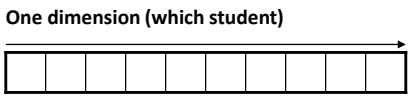
Composite Types, Lists Part 2

- When to use multi-dimensional lists
- Creating 2D lists
- How to access a 2D list and its parts
- Basic 2D list operations: display, accessing parts, copying the list
- Other composites: strings and tuples

When To Use Lists Of Different Dimensions

- It's determined by the data – the number of categories of information determines the number of dimensions to use.
- Examples:
- (1D list)
  - –Tracking grades for a class (previous example)
  - –Each cell contains the grade for a student i.e., `grades[i]`
  - –There is one dimension that specifies which student's grades are being accessed



- (2D list)
  - –Expanded grades program (table: grades for multiple lectures)
  - –Again there is *one dimension* that specifies which student's grades are being accessed
  - –The *other dimension* can be used to specify the lecture section

When To Use Lists Of Different Dimensions (2)

- (2D list continued)

Student

Lecture section

	First student	Second student	Third student	…
L01				
L02				
L03				
L04				
L05				
:				
L0N				

When To Use Lists Of Different Dimensions (3)

- (2D list continued)
- Notice that each row is merely a 1D list
- (A 2D list is a list containing rows of 1D lists)

Important:

- List elements are specified in the order of `[row] [column]`
- Specifying only a single set of brackets specifies the row

Columns (e.g. grades)

	[0]	[1]	[2]	[3]
[0] L01				
[1] L02				
[2] L03				
[3] L04				
[4] L05				
[5] L06				
[6] L07				

Rows (e.g. lecture section)

Creating And Initializing A Multi-Dimensional List In Python (Fixed Size During Creation)

General structure

```
<list_name> = [ [<value 1>,  <value 2>, ... <value n>],
                [<value 1>,  <value 2>, ... <value n>],
                     : :        :
                     : :        :
                [<value 1>,  <value 2>, ... <value n>] ]
```

Rows

Columns

Creating And Initializing A Multi-Dimensional List In Python (2): Fixed Size During Creation

Name of the example program: `1display2DList.py`

Learning: creating, displaying a fixed size 2D list

```
table = [ [0, 0, 0],
          [1, 1, 1],
          [2, 2, 2],
          [3, 3, 3]]
```

```
      c=0  c=1  c=2
r = 0 [0,  0,  0]
r = 1 [1,  1,  1]
r = 2 [2,  2,  2]
r = 3 [3,  3,  3]
```

```
for r in range (0, 4, 1):
   print (table[r])  #Each call to print displays a 1D list
```

```
for r in range (0,4,1):
   for c in range (0,3,1):
       print(table[r][c], end="")
   print()          #Displays a list element
```

```
    0 1 2 (col)
r = 0 000
r = 1 111
r = 2 222
r = 3 333
```

```
print(table[2][0]) #Displays 2 not 0
```

```
2
```

2D Lists: Levels Of Access

```
table = [ [0, 0, 0],
          [1, 1, 1],
          [2, 2, 2],
          [3, 3, 3]]
print(table)  #Entire list
print(table[0]) #First row
print(table[3][1]) #4th row, 2nd column
print(table[0][0][0]) #What does this do?
```

```
[[0, 0, 0], [1, 1, 1], [2, 2, 2], [3, 3, 3]]
[0, 0, 0]
3
TypeError: 'int' object is not subscriptable
```

```
table = [ [["a","b"], 0, 0],
          [1, 1, 1],
          [2, 2, 2],
          [3, 3, 3]]

print(table[0][0][0]) #Now what does this do?
```

James Tam

In Python: List Elements Need Not Store The Same Data Type

- This is one of the differences between Python lists and arrays in other languages
- Example:

```
aList = [False, "James", "Tam", "210-9455", 707, 10.5]
```

Copying Lists

- Important: A variable that appears to be a list is really a reference to a list.
  - Recall: the reference and the list are two separate memory locations!

```
matrix = [ [0, 0, 0],
           [1, 1, 1],
           [2, 2, 2],
           [3, 3, 3]]
```

  - Wrong way to 'copy' a 2D list

```
aList1 = aList2
```

(Why is this wrong? Hint: recall what is stored in `aList1` and `aList1`)

James Tam

Copying Lists: Example

- **Name of the example program:** `2copyingListsBothWays.py`
- This is the **wrong way**.

```
aGrid1 = create()
aGrid2 = aGrid1
aGrid1[3][3] = "!"
print("First list")
display(aGrid1)
print("Second list")
display(aGrid2)
```

```
# FYI:
def create():
    aGrid = [["*","*","*","*"],
             ["*","*","*","*"],
             ["*","*","*","*"],
             ["*","*","*","*"]]
    return(aGrid)
```

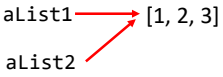
James Tam

New Terminology

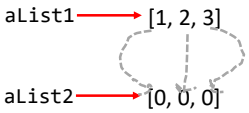
- **Shallow copy**: copies what's stored in the reference (location of a list).

Code

```
aList1 = [1,2,3]
aList2 =aList1
```



- **Deep copy**: copies the data from one list to another.
  - Create a new list e.g. aList2 = [0]*3
  - Copy each piece of data (list elements) from one list to another e.g. `aList2[0] = aList1[0]`



James Tam

Copying Lists: Example (2)

- This is the right way.

```
aGrid1 = create()
aGrid2 = create()
copy(aGrid1,aGrid2)
```

```
def copy(destination,source):
    for r in range (0,SIZE,1):
        for c in range (0,SIZE,1):
            destination[r][c] = source[r][c]
```

```
copy(aGrid1,aGrid2)
aGrid1[0][0] = "?" #These statements prove there's two lists
aGrid1[3][3] = "?"
print("First list")
display(aGrid1)
print("Second list")
display(aGrid2)
```

James Tam

Copying Lists: Write The Code Yourself

- For this class you should not use some else's pre-created list `copy` method (e.g. those defined when you "`import copy`")
- Not all programming languages have this capability (you will need to know how to do it yourself).
- Writing the code yourself will provide you with extra practice and help you become more familiar with list (in other languages 'array') operations.

James Tam

Extra Practice

List operations:

- For a numerical list: implement some common mathematical functions (e.g., average, min, max, mode – last one is challenging).
- For any type of list: implement common list operations (e.g., displaying all elements one at a time, inserting elements at the end of the list, insert elements in order, searching for elements, removing an element, finding the smallest and largest element).
  - In order to develop your programming skills you should write the code yourself rather than using predefined python methods such as `append`, `min`, `max` etc.

After This Sub-Section You Should Now Know

- When to use lists of different dimensions
- Basic operations on a 2D list
- How to create a 2D list: fixed size and by dynamically creating it
- How to access a 2D list: the whole list, rows in the list and individual elements
- Python lists need not be homogenous (contain the same type of element)
- How to properly copy the contents of a 2D list into another 2D list as well as a common mistake when copying lists

Composite Types: Other Composites

You will learn how to create new variables that are collections of other entities: strings (character composite), tuples (similar to a list but immutable)

ASCII Values (Reminder)

- Each character is assigned an ASCII code e.g., 'A' = 65, 'b' = 98
- The `chr()` function can be used to determine the character (string of length one) for a particular ASCII code (number to character)
- The `ord()` function can be used to determine the ASCII code for a 'character' - string of length one (character to number)
- **Name of the example program**: `1ascii.py`
  - Learning: converting to/from ASCII codes to the equivalent character.

```
aChar = input("Enter a character whose ASCII value that you wish to
  see: ")
print("ASCII value of %s is %d" %(aChar,ord(aChar)))
aCode = int(input("Enter an ASCII code to convert to a character: "))
print("The character for ASCII code %d is %s" %(aCode,chr(aCode)))
```

```
Enter a character whose ASCII value that you wish to see: A
ASCII value of A is 65
Enter an ASCII code to convert to a character: 66
The character for ASCII code 66 is B
```

String: Composite

- Strings are just a series of characters (e.g., alpha, numeric, punctuation etc.)
  - Like a list a string is:
    - A composite type (can be treated as one entity or individual parts can be accessed).
  - **Name of example**: "`2stringComposite.py`"
    - Learning: strings are composite, how to access the entire composite string and how to access individual elements

```
aString1 = "hello"
print("Whole string %s" %(aString1))
print("Sub string %s-%s" %(aString1[1],aString1[4]))
```

```
Whole string hello
Sub string e-o
```

Passing Strings As Parameters

- A string is composite so either the entire string or just a sub-string can be passed as a parameter.
- **Name of example**: `3stringParameters.py`
  - Learning: How to pass a string (or substring) to a function.

```
def fun1(str1):
    print("Inside fun1 %s" %(str1))

def fun2(str2):
    print("Inside fun2 %s" %(str2))

def start():
    str1 = "abc"
    print("Inside start %s" %(str1))
    fun1(str1)
    fun2(str1[1])
```

```
Inside start abc
Inside fun1 abc
Inside fun2 b
```

Passing whole string

Passing part of a string

James Tam

Mutable, Constant, Immutable,

- Mutable types:
  - The original memory location *can* change

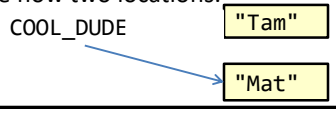
```
num = 12
num = 17
```

num | 17

- Constants
  - Memory location *shouldn't* change (Python): may produce a logic error if modified e.g. `GST_RATE = 0.05`
  - Memory location syntactically *cannot* change (C++, Java): produces a syntax error (violates the syntax or rule that constants cannot change)

- Immutable types:
  - The *original* memory location *won't change*
  - Changes to a variable of a pre-existing immutable type creates a new location in memory. There are now two locations.

```
COOL_DUDE = "Tam"
COOL_DUDE = "Mat"
```



James Tam

Lists Are Mutable

- **Example**

```
aList = [1,2,3]
aList[0] = 10
print(aList)  # [10,2,3]
```

The original list can change (modifying an element) making this type mutable

James Tam

Strings Are Immutable

- Even though it may look a string can change they actually cannot be edited (original memory location cannot change).
- **Name of the example program**: `4immutableStrings.py`
  - Learning: strings are immutable:
    - Using the assignment operator in conjunction with the name of the whole string produces a new string (string variable refers to a new string not the original string).
    - Attempting to modify a string produces an error.

```
s1 = "hi"
print(s1)
s1 = "bye"      # New string created
print(s1)
s1[0] = "G"     # Error
```

hi

bye

Cannot modify the characters in a string (immutable)

```
Traceback (most recent call last):
  File "12immutableStrings.py", line 7, in <module>
    s1[0] = "G"
TypeError: 'str' object does not support item assignment
```

Substring Operations

- Sometimes you may wish to extract out a portion of a string.
  - E.g., Extract first name “James” from a full name “James T. Kirk, Captain”
- This operation is referred to as a ‘substring’ operation in many programming languages.
- There are two implementations of the substring operation in Python:
  - String slicing
  - String splitting

1 The name James T. Kirk is © CBS

String Slicing

Included in the slice

Excluded in the slice

- Slicing a string will return a portion of a string based on the indices provided
- The index can indicate the start (include) and end point (exclude) of the substring.
- **Format**:
  string_name [*start_index* : *end_index*]
- **Name of example**: `5stringSlicing.py`
  - Learning: how the slicing operator works


```
aString = "abcdefghij"
print(aString)
temp = aString[2:5]
print(temp)
temp = aString[:5]
print(temp)
temp = aString[7:]
print(temp)
```

```
abcdefghij
cde
abcde
hij
```

From start to the end (exc)

From 7 (included) until the end

Example Use: String Slicing

- Where characters at fixed positions must be extracted.
- Example: area code portion of a telephone number

  "403-210-9455"

  –The "403" area code could then be passed to a data base lookup to determine the province.

James Tam

String Splitting

- Divide a string into portions with a particular character determining where the split occurs.
- Practical usage
  - The string "The cat in the hat" could be split into individual words (split occurs when spaces are encountered).
  - "The" "cat" "in" "the" "hat"
  - Each word could then be individually passed to a spell checker.

String Splitting (2)

- **Format**:
  string_name.split ('<character used in the split')
- **Online example**: 6stringSpliting.py
  - Learning: how the slicing operator works.

```
aString = "man who smiles"
 # Default split character is a space
 one, two, three = aString.split()
 print(one)
 print(two)
 print(three)
 aString = "James,Tam"
 first, last = aString.split(",")
 nic = first + " \"The Bullet\" " + last
 print(nic)
```

```
man
who
smiles
```

```
James "The Bullet" Tam
```

James Tam

String Testing Functions[1]

- These functions test a string to see if a given condition has been met and return either "`True`" or "`False`" (Boolean).
- **Format**:
  string_name.function_name()

1 These functions will return false if the string is empty (less than one character).

String Testing Functions (2)[1]

Boolean Function	Description
`isalpha()`	Only true if the string consists only of alphabetic characters.
`isdigit()`	Only returns true if the string consists only of digits.
`isalnum()`	Only returns true if the string is composed only of alphabetic characters or numeric digits (alphanumeric)
`islower()`	Only returns true if the alphabetic characters in the string are all lower case.
`isspace()`	Only returns true if string consists only of whitespace characters (" ", "\n", "\t")
`isupper()`	Only returns true if the alphabetic characters in the string are all upper case.

1 Each one of this functions ('method') must be preceded by a string variable and a dot e.g. `aStr.isalpha()` `#where aStr refers to a string`

Applying A String Testing Function

Name of the example: 7stringTestFunctions.py

- Learning: using the `isdigit()` function to check for invalid types (float instead of integer)

```
ok = False
while(ok == False):
   temp = input("Enter an integer: ")
   ok = temp.isdigit()
   if (ok == False):
      print(temp, "is not an integer")
num = int(temp)
num = num + num
print(num)
```

```
Enter an integer: abc
abc is not an integer
```

```
Enter an integer: 11.2
11.2 is not an integer
```

```
Enter an integer: 12
24
```

Heuristic (end of "loops") applied also (good error message)

Functions That Return Modified Copies Of Strings (IF There Is Time)[1]

- These functions return a modified version of an existing string (leaves the original string intact). Common whitespace characters = `sp, tab, enter`

Function	Description
`lower()`	Returns a copy of the string with all the alpha characters as lower case (non-alpha characters are unaffected).
`upper()`	Returns a copy of the string with all the alpha characters as upper case (non-alpha characters are unaffected).
`strip()`	Returns a copy of the string with all leading and trailing whitespace characters removed.
`lstrip()`	Returns a copy of the string with all leading (left) whitespace characters removed.
`rstrip()`	Returns a copy of the string with all trailing (right) whitespace characters removed.
`lstrip(char)`	Returns a copy of the string with all leading instances of the character parameter removed.
`rstrip(char)`	Returns a copy of the string with all trailing instances of the character parameter removed.

1 Each one of this functions ('method') must be preceded by a string variable and a dot e.g. `aStr.lower()  #where aStr refers to a string`

Examples: Functions That Return Modified Copies (IF There Is Time)

Name of the example program: `8stringModificationFunctions.py`

`Learning: learning how common string functions operate`

```
aString = "talk1! AbouT"
print(aString)
aString = aString.upper()
print(aString)

aString = "xxhelxlo therex"
print(aString)
aString = aString.lstrip("x")
print(aString)
aString = "xxhellx thxrx"
aString = aString.rstrip("x")
print(aString)
```

```
talk1! AbouT
TALK1! ABOUT
xxhelxlo therex
helxlo therex
xxhellx thxr
```

Tuples

- Much like a list, a tuple is a composite type whose elements can consist of any other type.
- Tuples support many of the same operators as lists such as indexing.
- However tuples are immutable.
- Like lists each element of a tuple is not confined to characters (string of length 1).
- But unlike a list a tuple is immutable.
  - It stores data that **should not change**.
  - In that way it's somewhat analogous to a named constant (e.g. `PI = 3.14`) but unlike this named constant changes can only produce a new tuple.

Creating Tuples

- **Format**:

  tuple_name = (*value*[1], *value*[2]...*value*[n])

- **Example**:

```
tup = (1,2,"foo",0.3)
```

A Small Example Using Tuples

- **Name of the online example**: `9simpleTupleExample.py`
  - Learning: accessing an entire tuple, accessing individual elements, tuples are an immutable type.

```
tup = (1,2,"foo",0.3)
print(tup)
print(tup[2])
tup[2] = "bar"
```

```
(1, 2, 'foo', 0.3)
foo
```

Error (trying to change an immutable):
"TypeError: object does not support item assignment"

Function Return Values

- Although it appears that functions in Python can return multiple values they are in fact consistent with how functions are defined in other programming languages.
- Functions can either return zero or *exactly one value* only.
- Specifying the return value with brackets merely returns one tuple back to the caller.

```
def fun():
    return(1,2,3)
```

Returns: A tuple with three elements

```
def fun(num):
    if (num > 0):
        print("pos ")
        return()
    elif (num < 0):
        print("neg")
        return()
```

Nothing is returned back to the caller (empty tuple)

Functions Changing Multiple Items

- Because functions only return 0 or 1 items (Python returns one composite) the mechanism of passing by reference (covered earlier in this section) is an important concept.
  - What if more than one change must be communicated back to the caller (only one entity can be returned).
  - Multiple changes to parameters (>1) **must** be passed by reference.

Proving That Python Functions Return A Tuple

- **Name of the online example**: `10functionReturnValues.py`
  - Learning:
    - Demonstrating functions return tuples
    - Iterating a tuple using loops: for, while.

```
def fun():
    tupleInFun = (1.5,2,7,0.3)
    return(tupleInFun)

def start():
    tupleInStart = fun()
    print("Iterating using a for-loop in conjunction with
      the 'in' operator")
    for element in tupleInStart:
        print("%.1f" %(element))
```

James Tam

Proving That Python Functions Return A Tuple (2)

```
print()
i = 0
numElements = len(tupleInStart)
print("Iterating using a while-loop in conjunction with
  the len() function")
while (i < numElements):
    print("%.1f" %(tupleInStart[i]))
    i = i + 1
```

James Tam

Extra Practice

String:

- Write the code that implements string operations (e.g., splitting) or string functions (e.g., determining if a string consists only of numbers)

After This Section You Should Now Know

- What is the difference between a mutable and an immutable type
- How strings are actually a composite type
- Common string functions and operations
- How a tuple is a composite, immutable type.
- Iterating tuples using for and while loops

After This Section You Should Now Know (2)

- When to use lists of different dimensions
- Basic operations on a 2D list
- What is a tuple, common operations on tuples such as creation, accessing elements, displaying a tuple or elements
- How functions return zero or one item
- What is a reference and how it differs from a regular variable
- Why references are used
- The two parameter passing mechanisms: pass-by-value and pass-by-reference