message is printed if the assertion **fails**.

```
>>> x = -5
>>> assert x > 0, 'expected x to be positive'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: expected x to be positive
```

Assert Messages

- ◎ Pro tip: You can use **f-strings** to include variable values in your assert message.

```
>>> speed = 3.5
>>> time = -4
>>> dist = speed * time
>>> assert dist > 0, f'expected dist to be positive, got {dist:.2f}'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: expected dist to be positive, got -14.00
```

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid grey and others are hollow with a grey outline. The lines connecting them are thin and grey, creating a dense, organic structure.

Testing Code

Why Test?

- ◎ Even the **best programmers** make **mistakes**.
 - Debugging is a normal part of programming.
- ◎ Making mistakes while programming generally isn't a problem *unless those mistakes go unnoticed*.
- ◎ Testing programs helps us find mistakes.
 - Yes, this includes **logic errors**.

Why Test?

- © Real-world programs can have complex algorithms and **behave differently** based on **different inputs**.
- © **Just because a program works for one specific input doesn't mean that it will work for all possible inputs.**
- © Comprehensive testing **increases** our **confidence** that software actually **works** correctly.

Software Testing

- ◎ **Software testing** is the process of investigating a program to ensure its quality.
- ◎ Good software testing will reveal any flaws in a program's logic.
- ◎ When testing **reveals** an error, it is the **programmer's job** to **modify** the program's **code to fix it**.

Black-Box Testing

- ◎ We will focus on a common form of software testing called **black-box testing**.
- ◎ The program is treated as a "black-box" which which **accepts *inputs*** and **produces *outputs***.
- ◎ **Inputs** are supplied, and the actual program **outputs** are compared to **expected** program **outputs**.
- ◎ Black-box testing is concerned with ***what*** a program does (the result), not the details of ***how*** it does it (the steps taken).

Designing Test Cases

- ◎ A **test case** describes what a program is expected to do given a particular set of inputs.
 - Consists of **defined inputs** and **expected outputs**.
- ◎ For example, a test case for a program which **squares** number might be:
 - Example **input**: 4
 - Expected **output**: 16

Designing Test Cases

- ◎ In general, it is **impossible** to test every possible input.
- ◎ So, in practice, we must **choose** a set of **test cases** to cover the behaviour of the program.
- ◎ Test design **techniques** can help us decide on which test cases to choose.
- ◎ We will consider **two** important test design techniques:
 - **Equivalence partitioning**, and
 - **Boundary-value** analysis.

Example: Income Tax Calculator

- ◎ To further explore test design, we will consider the example of an income tax calculator.
- ◎ The program is expected to take a dollar amount as **input**, and **output** the **amount of income tax** for that dollar amount.

Taxable income	Tax on this income
0 – \$18,200	Nil
\$18,201 – \$45,000	19 cents for each \$1 over \$18,200
\$45,001 – \$120,000	\$5,092 plus 32.5 cents for each \$1 over \$45,000
\$120,001 – \$180,000	\$29,467 plus 37 cents for each \$1 over \$120,000
\$180,001 and over	\$51,667 plus 45 cents for each \$1 over \$180,000

Source: [ATO resident tax rates 2020-21](#)

Equivalence Partitioning

- ⊙ **Equivalence partitioning** is a test design technique which involves **partitioning** all **possible** inputs into **equivalence classes** for which the software is expected to behave similarly.
- ⊙ One test case is created per **equivalence** class by selecting representative **input(s)** for that class.
- ⊙ For the tax calculator example, each tax **bracket** is an **equivalence** class.

Example: Income Tax Calculator

- From **equivalence partitioning**, we have created **5 test cases**.
 - Expected outputs are calculated manually.
- Notice that we select **input** values which are well within each **equivalence class**.

EQUIVALENCE CLASS	EXAMPLE INPUT	EXPECTED OUTPUT
0–\$18,200 bracket	5,000	0
\$18,201–\$45,000 bracket	35,000	3,192
\$45,001–\$120,000 bracket	90,000	19,717
\$120,001–\$180,000 bracket	150,000	40,567
\$180,001+ bracket	900,000	375,667

Boundary-Value Analysis

- ◎ **Boundary-value analysis** is a test design technique which involves **identifying inputs** which are **close** to the **boundaries** of equivalence classes.
- ◎ Test cases are created for the minimum and maximum **edges** of each equivalence class.

Boundary-Value Analysis

- ◎ Boundary-value analysis helps to catch errors which may arise from things like:
 - Including/omitting equality in a comparison.
 - ◎ e.g. using `<=` instead of `<`.
 - Off-by-one errors
 - ◎ e.g. using `range(1, n)` instead of `range(1, n+1)`.

Example: Income Tax Calculator

EQUIVALENCE CLASS	BOUNDARY	EXAMPLE INPUT	EXPECTED OUTPUT
0–\$18,200 bracket	Minimum	0	0
	Maximum	18,200	0
\$18,201–\$45,000 bracket	Minimum	18,201	0.19
	Maximum	45,000	5,092
\$45,001–\$120,000 bracket	Minimum	45,001	5,092.325
	Maximum	120,000	29,467
\$120,001–\$180,000 bracket	Minimum	120,001	29,467.37
	Maximum	180,000	51,667
\$180,001+ bracket	Minimum	180,001	51,667.45

Check Your Understanding

Q. How many **equivalence classes** are there for the shown task definition?

Task definition

Create a train ticketing program which displays the kind of fare that a passenger must pay. If the passenger is age 12 or under, they are considered a child. If the passenger is age 65 or older, they are considered a senior. Everyone else is full fare.

Check Your Understanding

Q. How many **equivalence classes** are there for the shown task definition?

A. 3.

- Ⓐ Children (age ≤ 12)
- Ⓑ Seniors (age ≥ 65)
- Ⓒ Everyone else ($12 < \text{age} < 65$)

Task definition

Create a train ticketing program which displays the kind of fare that a passenger must pay. If the passenger is age 12 or under, they are considered a child. If the passenger is age 65 or older, they are considered a senior. Everyone else is full fare.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting different levels of connectivity or importance. The lines are thin and gray, creating a mesh-like structure.

Summary

In This Lecture We...

- © Learnt the three main kinds of software errors: **syntax errors**, **runtime errors**, and **logic errors**.
- © Used **assert statements** to expose logic errors using runtime errors.
- © Explored techniques for testing software to reveal errors.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. Some nodes are highlighted with blue circles, and others with blue dots.

Lecture 4.3

Exceptions



Lecture Overview

1. Exceptions and Stack Traces
2. Handling Exceptions
3. Raising Exceptions

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting different levels or types of nodes. The lines are thin and gray, connecting the nodes in a non-linear fashion.

Exceptions and Stack Traces

Exceptions

- ⊙ An **exception** is a **runtime error** that can be recovered from.
- ⊙ We say that code which produces an exception "**raises** an **exception**".
- ⊙ We say that code which anticipates and recovers from an exception "**handles** an exception".

Exceptions

- ◎ An exception which is **not** handled will typically cause the program to **crash** and display information about the exception.
- ◎ Programmers can anticipate exceptions when writing code and explicitly handle them.
 - This can be done in such a way that the runtime error will not cause the program to crash.

Types of Exceptions

- ◎ Python uses different types of **exceptions** for different kinds of **runtime errors**.
- ◎ Examples include:
 - **ZeroDivisionError** (e.g. dividing a number by zero).
 - **IndexError** (e.g. using an index not in a list).
 - **KeyError** (e.g. using a key not in a dictionary).
 - **NameError** (e.g. using an undefined variable).
 - **FileNotFoundError** (e.g. opening a file that doesn't exist for reading).

Exception Messages

- ◎ In addition to its **type**, an exception can also carry a **message** with more **information**.
- ◎ The **type** and **message** will be displayed together when an exception is not handled.

```
>>> open('i_dont_exist.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'i_dont_exist.txt'
```

Type

Message

Exception Messages

- ◎ But what is this stuff here?
- ◎ To understand the first part of the exception output, we must first learn about the **call stack**.

```
>>> open('i_dont_exist.txt')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
FileNotFoundError: [Errno 2] No such file or directory: 'i_dont_exist.txt'
```

← ???

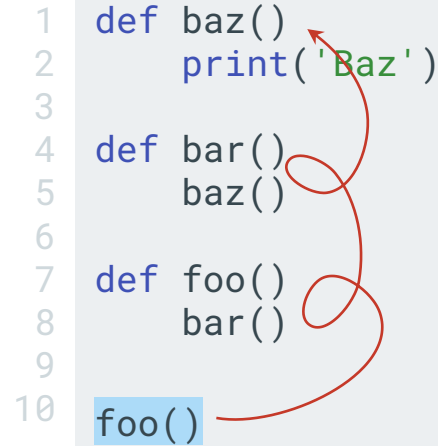
The Call Stack

- ◎ Recall that when a function is called, the **control** flow **moves** into that **function**.
- ◎ When the function is **finished**, control flow moves **back** to the place that the function was called from (the **call site**).

```
# This is a function.  
def print_greeting():  
    print('Hello!')  
  
# This is a call site.  
print_greeting()  
  
# This is a second call site.  
print_greeting()
```

The Call Stack

- Functions can call other functions, resulting in a **call stack** describing how to return to each call site.
- In the example shown, line 10 calls `foo` which calls `bar` which calls `baz`.
 - The call stack keeps track of all of these locations so that Python knows where to go when each function finishes.



```
1 def baz()  
2     print('Baz')  
3  
4 def bar()  
5     baz()  
6  
7 def foo()  
8     bar()  
9  
10 foo()
```

Stack Traces

- ◎ The confusing-looking part of the exception output starting with "Traceback" is called a **stack trace**.
- ◎ The stack trace is a text-based description of the call stack at the time an exception is raised.

```
>>> open('i_dont_exist.txt')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
FileNotFoundError: [Errno 2] No such file or directory: 'i_dont_exist.txt'
```

Stack trace

Reading Stack Traces

- ◎ Stack traces describe the **chain** of call sites that lead to the **exception**, in order.
- ◎ Each entry in a **stack** trace "zooms in" further on the source of the exception.
- ◎ The place where the exception is first raised is shown **last**.
- ◎ The true cause of the error may be more evident on some **levels** than on others.
 - Which part of the stack trace you focus your attention on will **change depending** on the program.

Example: Reading Stack Traces

```
1 def get_apple_count(fruit):  
2     return fruit['apples']  
3  
4 def print_apple_count(fruit):  
5     count = get_apple_count(fruit)  
6     print(f'Apple count: {count}')7  
8 fruit = {'pears': 2, 'bananas': 7}  
9 print_apple_count(fruit)
```

```
Traceback (most recent call last):  
  File "fruit.py", line 9, in <module>  
    print_apple_count(fruit)  
  File "fruit.py", line 5, in print_apple_count  
    count = get_apple_count(fruit)  
  File "fruit.py", line 2, in get_apple_count  
    return fruit['apples']  
KeyError: 'apples'
```

- ◎ The code (above left) is written in a script file called "fruit.py" and crashes with an error message (above right) when run.

Example: Reading Stack Traces

```
1 def get_apple_count(fruit):  
2     return fruit['apples']  
3  
4 def print_apple_count(fruit):  
5     count = get_apple_count(fruit)  
6     print(f'Apple count: {count}')7  
8 fruit = {'pears': 2, 'bananas': 7}  
9 print_apple_count(fruit)
```

```
Traceback (most recent call last):  
  File "fruit.py", line 9, in <module>  
    print_apple_count(fruit)  
  File "fruit.py", line 5, in print_apple_count  
    count = get_apple_count(fruit)  
  File "fruit.py", line 2, in get_apple_count  
    return fruit['apples']  
KeyError: 'apples'
```

- ◎ The exception type and message gives us information about the error, but does not tell us where to look in the code.

That information is contained in the stack trace.

Example: Reading Stack Traces

```
1 def get_apple_count(fruit):  
2     return fruit['apples']  
3  
4 def print_apple_count(fruit):  
5     count = get_apple_count(fruit)  
6     print(f'Apple count: {count}')  
7  
8 fruit = {'pears': 2, 'bananas': 7}  
9 print_apple_count(fruit)
```

```
Traceback (most recent call last):  
  File "fruit.py", line 9, in <module>  
    print_apple_count(fruit)  
  File "fruit.py", line 5, in print_apple_count  
    count = get_apple_count(fruit)  
  File "fruit.py", line 2, in get_apple_count  
    return fruit['apples']  
KeyError: 'apples'
```

- ◎ The last entry in the stack trace tells us where the exception is raised from.
 - In this case, it's a statement in `get_apple_count`.
But where was `get_apple_count` called from?

Example: Reading Stack Traces

```
1 def get_apple_count(fruit):  
2     return fruit['apples']  
3  
4 def print_apple_count(fruit):  
5     count = get_apple_count(fruit)  
6     print(f'Apple count: {count}')  
7  
8 fruit = {'pears': 2, 'bananas': 7}  
9 print_apple_count(fruit)
```

```
Traceback (most recent call last):  
  File "fruit.py", line 9, in <module>  
    print_apple_count(fruit)  
  File "fruit.py", line 5, in print_apple_count  
    count = get_apple_count(fruit)  
  File "fruit.py", line 2, in get_apple_count  
    return fruit['apples']  
KeyError: 'apples'
```

- ◎ Moving up in the stack trace, we can find where `get_apple_count` was called from when the error occurred.
 - In this case, it's a statement in `print_apple_count`.
But where was `print_apple_count` called from?

Example: Reading Stack Traces

```
1 def get_apple_count(fruit):  
2     return fruit['apples']  
3  
4 def print_apple_count(fruit):  
5     count = get_apple_count(fruit)  
6     print(f'Apple count: {count}')7  
8 fruit = {'pears': 2, 'bananas': 7}  
→ print_apple_count(fruit)
```

```
Traceback (most recent call last):  
  File "fruit.py", line 9, in <module>  
    print_apple_count(fruit)  
  File "fruit.py", line 5, in print_apple_count  
    count = get_apple_count(fruit)  
  File "fruit.py", line 2, in get_apple_count  
    return fruit['apples']  
KeyError: 'apples'
```

- ⊙ Moving up in the stack trace, we can find where `print_apple_count` was called from when the error occurred.
- ⊙ In this case, we might find this to be a particularly useful place to look since we can see that the fruit dict does not contain apples.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid grey and others are hollow with a grey outline. The lines connecting them are thin and grey, creating a dense, organic structure.

Handling Exceptions

Exception Handling

- ◎ In Python, exceptions can be handled using the `try` and `except` keywords.
- ◎ The `try` block surrounds the statements which may raise an exception.
- ◎ The `except` block contains statements to execute if an exception is `raised` in the associated `try` block.
- ◎ When an exception is `raised` in the `try` block, control flow `immediately` moves to an associated `except` block.

Example: Handling Invalid User Input

```
1 # File: age.py
2 age = int(input('Enter your age: '))
3 print('Thank you.')
```

```
$ python age.py
Enter your age: twenty-five
Traceback (most recent call last):
  File "age.py", line 2, in <module>
    age = int(input('Enter your age: '))
ValueError: invalid literal for int() with base 10: 'twenty-five'
```

- © When the user enters text that can't be converted into an integer, the program will crash.

Example: Handling Invalid User Input

```
1 # File: age.py
2 while True:
3     try:
4         age = int(input('Enter your age: '))
5         break
6     except:
7         print('Invalid input, try again please.')
8 print('Thank you.')
```

- ◎ When line 4 raises an exception, control flow will immediately jump to the `except` block.
 - The `break` statement will not be executed when this happens, causing the loop to repeat.

Example: Handling Invalid User Input

```
1 # File: age.py
2 while True:
3     try:
4         age = int(input('Enter your age: '))
5         break
6     except:
7         print('Invalid input, try again please.')
8 print('Thank you.')
```

```
$ python age.py
Enter your age: twenty-five
Invalid input, try again please.
Enter your age: 25.0
Invalid input, try again please.
Enter your age: 25
Thank you.
```

Handling Specific Exception Types

- ◎ By default, an **except block** handles any type of **exception**.
- ◎ If you **specify** the type of exception to be handled, the except block will **only** handle exceptions of that **type**.
 - Other types of exceptions will **not** be handled by that block.
- ◎ You can specify multiple except blocks for one try block.
- ◎ Remember, the output produced by unhandled exceptions will tell you their **type**.

Example: Reciprocal Calculator

```
while True:
    try:
        denom = int(input('Enter a number: '))
        result = str(1 / denom)
        break
    except:
        print('Invalid input, or perhaps division by zero?')
print(f'The reciprocal of {denom} is {result}.')
```

- ⊙ There are two kinds of errors that could occur (**invalid** conversion to an **integer** or **division** by **zero**).

Currently we can't tell them apart (they are all handled by the same except block).

Example: Reciprocal Calculator

```
while True:
    try:
        denom = int(input('Enter a number: '))
        result = str(1 / denom)
        break
    except ZeroDivisionError:
        result = 'infinity'
        break
    except ValueError:
        print('Invalid input, try again please.')
print(f'The reciprocal of {denom} is {result}.')
```

With multiple except blocks that handle different types of exceptions, we can handle the errors with different code.

Example: Reciprocal Calculator

```
while True:
    try:
        denom = int(input('Enter a number: '))
        result = str(1 / denom)
        break
    except ZeroDivisionError:
        result = 'infinity'
        break
    except ValueError:
        print('Invalid input, try again please.')
print(f'The reciprocal of {denom} is {result}.')
```

```
Enter a number: five
Invalid input, try again please.
Enter a number: 0
The reciprocal of 0 is infinity.
```

To Check or to Handle?

- ◎ There are two different principles you can follow for dealing with runtime errors: **EAFP** and **LBYL**.

To Check or to Handle?

- ◎ **EAFP** = "it's **e**asier to **a**sk for forgiveness than **p**ermission".
- ◎ This means that you write code that attempts to do something, handling exceptions using try/except blocks.

- ◎ **LBYL** = "**l**ook **b**efore **y**ou **l**eap".
- ◎ This means that you write code which performs a check before attempting to do something.

Example: Reading a File

```
# EAFP-style code.  
filename = input('Filename: ')  
try:  
    file = open(filename)  
    contents = file.read()  
    print(contents)  
except FileNotFoundError:  
    print('File not found.')
```

```
# LBYL-style code.  
import os  
filename = input('Filename: ')  
if os.path.isfile(filename):  
    file = open(filename)  
    contents = file.read()  
    print(contents)  
else:  
    print('File not found.')
```

- ◎ These programs essentially do the same thing.
- ◎ The "official" recommendation in Python is to follow EAFP.
- ◎ My recommendation is to follow your own preference on a case-by-case basis.

Check Your Understanding

Q. How would you rewrite the following snippet of code to take a LBYL (“look before you leap”) approach?

```
denom = int(input('Enter a number: '))  
try:  
    result = str(1 / denom)  
except ZeroDivisionError:  
    result = 'infinity'
```

Check Your Understanding

Q. How would you rewrite the following snippet of code to take a LBYL (“look before you leap”) approach?

```
denom = int(input('Enter a number: '))
try:
    result = str(1 / denom)
except ZeroDivisionError:
    result = 'infinity'
```

A. An LBYL equivalent is as follows:

```
denom = int(input('Enter a number: '))
if denom != 0:
    result = str(1 / denom)
else:
    result = 'infinity'
```

- ⊙ A `ZeroDivisionError` only occurs when `denom` is 0.
- ⊙ So we can check whether `denom` is 0 instead of handling that exception.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid grey and others are hollow with a grey outline. The lines connecting them are thin and grey, creating a dense, organic structure.

Raising Exceptions

Raise Statements

- ◎ In Python, exceptions can be raised using the `raise` keyword.
- ◎ This gives us access to Python's exception system for our own sources of errors.
- ◎ When writing your own functions, you should consider raising a `ValueError` for **invalid** argument values.
 - This will help you track down errors more easily later on.

Example: Factorials

- ◎ The "factorial" of an integer is the result of multiplying itself with all positive integers less than it.
 - e.g. "5 factorial" is $5 \times 4 \times 3 \times 2 \times 1 = 120$.
- ◎ "0 factorial" is defined as 1.
- ◎ The factorial of a negative number is not defined.
- ◎ Let's write our own function for calculating factorials.

Example: Factorials

```
def factorial(n):  
    result = 1  
    while n > 1:  
        result = result * n  
        n = n - 1  
    return result
```

- ◎ This code works, but **can** be called with **negative** numbers (which are not valid).
- ◎ We would like to know when our function is called with **invalid input** so that we can fix the problem in the calling code.

Example: Factorials

```
def factorial(n):  
    if n < 0:  
        raise ValueError('negatives not allowed')  
    result = 1  
    while n > 1:  
        result = result * n  
        n = n - 1  
    return result
```

- ◎ Here we raise a **ValueError** if the input to our factorial function is negative.
- ◎ This will tell us when our function is being used incorrectly.

Raising Exceptions of Different Types

- ◎ Of course, you can raise other types of exceptions (not just **ValueError**).
- ◎ The type of the exception is mainly for conveying meaning about the nature of the error.
- ◎ **ValueError** implies that an **invalid** value was encountered.
- ◎ Raise a **RuntimeError** when you are **unsure** of which type of exception to raise.

Check Your Understanding

Q. What is the output of the shown Python program?

```
def print_odd_only(n):  
    if n % 2 == 0:  
        raise ValueError(f'{n} is even!')  
    print(n)  
  
try:  
    print_odd_only(5)  
    print_odd_only(4)  
    print_odd_only(3)  
except ValueError:  
    print('Oops')  
  
print('Done')
```

Check Your Understanding

Q. What is the output of the shown Python program?

A. 5, Oops, Done.

- ⦿ `print_odd_only(5)` displays an output of 5.
- ⦿ `print_odd_only(4)` raises a **ValueError**. **Control** flow jumps directly to the **except** block which prints **Oops**.
- ⦿ **Since the exception was handled, the program does not crash.** Therefore **Done** is printed also.

```
def print_odd_only(n):  
    if n % 2 == 0:  
        raise ValueError(f'{n} is even!')  
    print(n)  
  
try:  
    print_odd_only(5)  
    print_odd_only(4)  
    print_odd_only(3)  
except ValueError:  
    print('Oops')  
  
print('Done')
```

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting different levels of connectivity or importance. The lines are thin and gray, creating a mesh-like structure.

Summary

In This Lecture We...

- ◎ Learnt how to read Python **stack traces**.
- ◎ **Handled exceptions** using try-except structures to recover from expected runtime errors.
- ◎ **Raised exceptions** from our own code.

Next Lecture We Will...

- ◎ Import additional functionality into our programs using **modules**.

Thanks for your attention!

The slides and lecture recording will be made available on LMS.

The [“Cordelia” presentation template](#) by [Jimena Catalina](#) is licensed under [CC BY 4.0](#).
PPT Acknowledgement: Dr Aiden Nibali, CS&IT LTU.